50376
1993
107

NUMERO D'ORDRE : **1054**

50376
1993
107

ANNEE : 1993

USTL

LIFL

U.R.A. 369 du C.N.R.S.

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

50376
1993
107

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Mohand Tahar KECHADI

Un modèle de fonctionnement désordonné pour les systèmes multiprocesseurs pipelines vectoriels à mémoires partagées

---

Définition, modélisation et proposition d'architecture

UNIVERSITE DES Sc. et Techn. BIBLIOTHEQUE UNIVERSITAIRE LILLE

Thèse soutenue le 01 Mars 199[illegible] devant la commission d'examen :

Président :	V. CORDONNIER	LIFL
Directeur de Thèse	J-L. DEKEYSER	LIFL
Rapporteurs :	D. LITAIZE	IRIT
	B. PLATEAU	IMAG
Examinateurs :	Ph. PREUX	LIFL
	Ph. TESSON	Cray -- France

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

Table des matières

Introduction

Nombreuses sont les applications, dans divers domaines, qui demandent une puissance de calcul de plus en plus grande. La recherche de cette puissance de calcul nécessite le développement de calculateurs parallèles très puissants. Les machines multiprocesseurs vectorielles à mémoire partagée occupent une place importante parmi les supercalculateurs les plus puissants. Malheureusement sur des applications réelles les performances théoriques (puissance de crête) sont loin d'être atteintes et ceci pour les deux raisons suivantes :

1) La sous-utilisation de toutes les ressources de la machine, par exemple :

- le faible pourcentage des opérations vectorielles dans la majorité des applications,
- le faible taux de tâches parallélisables qui réduit le nombre de processeurs travaillant en parallèle [Hockney88],
- le faible taux d'utilisation du mécanisme de chaînage.

2) La seconde raison que nous qualifions de plus importante est la faible capacité de transfert de données entre les processeurs et la mémoire. Elle a un impact direct sur les performances de ces machines [Potier87]. L'utilisation des mémoires parallèles permet d'augmenter cette capacité, nous pouvons exécuter autant d'accès concurrents qu'il y a de mémoires. Cependant ces mémoires sont sujettes à de dramatiques chutes de performances dues aux conflits d'accès.

Nous avons étudié ce problème dans le cas du traitement vectoriel dans un environnement multiprocesseur pipeline et nous avons proposé un modèle de fonctionnement désordonné. Chaque processeur, au lieu d'accéder aux composantes d'un vecteur l'une après l'autre suivant leur position dans le vecteur, tente l'accès à des composantes qui référencent les bancs mémoires libres. En d'autres termes les accès vectoriels sont traités en fonction de la disponibilité des bancs mémoires.

Des architectures ont proposé un modèle de fonctionnement équivalent. Les accès mémoires sont réalisés dans le désordre (c'est-à-dire l'ordre d'accès aux éléments d'un vecteur n'est pas respecté, ils sont effectués en fonction de la disponibilité des bancs mémoires qu'ils référencent). A la sortie de la mémoire les éléments sont réordonnés avant d'entamer le calcul sur ce vecteur, d'où la rupture du chaînage due à l'opération de réordonnancement. Par exemple sur le Cray-2 les instructions vectorielles ne sont pas chaînées.

A la différence du Cray-2, nous n'attendons pas la fin des opérations d'accès pour entamer le calcul. Les composantes sont traitées en fonction de leur disponibilité à la sortie de la mémoire. Ainsi la rupture du chaînage des pipelines est évitée. Le gain obtenu par le déclenchement des accès est conservé lors du calcul.

Ce document est structuré en cinq chapitres.

Le premier chapitre présente l'état de l'art des systèmes multiprocesseurs. Nous commençons par présenter les différentes classes des systèmes multiprocesseurs ainsi que les différents traitements supportés par chacune des classes. Nous présentons ensuite les machines vectorielles pipelines et leurs caractéristiques essentielles. La dernière partie du chapitre est consacrée aux techniques de résolution des conflits d'accès mémoires dans les systèmes multiprocesseurs à mémoire partagée. Nous les avons regroupés en deux classes : 1) les techniques locales qui étudient le problème de répartition des données d'une structure sur les bancs mémoires, 2) les techniques globales qui exposent le problème de l'organisation mémoire; c'est à dire l'organisation des accès à cette mémoire dans le temps pour éviter les conflits.

Le deuxième chapitre présente notre modèle de traitement vectoriel désordonné (TVD). Deux aspects sont exposés : une méthode d'accès mémoires et un modèle d'exécution vectoriel au niveau des unités fonctionnelles de chaque processeur. La méthode d'accès est introduite afin d'améliorer la capacité de transfert de données, le modèle d'exécution a pour rôle de garder le chaînage des unités fonctionnelles de calcul avec les unités d'accès mémoires.

Dans le troisième chapitre nous avons implémenté le modèle sur une architecture usuelle des supercalculateurs vectorielles pipelines à mémoire commune. Ainsi nous avons étudié les conséquences du modèle sur les architectures de ces calculateurs.

Pour évaluer les performances du modèle TVD nous avons développé une étude théorique basée sur le principe des files d'attentes, puis nous avons simulé le fonctionnement de la machine TVD et le fonctionnement d'une machine classique équivalente. Ceci fait l'objet du chapitre 4 et du chapitre 5 respectivement. L'étude théorique renferme deux modèles; un modèle analytique décrivant le fonctionnement de la machine TVD et un autre modèle décrivant le fonctionnement d'une machine sans le modèle TVD. Ces deux modèles sont ensuite validés par des simulations. Différentes expériences ont été faites pour comparer les deux modèles et évaluer leurs performances sous différents paramètres architecturaux.

Chapitre 1

Les systèmes multiprocesseurs et les conflits d'accès à la mémoire

1.1 Les systèmes multiprocesseurs

1.1.1 Introduction

L'exploitation du parallélisme est vraisemblablement l'approche idéale pour augmenter les performances des calculateurs. En terme de software, cela veut dire structurer un programme en un ensemble de sous-tâches indépendantes, pouvant être exécutées en parallèle ou identifier des traitements sur des données qui peuvent être effectués en parallèle. Evidemment, structurer un programme en sous-tâches totalement indépendantes n'est pas un problème facile, il varie d'un programme à un autre ou plus encore, il existe des programmes, dits intrinsèquement séquentiels, où ceci est impossible. En terme de hardware, cela veut dire fournir de multiples unités de traitement qui peuvent être actives simultanément. La construction d'un système à plusieurs processeurs doit prendre en considération la coopération inter-processeur pour une exécution rapide et efficace.

Un système multiprocesseur est caractérisé par :

- le nombre de processeurs et leur architecture,
- l'organisation du système mémoire,
- l'architecture du réseau d'interconnexions entre les processeurs et le système mémoire.

Suivant l'organisation mémoire on distingue deux grandes familles d'architectures multiprocesseurs :

- les multiprocesseurs à mémoire commune,
- les multiprocesseurs à mémoire locale.

Il existe des architectures qui combinent les deux organisations mémoires, elles utilisent à la fois une mémoire commune et une mémoire locale pour chaque processeur (figure (1.1)). Nous appelons ces systèmes *"multiprocesseurs mixtes"*. L'intégration d'une mémoire locale dans chaque processeur permet de réduire la fréquence des accès à la mémoire commune et ainsi diminuer le nombre de requêtes à cette ressource partagée. Nous les classerons dans la première famille par identification du mode de fonctionnement.

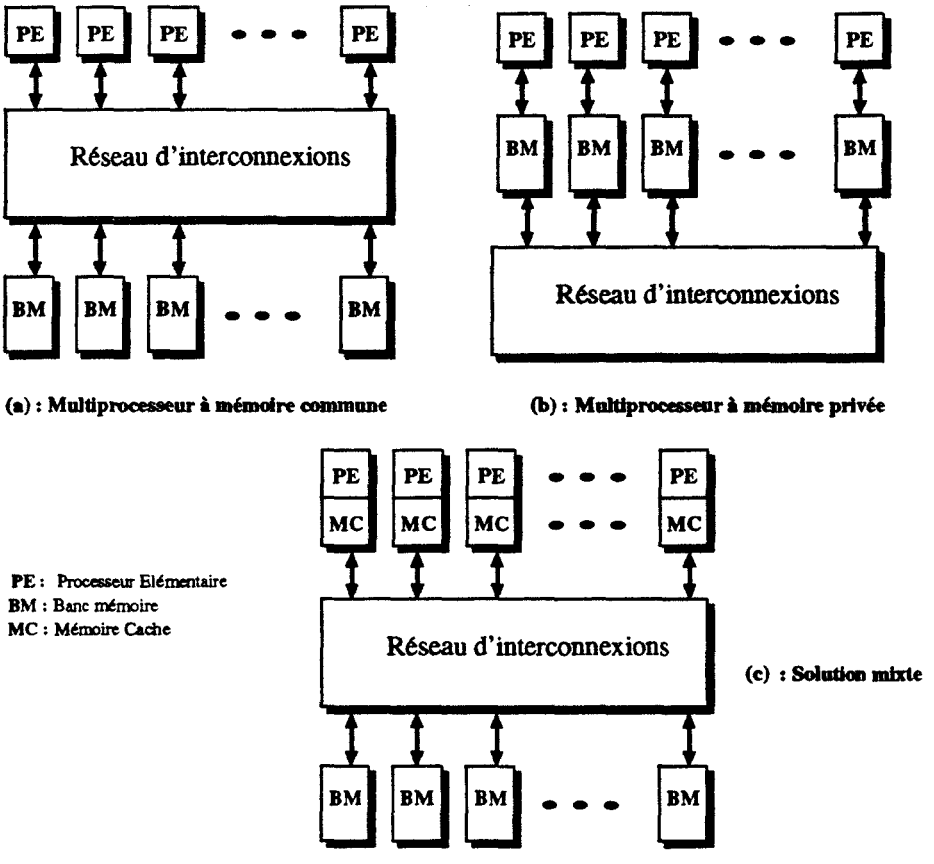


Figure 1.1 : Les architectures types multiprocesseurs.

Les deux familles co-existent, la complexité du réseau d'interconnexions entre les processeurs et les mémoires permet de les identifier. Celui-ci constitue un goulot d'étranglement, et limite le nombre de processeurs à quelques dizaines dans le cas des systèmes multiprocesseurs à mémoire commune. Par conséquent la solution d'éclater la mémoire sur les processeurs est devenue évidente pour accroître le nombre de processeurs. Chaque processeur réalise des accès à sa mémoire locale sans conflit, il communique avec les autres processeurs via le réseau.

De nos jours ces deux grandes familles sont les bases des supercalculateurs les plus puissants : d'une part nous avons les multiprocesseurs à mémoire commune constitués d'un petit nombre de processeurs très puissants tels que le Cray C-90 à 16 processeurs développant chacun une puissance de calcul de 1 Gflops [Brooks et al.91], [Simmons et al.91a] ou le Nec SX-3 à 6 processeurs d'une puissance de crête de 5,517 Gflops chacun [Eoyang et al.91], [Simmons et al.91b]; D'autre part nous avons les multiprocesseurs à mémoires locales constitués de quelques milliers de processeurs de faible puissance telles que la CM-5 de TMC à 16386 processeurs SPARC d'une puissance de 80 Mflops chacun [Gottlieb92] [Leiserson et al.91], ou Paragon de Intel à 4096 processeurs i860 XP d'une puissance de 75 Mflops chacun [Intel91].

Nous nous limiterons dans notre étude aux systèmes multiprocesseurs à mémoire commune.

1.1.2 Les systèmes multiprocesseurs à mémoire commune

Pour une architecture à mémoire commune le calculateur est un ensemble de processeurs qui partagent l'accès à une mémoire globale. Aucun processeur n'est prioritaire à une zone quelconque de la mémoire. Ainsi le réseau d'interconnexions doit assurer la liaison de chaque processeur à chaque module mémoire.

Ces systèmes restent très attractifs car ils sont simples à programmer et ils sont d'une grande simplicité d'implémentation logiciel. D'un autre côté, ils restent toujours complexes et très difficiles à construire à cause de la fréquence d'horloge très élevée et de l'existence d'une topologie du réseau d'interconnexions capable d'intégrer plusieurs processeurs et plusieurs modules mémoires.

1.1.2.1 La mémoire commune

La mémoire des multiprocesseurs joue un double rôle, d'une part elle contient les instructions et les données des programmes à exécuter, d'autre part elle permet d'effectuer des communications entre les processeurs via des accès aux variables partagées.

La mémoire est fortement sollicitée par les processeurs élémentaires (PE). Afin d'avoir un débit suffisant, au moins égal au débit des processeurs, la mémoire est divisée en un ensemble de bancs mémoires indépendants. Ces bancs mémoires sont entrelacés de façon que deux adresses successives se trouvent dans des bancs consécutifs. Cette organisation n'est pas à l'abri des conflits d'accès simultanés de deux processeurs à un même banc mémoire. Malgré l'évolution des processeurs vers des structures pipelines à un temps de cycle de plus en plus court, ils resteront toujours limités par le temps de latence de la mémoire. Ces deux problèmes, à savoir les conflits d'accès et le temps de latence de la mémoire, réduisent considérablement les performances des processeurs.

La solution qui consiste à proposer un nombre assez important de bancs mémoires permet de réduire la fréquence d'apparition des conflits, cependant le réseau d'interconnexions qui assure l'échange de données sans conflit entre les processeurs et les mémoires devient complexe et coûteux.

1.1.2.2 Architecture du réseau d'interconnexions

L'évolution des calculateurs à mémoire commune est liée à celle du réseau de connexions des processeurs aux mémoires. Ce réseau doit :

1. être simple et peu coûteux en réalisation hardware,
2. assurer un débit important, c'est à dire assurer le transfert parallèle et rapides des données,
3. assurer un temps de commutation très court. Si le temps de réponse du réseau n'est pas court, il est vite saturé et, ainsi, l'attente est inévitable.

Pratiquement il est très difficile sinon impossible de satisfaire les trois critères. La décision est souvent une solution intermédiaire pour ces trois critères. Le crossbar, par exemple, satisfait très bien le critère 2, comme le bus commun satisfait le premier critère. Cependant le crossbar est très coûteux et le bus commun a un débit très faible. Le critère 3 ne dépend pas uniquement

du type de réseau, il dépend aussi de la taille du système (N processeurs et M bancs mémoires), de la technologie et de son évolution. C'est l'un des principaux facteurs qui limitent ce type de système à quelques dizaines de processeurs.

On distingue quatre types de réseaux d'interconnexions pour les systèmes multiprocesseurs à mémoire commune; les deux solutions extrêmes que sont le bus commun et le crossbar et deux solutions intermédiaires; le réseau multi-bus et le réseau multi-étages.

- **Le bus commun**

  Les processeurs et les mémoires sont reliés entre eux par un seul lien physique. C'est la configuration la plus simple contre une performance très faible. La communication se fait en déposant sur le bus l'information à envoyer au destinataire. Une seule information est présente sur le bus à un instant donné. Cette solution n'est pas adéquate à la connexion d'un grand nombre de processeurs.

- **Le réseau multi-bus**

  Une solution intermédiaire qui améliore les performances du bus commun et qui permet d'interconnecter jusqu'à une centaine de processeurs est le réseau multi-bus. L'analyse de la bande passante et les performances des systèmes multiprocesseurs à base de ce réseau sont présentées dans [Mudge et al.87], [Das et al.85] et [Marsan et al.82].

- **Le crossbar**

  Le réseau crossbar ou *réseau à barres croisés* relie chaque processeur directement à un banc mémoire par un chemin unique. La communication de N processeurs avec N bancs mémoires se fait sans interférence ni blocage avec une fonction de routage simple. Malheureusement ces deux propriétés se payent en complexité qui est de l'ordre de N^2 en nombre de liens physiques nécessaires à sa réalisation. Lorsque N est petit la réalisation d'un tel système est faisable, mais pour construire un calculateur parallèle à grand nombre de processeurs il est moins coûteux d'utiliser une autre solution que celle du crossbar.

- **Le réseau multi-étages**

  Les réseaux multi-étages sont des solutions intermédiaires plus proches du crossbar. Ils sont introduits pour permettre la communication de N processeurs à N bancs mémoires tout en utilisant moins de liens physiques. Ils sont composés d'un ensemble de commutateurs disposés en niveaux. Chaque niveau constitue un étage du réseau. Ce type de réseau est utilisé dans la majorité des architectures multiprocesseurs. Ces réseaux ne sont pas à l'abri des interférences reconfigurables. Certains réseaux particuliers sont définis à base d'un même commutateur, éventuellement un crossbar. On peut citer le réseau de bénès, le réseau Omega, le réseau theta, etc.

1.2 Les machines vectorielles pipelines

Contrairement aux machines à grand nombre de processeurs, la classe de machines vectorielles pipelines doit ses performances à la puissance des processeurs. Ces machines possèdent quelques processeurs très puissants. Elles regroupent les deux formes de parallélisme à savoir le parallélisme à gros grain (MIMD) et le parallélisme à grain fin (SIMD). Le parallélisme à grain fin se situe au niveau des pipelines et la multiplicité des unités fonctionnelles. Le parallélisme à gros grain se situe au niveau des processeurs. Chaque processeur exécute une fraction importante de l'application. Ces calculateurs sont caractérisés par :

- Le fonctionnement pipeline.
- La multiplicité des unités fonctionnelles.
- Les registres vectoriels.
- Les processeurs multiports d'entrées/sorties.
- L'organisation mémoire.
- Les instructions d'accès à la mémoire.
- Les accès mémoires

1.2.1 Fonctionnement pipeline

L'approche pipeline a connu beaucoup de réussite. Le principe des architectures pipelines est décrit dans [Ramamoorthy et al.77] et [Hwang et al.84]. Rappelons qu'un pipeline est une unité segmentée en sous-unités, appelées étages du pipeline. Le type d'instruction qu'il traite est aussi décomposée en séquences de phases. L'exécution de l'instruction consiste à traiter chaque phase dans un étage du pipeline. La décomposition d'une unité en étages est définie par les différentes phases des instructions qu'elle exécute.

1.2.2 Multiplicité des unités fonctionnelles

La puissance d'un processeur vectoriel provient du fonctionnement pipeline et de la multiplicité des unités fonctionnelles. Ces unités peuvent travailler: 1) en parallèle en exécutant des instructions indépendantes (pas de dépendances des données) ou 2) en partageant l'exécution d'une même instruction vectorielle, ou 3) en mode pipeline, lorsque le résultat de l'une est l'opérande de l'autre, et ceci grâce au mécanisme du chaînage des unités fonctionnelles. Le mécanisme de chaînage est présenté dans le chapitre suivant. Ces caractéristiques sont communes à la majorité des machines vectorielles.

Citons, à titre d'exemple, le Cray C-90 qui possède 2 unités vectorielles de calcul en virgule flottante et chaque unité est composée de 4 unités fonctionnelles pipelines. L'unité de calcul flottant supporte 4 opérandes et produit 2 résultats en parallèle. Au total un processeur du C-90 peut produire 4 résultats par cycle de $4.0ns$; d'où une puissance de 1 Gflops. Un processeur de Nec SX-3 possède 4 unités pipelines de 4 unités fonctionnelles chacune. Toutes les unités fonctionnelles peuvent opérer en parallèle.

1.2.3 Les registres vectoriels

On distingue deux familles de processeurs vectoriels pipelines selon qu'ils sont équipés de registres vectoriels. Les processeurs qui en possèdent sont dits processeurs registre-à-registre et les autres sont dits processeurs mémoire-à-mémoire. Pour ces dernières les opérandes vectoriels sont toujours en mémoire. Le calculateur le plus puissant pour cette gamme de machine est ETA^{10} qui développe 12,8Gflops [Schonauer87] [Fatoohi89].

Dans les calculateurs registre-à-registre, les opérandes des unités fonctionnelles vectorielles sont les registres vectoriels. L'exécution d'une opération sur des vecteurs en mémoire nécessite, avant tout, le chargement des vecteurs dans les registres. Par ailleurs certains calculateurs

peuvent charger un vecteur depuis la mémoire et le second depuis un registre vectoriel (c'est le cas d'IBM 3090 VF).

Les processeurs registre-à-registre présentent deux avantages par rapport aux calculateurs mémoire-à-mémoire: 1) réduction des accès mémoires inutiles, par exemple, le chargement d'un vecteur qui sera l'opérande de deux instructions successives, le rangement d'un temporaire, etc. 2) Les accès aux registres vectoriels sont plus rapides (égal au cycle du processeur).

La taille des registres vectoriels peut être fixe (les machines Cray, Hitachi S810/20 [Lubeck et al.85a], [Lubeck et al.85b]...) ou variable c'est-à-dire reconfigurable dynamiquement. Le Fujitsu Facom VP-2600 [Simmons et al.91b] possède $128K$octets de capacité de registres vectoriels qui peuvent être configurés en mots de 32 ou 64 bits. En mots de 64 bits, par exemple, la taille des registres peut varier de 256 registres de 64 éléments, à 8 registres de 2048 éléments.

Cette taille (`VL`) est souvent insuffisante pour contenir tout un vecteur. Dans ce cas, les vecteurs sont traités par morceaux de (`VL`) éléments. Ce découpage en morceaux est appelé `strip-mining` [Weiss91]. Il est souvent réalisé par software, cependant il peut être réalisé par le matériel, comme dans Hitachi S820/80 [Eoyang et al.88].

1.2.4 Les processeurs multiports

Pour satisfaire toutes les unités fonctionnelles en opérandes et pouvoir ranger les résultats en mémoire, le processeur est doté de plusieurs ports d'entrées/sorties. Sur la machine Nec SX-3 chaque groupe de 2 processeurs vectoriels partage 3 ports d'accès à la mémoire (2 pour le chargement et un pour le rangement). Un processeur de VP-2600 a 2 ports non spécialisés et un processeur d'un Cray X-MP [August et al.89] ou Y-MP [inc88] possède 4 ports (3 ports mémoires: 2 chargements et un rangement, et un port d'entrée/sortie).

1.2.5 L'organisation mémoire

L'organisation mémoire dominante pour ce type de machines est la mémoire entrelacée à deux niveaux. La mémoire est partitionnée en (2^m) modules qui sont à leur tour partitionnés en (2^b) bancs. Les modules et les bancs sont adressables en lower-order : Les m bits d'adresses de poids faible indiquent l'adresse du module et les $m+b$ bits de poids faible indiquent l'adresse du banc (Cray, Fujitsu, Nec, Convex, ...). Bien que cette organisation soit la plus raisonnable, elle cause une baisse des performances car elle entraîne des conflits d'accès qui obligent les processeurs à attendre.

Une autre organisation, très peu utilisée, est la mémoire hiérarchisée. On la trouve dans Alliant FX/8 [Abusufah et al.86] [Wasserman et al.88] où les 8 processeurs partagent une mémoire cache pour résoudre le problème de cohérence de tâches [Litaize90], le deuxième niveau est une mémoire commune entrelacée de 64 bancs.

Une autre solution consiste à intégrer une mémoire cache dans chaque processeur. Elle constitue la troisième famille des machines multiprocesseurs. Cette solution est implémentée par la majorité des machines IBM. ETA^{10} fait partie de cette famille de multiprocesseurs. Il possède 8 processeurs dotés d'une mémoire cache de 128 bancs, ceux-ci partagent une mémoire de $256M$ mots de 64 bits. Sa mémoire globale n'est pas structurée en bancs. Une requête à cette mémoire est uniquement un ou une séquence de blocs de 64 mots. Les accès simultanés sont décalés dans le temps (sérialisés) car le taux de transfert est d'un mot par cycle.

1.2.6 Les instructions d'accès à la mémoire

Parfois le calcul ne doit être effectué que sur un ensemble d'éléments d'une structure vectorielle. Le traitement le plus usuel consiste donc **1)** à extraire cet ensemble, **2)** le traiter, **3)** puis ranger le résultat aux bons emplacements dans le vecteur initial. Il y a deux solutions pour leurs implémentations: par vecteur d'index ou par vecteur de bits. L'opération **(1)** est appelée regroupement, en anglais **gather** (resp. **compress**) pour l'implémentation par vecteur d'index (resp. par vecteur de bits), et l'opération **(3)** est appelée éclatement, en anglais **scatter** (resp. **extend**) pour l'implémentation par vecteur d'index (resp. par vecteur de bits).

1.2.6.1 Opération Gather/Scatter

Le vecteur d'index contient les adresses des éléments à traiter de la structure vectorielle. En lecture (gather), il contient les index des composantes à charger dans un registre vectoriel. En écriture (scatter), il contient les index du vecteur destination. La figure (1.2) illustre leurs fonctionnements.

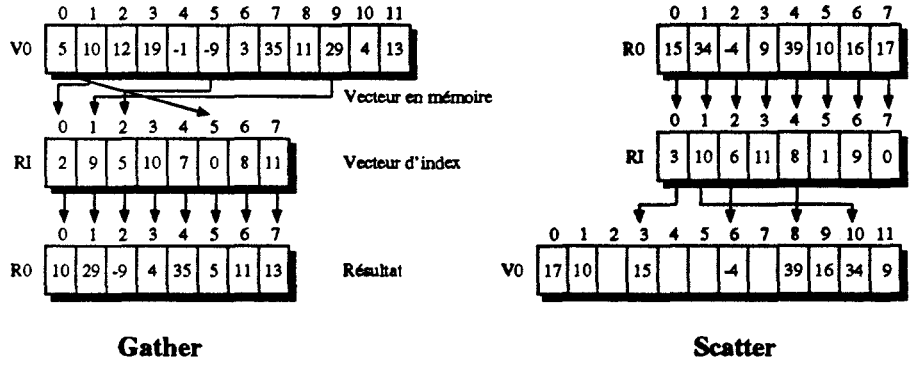


Figure 1.2: Fonctionnement des opérations **Gather/Scatter**.

1.2.6.2 Opération Compress/Extend

Le vecteur de bits est appelé registre masque. Il contient une chaîne de bits. Comme son nom l'indique, l'opération consiste à masquer les éléments qui ne sont pas concernés par le traitement. Ces deux opérations sont illustrées dans la figure (1.3).

Les opérations Gather/Scatter et Compress/Extend sont souvent disponibles sur les machines vectorielles. L'utilisation de l'un ou l'autre type d'opération dépend du taux de composantes à traiter. On a montré que pour un taux supérieur à 10%, il est préférable d'utiliser les opérations Compress/Extend [Ibbett et al.89]. Néanmoins, les opérations Gather/Scatter sont plus générales. Avec un vecteur de bits on ne peut pas réaliser de "croisement" comme celui de la figure (1.2). Les opérations Compress/Extend respectent l'ordre croissant des composantes.

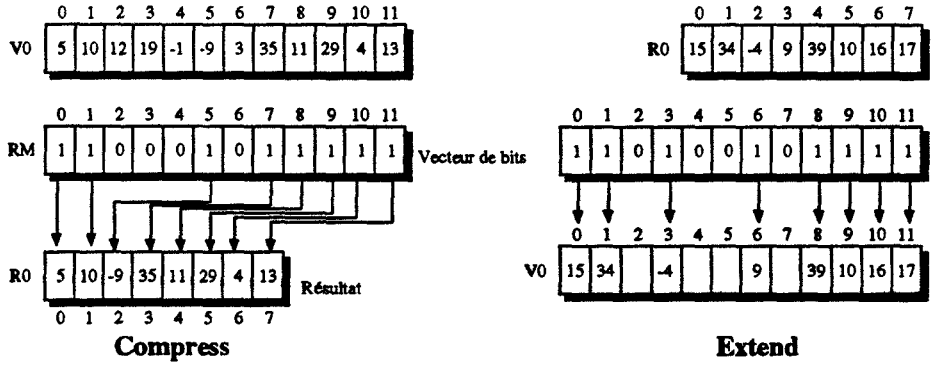


Figure 1.3 : Fonctionnement des opérations **Compress/Extend**.

1.2.7 Les accès mémoires

La capacité de transfert de données entre les processeurs et les mémoires est le facteur le plus important qui limite les performances de la machine. Ce facteur dépend :

- du débit effectif du système mémoire (nombre de bancs),
- du débit effectif du réseau d'interconnexions (type du réseau),
- et surtout du type des accès mémoires.

Nous ne nous attarderons pas sur les deux premiers critères: ils ont déjà été traités dans les sections précédentes et sont plutôt des contraintes matérielles.

Un banc mémoire ne peut satisfaire qu'un accès par cycle mémoire. La demande d'accès à un même banc par plusieurs processeurs simultanément provoque des conflits d'accès, donc des chutes de performances. D'une manière générale, plus les accès sont fréquents, plus la probabilité d'avoir un conflit est importante. Par conséquent une première solution consiste à réduire le nombre d'accès à la mémoire. Ceci peut se faire par :

- les accès à des super-mots, ou
- l'utilisation de mémoires caches.

L'accès à des super-mots consiste à charger ou décharger plusieurs mots mémoires contigus en un seul cycle mémoire. Les machines vectorielles mémoire-à-mémoire ont généralement ce genre d'accès. Le multiprocesseur ETA[10] a adopté cette technique qu'il a hérité du Cyber 205. Cependant les accès mémoires ne sont pas toujours des accès à des mots contigus. On peut générer des accès par pas lors de l'accès à une colonne d'une matrice qui est rangée ligne par ligne ou bien lors de l'exécution d'une opération gather/scatter. L'accès par super-mots, dans ce cas, induit des chutes de performances considérables.

L'utilisation d'une mémoire cache peut effectivement réduire les accès à la mémoire commune, mais à chaque défaut de page le traitement sera suspendu pendant plusieurs cycles mémoires. Un autre problème qui rend le cache inefficace se pose lorsque la structure de donnée vectorielle

(vecteur, matrice, ...) ne peut pas être contenue dans le cache, de même l'accès par pas ou par opérations gather/scatter peut impliquer plusieurs défauts de pages, d'où dégradation des performances.

La deuxième solution consiste à éviter les conflits sans réduire le nombre d'accès. Notre travail se place dans cette catégorie de solutions. La suite de ce chapitre est consacrée aux différentes solutions qui permettent de réduire les conflits d'accès mémoires.

1.3 Les conflits d'accès, les solutions apportées

Les systèmes multiprocesseurs actuels sont dotés de processeurs très puissants, de l'ordre du Gflops, et d'un système mémoire de très grande capacité et de très haut débit, qui reste moins rapide que les processeurs. Comme nous l'avions signalé précédemment, l'architecture de ces systèmes est basée essentiellement sur le choix et le fonctionnement du réseau d'interconnexions. Le réseau idéal pour ce genre de systèmes est le crossbar, mais vu le coût et la complexité qu'il engendre, on a plus tendance à utiliser des réseaux moins coûteux et simples à réaliser. Ces solutions intermédiaires génèrent des conflits d'accès, ces conflits sont de différents types à savoir (voir (2.3) pour définition précise) :

- les conflits de mémoire occupés,
- les conflits de réseau d'interconnexions et
- les conflits de simultanéité.

Plusieurs techniques, pour minimiser le nombre de conflits, ont été proposées, nous les avons réparties en deux catégories : **les techniques locales** qui concernent les accès sans conflits aux structures qui nous intéressent (ligne, colonne, diagonale, ...). Elles sont liées à ces structures, donc à une zone précise de la mémoire. **Les techniques globales** sont indépendantes des structures d'une application, elles opèrent sur la totalité de l'espace d'adressage de la mémoire.

1.3.1 Techniques locales

Le mot "local" est employé pour désigner les techniques qui s'intéressent uniquement au rangement d'une structure (des tableaux à n-dimensions) en fonction de son utilisation dans l'application. Leur rôle est de permettre des accès sans conflit aux sous-structures de la structure initiale. Dans un tableau à deux dimensions, on s'intéressera aux accès sans conflit à des lignes, colonnes, diagonales, sous-blocs carrés, etc. Ces techniques sont mieux adaptées à des machines tableaux de N PEs et N bancs mémoires (figure (1.4)).

Les premières recherches sur ces méthodes s'intéressent plutôt à trouver un schéma de rangement qui permettrait d'accéder sans conflits à tout ensemble de sous-structures. La réponse à cette question est apportée par Budnik & Kuck [Budnik et al.71]. Leurs travaux consistaient à ranger un tableau à 2 dimensions dans une mémoire à M bancs de façon à ce que les sous-structures linéaires puissent être chargées en parallèle. Ils ont montré qu'il n'y a aucun schéma de rangement qui permet l'accès sans conflits en utilisant tous les pas de déplacement (sous-structures linéaires), cependant ce schéma existe si le nombre de bancs M est premier. Ce résultat a été généralisé dans [Shapiro78], [Frailong et al.87], [Wijshoff et al.85], [Wijshoff et al.87].

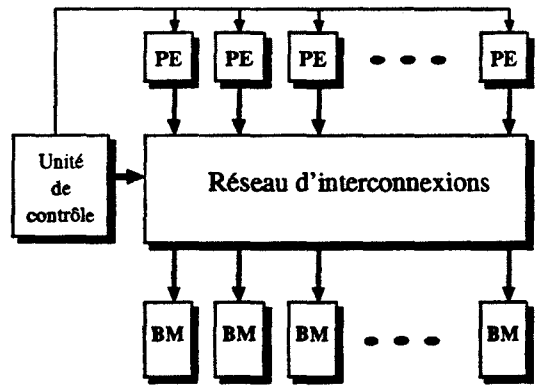


Figure 1.4 : Architecture d'une machine SIMD.

Budnick & Kuck, suivi par d'autres auteurs, ont proposé d'autres schémas de rangement qui sont des variantes de celui de la figure (1.5). Dans la figure (1.5a) une colonne n'est pas accessible en parallèle, et la figure (1.5b) montre comment ranger les éléments du tableau pour accéder à toute colonne en parallèle. Ces schémas sont appelés **schémas de rangement oblique** dans la littérature.

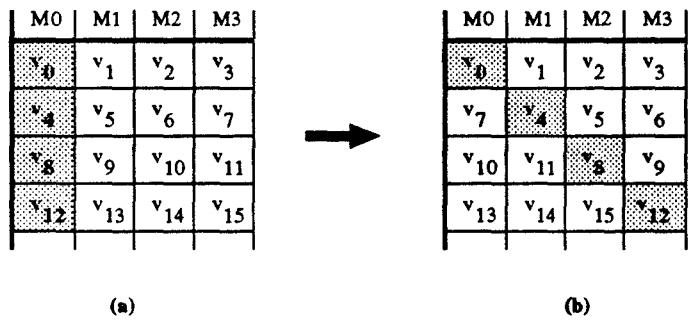


Figure 1.5 : (a) : le schéma de rangement séquentiel. (b) : Un schéma de rangement oblique.

Les paragraphes suivants présentent les schémas de rangement oblique selon la classification de Schapiro. Il a classé ces schémas en deux classes qui sont les rangements obliques linéaires et non linéaires.

1.3.1.1 Schéma de rangement oblique linéaire

Un schéma de rangement oblique est une fonction qui associe de manière unique un élément d'une structure à une adresse mémoire. Cette fonction est $S(A(i,j)) = k$ qui signifie que l'élément $A(i,j)$ est rangé dans le banc mémoire k. S décrit tous les schémas de rangement oblique.

De manière formelle, un schéma de rangement oblique est dit linéaire si l'adresse du banc mémoire dans lequel est rangé l'élément d'une structure à n dimensions $x(i_1, i_2, \cdots, i_n)$ est donnée par la fonction : $f(i_1, i_2, \cdots, i_n) = (\lambda_1 i_1 + \lambda_2 i_2 + \cdots + \lambda_n i_n) \bmod M$, où $\lambda_{i,(i=1,\cdots,n)}$

sont des constantes et M est le nombre de bancs mémoires [Shapiro78], [Wijshoff et al.85].

Le problème qui se pose est de choisir $\lambda_{i,(i=1,\cdots,n)}$ afin de permettre l'accès sans conflit aux sous-structures qui nous intéressent. Ici, on se limitera aux structures à 1, 2 dimensions car ce sont les plus utilisées en pratique et que toute structure à n-dimensions peut se réduire à un schéma à 2 voire à 1 dimensions.

Dans ce qui suit nous allons présenter sur quelques exemples, certains schémas de rangement oblique linéaire. Nous présenterons deux types de solutions utilisant ou non d'un réseau de réalignement.

1.3.1.1.1 Schémas sans réseau de réalignement

La solution de Budnick & Kuck [Budnik et al.71] consiste à ranger une matrice $(M \times M)$ ligne par ligne dans une mémoire à M bancs de telle façon que la ligne (i) soit décalée de S, appelé schéma de rangement, par rapport à la ligne $(i-1)$. En d'autres termes, le schéma de rangement S est donné par la distance, en nombre de colonnes, entre deux lignes consécutives de la matrice. Par exemple, les deux rangements (a et b) donnés en figure (1.5) sont obtenus respectivement pour $S = 0$ et $S = 1$. Soit d la distance (modulo M), mesurée en colonnes, entre deux éléments successifs d'une structures linéaire (ligne, colonne, diagonale, ...). L'expression du déplacement entre deux éléments successifs est : $\delta = (d + (j-i)S) \bmod M$. Ce schéma de rangement peut être formalisé de la façon suivante : $m_s(i,j) = (Si + j) \bmod M$. C'est à dire l'adresse du banc dans lequel est rangé l'élément (i,j) est donnée par la fonction $m_s(i,j)$.

La solution de HarperIII [Harper et al.87] est semblable à la précédente, mais de manière beaucoup plus formelle. Il ne traite que des structures mono-dimensionnelles. Le passage d'une structure multi-dimensionnelle à une structure mono-dimensionnelle peut être fait à la compilation. Soient a une adresse de base et S le pas de déplacement, l'adresse du banc qui contient la composante (i) est : $m_s(i) = (Si + a + \lfloor \frac{Si+a}{M} \rfloor) \bmod M$. Notons que le symbole $\lfloor\ \rfloor$ désigne la partie entière d'un nombre.

Inconvénients

Ces deux solutions, ne permettent pas l'accès sans conflit à toutes les structures les plus courantes (colonnes, diagonales principale et secondaire, etc.) pour un schéma de rangement donné S. Dans l'exemple de la figure (1.6) on peut accéder sans conflits aux lignes, mais il n'en est pas de même pour les colonnes (1.6b) et la diagonale (1.6a et 1.6b).

Un autre aspect négatif est lié au réseau de réalignement de données. Budnick & al. propose un crossbar qui est très coûteux dans les architectures SIMD [Flynn72] utilisant plusieurs PEs. Dans la deuxième solution on a proposé deux bus reliant les mémoires aux unités centrales. Le problème de réalignement ne se pose pas, mais le débit est très faible.

Améliorations

Pour améliorer ces solutions, Budnick & al., pour leur part, ont proposé l'utilisation d'un nombre premier de bancs. Cette solution permet l'accès aux lignes, colonnes et diagonales, cependant elle pose d'autres problèmes qui sont :

1. la complexité du calcul d'adresses,

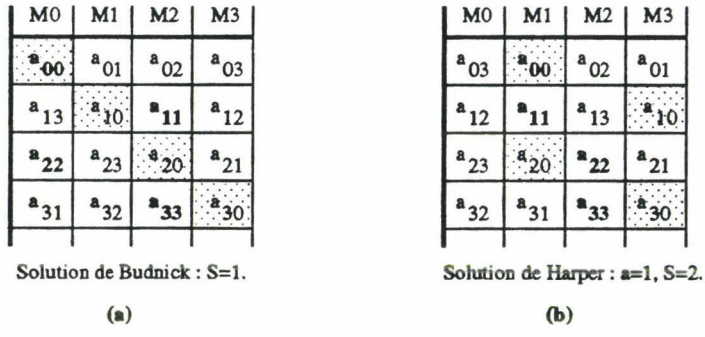


Figure 1.6 : (a) Budnick & al. : schéma de rangement pour $S = 1$ (b) HarperIII & al. : schéma de rangement pour $S = 2$ et $a = 1$.

2. la non-symétrie du réseau de réalignement, et

3. une perte d'espace mémoire (c'est le cas de la machine BSP utilisant 16 processeurs et 17 bancs mémoires).

Chaque valeur du schéma S permet l'accès à quelques structures bien précises. HarperIII [Harperiii et al.91], proposent un **schéma de rangement dynamique**. Chaque structure est rangée en mémoire suivant le type d'accès à effectuer. Cette solution exige que le pas d'accès soit connu à l'avance (à la compilation). Ils résolvent notamment le problème du rangement d'une structure à accès multiple. Les pas d'accès ne sont pas toujours connus à la compilation, l'interet de cette méthode est grandement réduit.

Les méthodes suivantes proposent un schéma de rangement et le réseau de réalignement adéquat.

1.3.1.1.2 Schémas avec réseau de réalignement

Lawrie [Lawrie75] a proposé un schéma de rangement à deux dimensions. Etant donnée une matrice $A(N \times N)$, l'adresse du banc est donnée par une fonction $m(A_{i,j}) = (\delta_1 i + \delta_2 j) \bmod M$. δ_i est la distance, en nombre de colonnes, $(\bmod M)$ suivant la i^{me} dimension. Ainsi, les lignes sont rangées suivant δ_1. On dit qu'elles sont **d'ordre** δ_1. Les colonnes sont d'ordre δ_2, les diagonales sont d'ordre $\delta_1 + \delta_2$ etc. (Un vecteur est d'ordre δ si chaque élément (i) est rangé dans le banc d'adresse $\delta i + a$, a étant l'adresse de base du vecteur). La figure (1.7) montre un exemple de rangement d'une matrice (4×4) sur 8 bancs mémoires avec un schéma de rangement $(\delta_1 = 3, \delta_2 = 2)$. L'adresse de base a est égale à 1.

Lawrie a montré que le schéma de rangement $(\delta_1, \delta_2) = (\sqrt{N} + 1, 2)$ permet l'accès aux lignes, colonnes, diagonales et sous-blocs carrés $(\sqrt{N} \times \sqrt{N})$ d'une matrice $(N \times N)$ lorsque $M = 2N$. Il a utilisé un réseau de réalignement de données appelé *réseau omega*. C'est un réseau multi-étage. Le calcul d'adresse est relativement simple car on ne manipule que des nombres en puissance de 2.

M0	M1	M2	M3	M4	M5	M6	M7
	a_{00}		a_{01}		a_{02}		a_{03}
a_{12}		a_{13}		a_{10}		a_{11}	
	a_{21}		a_{22}		a_{23}		a_{20}
a_{33}		a_{30}		a_{31}		a_{32}	

Figure 1.7 : Rangement oblique à 2-dimensions : $(\delta_1 = 3, \delta_2 = 2)$

Inconvénients

Cette méthode souffre de deux inconvénients majeurs :

- La sous-utilisation de la mémoire. Cette perte est de 50% lorsqu'on permet l'accès aux sous-structures linéaires.
- Le réseau de réalignement omega présente des conflits car il ne permet pas toutes les permutations possibles. Par conséquent le transfert ne peut pas s'effectuer en une seule passe.

Amélioration

Afin de réduire l'espace mémoire perdu, [Lawrie et al.82] proposent d'associer cette technique avec la technique du nombre premier de bancs utilisée par Budnick pour la machine BSP. Le calcul d'adresse est réalisé de manière récursive, mais la division par un nombre premier est inévitable. Ils ont élaboré deux fonctions de rangement : l'une "f" pour le calcul d'adresse du banc et l'autre "g" pour calculer l'adresse de l'élément dans le banc. Ces deux fonctions sont : $f(i,j) = (jI + i + base) \bmod M$ et $g(i,j) = (jI + i + base)/P$, où (I, J) sont les dimensions du tableau, P étant le nombre de processeurs tel que P est la plus grande puissance de 2 inférieure à M. Dans la machine BSP la fonction f est calculée par chaque processeur et le calcul de la fonction g est pris en compte par le banc mémoire. Ainsi on évite de transférer l'adresse du mot à travers le réseau. Le réseau de réalignement est un double crossbar (un crossbar dans chaque direction). Bien qu'il produise un débit optimum, il reste très coûteux.

1.3.1.2 Schéma de rangement oblique non linéaire

Un ***schéma de rangement oblique est dit non linéaire*** s'il n'est pas une combinaison linéaire des indices (i, j). Toute la difficulté demeure dans le choix de la fonction de génération de l'adresse k.

Les techniques non linéaires sont généralement caractérisées par une bonne utilisation de la mémoire (pas de perte d'espace mémoire) et une fonction d'adressage simple à implanter. Par contre le choix du réseau de réalignement n'est pas un problème facile (Cydra-5 [Rau et al.89b], IBM RP3 [Dongara87]). Une bonne technique doit prendre en compte le fonctionnement du réseau afin d'éviter des pertes de performances à ce niveau.

Etant donnée une adresse d d'un élément d'une structure, l'adresse du banc est obtenue en faisant une ou plusieurs opérations sur les bits de d. Dans les techniques linéaires, les adresses de rangement des éléments successifs d'une structure sont nécessairement soit en progression arithmétique soit en progression géométrique. Cela n'est pas le cas pour les techniques non-linéaires. Un exemple de schéma non-linéaire est montré dans la figure (1.8). Suivant les littératures, ces rangements sont dit brouillés [Lee88] ou pseudo-aléatoires [Rau91].

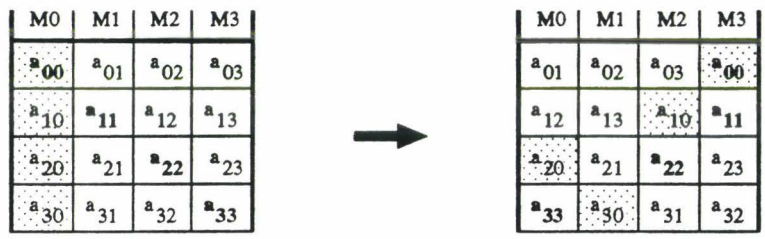


Schéma de rangement séquentiel. (a)

Schéma de rangement brouillé. (b)

Figure 1.8 : Exemple de rangement brouillé ou pseudo-aléatoire d'une matrice (4 × 4).

Nous avons identifié trois classes de fonctions de génération d'adresses des bancs :

- La fonction du type $F = f(a)$: simple et utilisant une seule transformation. Elle est décrite par une matrice binaire pseudo-aléatoire.
- La fonction du type $F = i \oplus f(j)$: définit de manière exacte le réseau de réalignement adéquat pour les machines tableaux.
- La fonction du type $F = f(i, j) \bmod M$: est une généralisation des deux classes précédentes.

Nous allons maintenant prsenter chacune d'elles.

1.3.1.2.1 Fonction du type $F = f(a)$

Soient a et b deux adresses telles que $b = f(a)$. a et b ont pour représentations binaires respectives $< a_{n-1}, \cdots, a_0 >$ et $< b_{n-1}, \cdots, b_0 >$. La recherche de b, revient à résoudre le système d'équations logiques suivant : $Ha = b$, où H est la matrice de f.

Ces méthodes sont simples et leur implémentation est directe, car la fonction F est une fonction logique qui opère sur les bits d'adresse de la donnée à ranger en mémoire. Elle est très souvent la fonction `XOR`. La seule condition imposée à f pour qu'elle définisse un schéma de rangement est qu'elle soit bijective.

Rau [Rau et al.89b] pense qu'un bon schéma de rangement doit réunir les conditions suivantes :

1. La fonction f doit couvrir la totalité de l'espace mémoire de la machine.
2. L'ensemble image de la fonction f est égal à l'espace mémoire alloué à la structure. Cette condition garantit la bonne efficacité d'utilisation de la mémoire. Il n'y a pas de perte d'espace mémoire.

3. La fonction f doit éviter la concentration d'un grand nombre de références à un même banc mémoire. Cela signifie qu'il faut utiliser le plus possible de bits de l'adresse initiale a pour la détermination de b.

4. La fonction f ne doit pas modifier les bits qui servent d'adresse de l'emplacement de la donnée à l'intérieur du banc.

Ils ont construit un schéma de rangement $H(i,j)$ et ils l'ont appelé *rangement entrelacé pseudo-aléatoire*. Ce rangement vérifie les conditions précédentes. $H(i,j)$ est défini par : $H(i,j) = 1 \Rightarrow a_{n-1-i}$ est une entrée d'une porte `XOR` qui a pour sortie f_j.. $H(i,j) = 0$ sinon.

Un exemple de schéma *pseudo-aléatoire* est donné en figure (1.9). Implanté sur la machine Cydra-5, cette méthode donne de bons résultats [Rau91]. Cependant cette machine est mono-processeur (pas d'effet d'interaction inter-processeur). En outre elle implémente la méthode de files d'attentes, aussi efficace, que nous allons décrire en section (1.3.2.3.1).

1	0	1	0	0	1	0	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	1	1	1	0
0	0	0	1	1	1	1	1	1	0	1	0	1	1	1	0	1	1	0	1	1	0	0	0	0	1	0
0	0	0	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	0	1	0	0	1
1	1	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1
0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	1

Figure 1.9 : Exemple d'une matrice (6 × 27) pseudo-aléatoire.

Contrairement aux autres schémas obliques destinés aux traitements SIMD sur machines tableaux, le schéma *pseudo-aléatoire* est implémenté sur une machine vectorielle pipeline à traitement data-flow. Son implémentation est simple avec un minimum de hardware (elle nécessite quelques portes `XOR`). Elle est efficace : le temps de calcul des adresses physiques est constant et très court.

Une nouvelle technique de construction de la matrice H est introduite dans [Kim et al.89] qui proposent d'utiliser le procédé **carré latin** (*Latin square*) pour définir un schéma de rangement. Latin square est un tableau $(n \times n)$ composé de nombres de 0 à $n-1$ tel qu'aucun nombre n'apparaisse plus d'une fois dans une ligne ou dans une colonne. Ainsi on pourra ranger une structure suivant ce carré latin dont les éléments seront les numéros des bancs, on accédera sans conflits aux lignes, colonnes, diagonales, etc.

La solution de Kim & al. permet la génération d'adresses à base de circuits simples en un temps constant. Nous n'allons pas décrire cette solution ici car elle est très longue et beaucoup de définitions sont à introduire. Elle est exposée dans [Kim et al.89].

Une théorie plus complète et précise sur les techniques de rangement à base de **Latin squares** a été récemment publiée par [Colbourn et al.92]. Les auteurs ont montré qu'avec ces techniques on peut contrôler la portion du nombre de bancs libres lors des accès aux sous-structures qui nous intéressent. Ils ont montré aussi que les meilleures méthodes d'accès aux sous-structures carrées $(n \times n)$ ou bloc $(n \times m)$ où n et m sont premiers entre eux, sont les techniques obliques linéaires.

Réflexions

Les auteurs ont solutionné le problème de perte d'espace mémoire et de circuit de calcul d'adresses, mais ils ont laissé de côté le problème du réseau de réalignement de données.

Cydra-5 implémente une méthode qui combine la méthode des files d'attentes et la méthode pseudo-aléatoire. Cette méthode est testée pour une taille de buffers égale à 1 (c'est-à-dire presque en absence de la méthode de files d'attentes), elle se révèle moins performante lorsque le temps de latence de la mémoire augmente [Raghavan et al.90].

Finalement les conditions de définition d'un bon schéma de rangement proposées par Rau sont :

- très générales, elles ne donnent aucune aide à la construction d'un bon schéma.
- Aucune des conditions ne considère le réalignement de données.
- Elles servent de définition à une bonne utilisation de la mémoire, mais elles ne garantissent pas une bonne efficacité du schéma d'accès.

Par conséquent les méthodes du type $F = f(a)$ sont insuffisantes pour prendre en compte le réseau de réalignement. En revanche, elles sont efficaces pour l'optimisation de de l'utilisation de la mémoire et du système d'adressage.

1.3.1.2.2 Fonction du type $F = i \oplus f(j)$

Ce type de fonctions est caractérisé par la composition d'une fonction `XOR` ($\oplus$) et d'une autre fonction qui est soit `modulo`, soit `pseudo-aléatoire`. En réalité ce type de fonctions est une variante de celle utilisée dans STARAN [Batcher77]. Lorsque f est une identité on retrouve exactement le schéma décrit dans la section (1.3.1.2.3.A). Nous préférons les classer à part car le réseau de réalignement est défini d'une manière bien particulière. Nous présentons ici deux méthodes : l'une est basée sur les permutations linéaires et l'autre est la méthode brouillée de Lee [Lee88].

A– Méthode de permutation linéaire

Raghavendra et Boppana ont proposé un schéma de rangement qui fournit des accès sans conflit aux sous- structures usuelles en utilisant le réseau Ω (Omega) implémenté dans plusieurs machines : IMB RP3, BBN Butterfly, NYU Ultracomputer et Cedar multiprocesseur [Raghavendra et al.90].

On considère un système à $N = 2^n$ processeurs et N bancs mémoires. Soit l'ensemble $V = \{0, 1, \cdots, N-1\}$. Une permutation linéaire sur V est une fonction qui transforme tout $i \in V$ en $j \in V$ tel que chaque bit de j est une combinaison linéaire des bits de (i mod 2). Cette permutation est décrite par une matrice binaire Q telle que $j = Qi$.

Etant donné un tableau A à deux dimensions, l'adresse du banc b où se trouve $A(i, j)$ est donnée par le schéma de rangement $b = i \oplus Qj$.

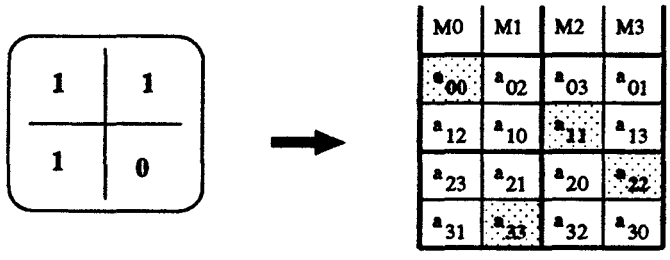


Figure 1.10 : Exemple de schéma par la méthode de permutation linéaire.

Réflexions

- Le circuit de calcul d'adresses comporte une mémoire qui contient la matrice booléenne Q et un réseau de portes logiques pour effectuer le produit Qj et l'opération $\bigoplus$. La complexité du circuit est de l'ordre de $(\log N)^2$.
- Comme le montre la figure (1.10), l'accès sans conflits aux lignes, colonnes, diagonales, est possible.
- Le réseau d'interconnexions est un simple réseau Ω légèrement modifié. Cependant ce type de réseau (Ω) présente des conflits. Par conséquent les données ne peuvent pas être débrouillées (unscrambled) en une seule passe.
- Le réseau Ω modifié ne donne pas la même priorité d'accès à tous les processeurs. Chaque processeur est plus prioritaire pour un banc mémoire particulier qu'on peut qualifier de mémoire locale.

B- Méthode brouillée

Le schéma oblique non linéaire de De-lei Lee [Lee88] range les éléments d'un tableau à deux dimensions dans N ($N = 2^n$ avec n pair) bancs mémoires moyennant la fonction : $f(i) = (\lfloor i/z \rfloor + iz) \bmod N$ avec $z = \sqrt{N}$.

L'adresse du banc pour un élément $A(i,j)$ du tableau est : $b = j \oplus f(i)$, et l'adresse locale dans le banc est j.

Lee a défini un réseau multi-étage appelé (**theta**). C'est un réseau ($N \times N$) à $n/2$ étages de $N/4$ commutateurs chacun. Un commutateur est un crossbar (4×4). Les entrées et les sorties du commutateurs sont numérotées de 00 à 11, et une variable de contrôle C à trois bits ($C_0C_1C_2$) positionne les sorties S_e des commutateurs d'un même étage en fonction des entrées de la manière suivante :

$$S_e(e_0e_1) = \begin{cases} e_0 \oplus C_1e_1 \oplus C_2 & \text{si } C_0 = 0, \\ e_1 \oplus C_1e_0 \oplus C_2 & \text{si } C_0 = 1. \end{cases}$$

Réflexions

- Le réseau de réalignement permet de brouiller et de débrouiller les données en utilisant une fonction de contrôle souple et facile à implémenter.

- Le circuit de génération d'adresses n'est composé que de portes logiques `XOR`, donc très réduit.
- Cette méthode permet l'accès sans conflits aux lignes, colonnes, blocs carrés et blocs carrés dispersés, mais pas à la diagonale.

1.3.1.2.3 Fonction du type $F = f(i,j) \bmod M$

A– Méthode STARAN

STARAN est une machine tableau à mémoire partagée conçue par Goodyear Aerospace en 1962 [Batcher77], [Hwang et al.84], [Hockney et al.88]. Cette machine a les caractéristiques suivantes :

- Elle est composée de 256 processeurs élémentaires d'un bit et de 32 modules mémoires de mots de 8 bits.
- La mémoire est composée de modules mémoires associatifs.
- Le réseau réseau de réalignement de données est très simple.
- Une méthode d'accès MDA (MultiDimensional Access) est utilisée pour accéder aux données en mémoire.

Les données sont accessibles en mémoire par `bit slice`. Ce type d'accès est très intéressant quand il s'agit de traiter des opérations vectorielles. A titre d'exemple, une opération vectorielle du type $B \leftarrow A + B$ exécutée sur des vecteurs de 256 éléments de 32 bits nécessite 96 cycles sur STARAN et plus de 256 cycles sur le Cray X-MP. Avec sa méthode d'accès MDA et son réseau de réalignement, cette machine permet l'accès par bit slice, par mot et la combinaison des deux.

Sur STARAN la méthode consiste à stocker un bit par mot dans un banc mémoire. L'accès par bit slice dans un banc produit 8 bits de 8 mots différents. Ici on exposera la méthode d'une autre manière, en considérant que chaque donnée est sur un seul banc. Ceci ne change pas la définition de la méthode.

La méthode MDA est une fonction logique appliquée sur les indices d'un élément d'un tableau à deux dimensions. L'élément $A_{i,j}$ est rangé à l'emplacement (i) du banc mémoire d'adresse $B = (i \oplus j) \bmod M$, M étant le nombre de bancs mémoires. Inversement, étant donné un emplacement i et un banc mémoire b, la donnée rangée à cet emplacement est $A_{i,i\oplus b}$.

On dit que les éléments du tableau A sont rangés de manière *"brouillée"* sur les bancs mémoires (voir la figure (1.11)).

Réflexions

- L'un des aspects très intéressant de la méthode est le réseau de réalignement de données. Il est très efficace : après avoir été brouillées en mémoire, les données sont "débrouillées" grâce à un mécanisme de contrôle efficace.
- Le calcul d'adresses est simple et son implémentation est directe (n portes logiques `XOR`).

M0	M1	M2	M3
a_{00}	a_{11}	a_{22}	a_{33}
a_{01}	a_{10}	a_{23}	a_{32}
a_{02}	a_{13}	a_{20}	a_{31}
a_{03}	a_{12}	a_{21}	a_{30}

Figure 1.11 : Schéma de rangement de STARAN.

- L'aspect négatif de la méthode est qu'elle ne permet pas l'accès à toutes les sous-structures usuelles. C'est le cas d'accès aux lignes par exemple (figure (1.11)).

Pour pallier à ce problème [Frailong et al.85] ont proposé un schéma plus général défini par la fonction : $f(A_{i,j}) = (FI \oplus GJ) \bmod M$, où F (resp. G) est une matrice $(M \times u)$ (resp. $(M \times v)$), et I (resp. J) est la représentation binaire de i (resp. j). A est un tableau $(u \times v)$.

Le schéma de rangement de STARAN de la figure (1.11) est un cas particulier de cette méthode, en posant $u = v = 4$, et F et G sont des matrices identités.

B- Méthode apériodique

Weiss [Weiss89] a introduit un schéma de rangement qui a la particularité d'être apériodique. Il est destiné, en priorité, aux machines vectorielles pipelines du type Cray.

Les accès par pas puissance de 2 sont très pénalisant parce que le nombre de bancs mémoires référencés est très réduit, et que le temps de latence de la mémoire est long. Lorsque le temps de latence est relativement important, un banc en activité pendant la période précédente reste occupé pendant la période en cours. Par conséquent le processeur qui référence ce banc doit attendre sa libération. Le Cray-1 [Russell78] (qui a 16 bancs et un temps de latence mémoire de 4 cycles) est un exemple de ce type de pénalité lorsque le pas d'accès est de 8, 16 ou multiple de 16. La situation est encore plus grave sur le Cray-2 (qui a 128 bancs et 61 cycles de temps de latence mémoire) [Schonauer87]. Avec cette méthode les bancs mémoires ne sont pas référencés de manière périodique.

Principe:

On considère un schéma de rangement dans lequel on modifie quelques bits des m ($m = \log M$) bits de poids faibles de l'adresse a en utilisant des opérations XOR. L'adresse du banc est déterminée par $b = f(a) \bmod M$. M étant le nombre de bancs.

La définition de f fait intervenir la composition de P fonctions élémentaires f_i. Chaque fonction élémentaire permet de changer un bit de l'adresse a comme suit :

Soit $< a_{N-1}a_{N-2}\cdots a_0 >$ la représentation binaire de a et $\{i_0, i_1, \cdots, i_n\}$ un sous-ensemble d'indices à valeurs dans $\{0 \cdots N-1\}$. On a

$f_{i,(i_0,i_1,\cdots,i_n)}(a) = a_{N-1}a_{N-2}, \cdots, a'_i, \cdots, a_0$, où $i \in \{0 \cdots M-1\}$, et $a'_i = a_{i_0} \oplus a_{i_1} \oplus \cdots \oplus a_{i_n}$.

$F(a) = f_{P-1,(p_0,\cdots,p_{q-1})} \hat{} \cdots \hat{} f_{1,(j_0,\cdots,j_{q-1})} \hat{} f_{0,(i_0,c\ldots,i_{q-1})}(a) \bmod M$ où $q = N/P$.

On fait intervenir la fonction ($\mathrm{mod}\, M$) car le nombre P de fonctions qui peuvent rentrer dans la composition de f est quelconque. En choisissant $P = M$, les $N - M$ bits de poids forts sont inchangés, et la condition (4) de Rau est vérifiée (on minimise la réalisation matérielle de la fonction de calcul d'adresses). Dans la figure (1.12) on donne un exemple de distribution d'adresses sur une mémoire composée de 8 bancs avec

$f(a) = f_{2,(4,3,2,0)} \hat{} f_{1,(5,4,1)} \hat{} f_{0,(3,1,0)}(a) \bmod 8.$

M0	M1	M2	M3	M4	M5	M6	M7
a_0	a_5	a_3	a_6	a_4	a_1	a_7	a_2
a_{13}	a_8	a_{14}	a_{11}	a_9	a_{12}	a_{10}	a_{15}
a_{23}	a_{18}	a_{20}	a_{17}	a_{19}	a_{22}	a_{16}	a_{21}
a_{26}	a_{31}	a_{25}	a_{28}	a_{30}	a_{27}	a_{29}	a_{24}

Figure 1.12 : Distribution d'adresses suivant une fonction apériodique de Weiss.

Réflexions

Cette méthode permet l'accès sans conflits aux lignes, colonnes. Cependant, l'étude ne nous informe pas comment choisir les fonctions élémentaires.

Le calcul d'adresse est une expression logique d'opérations $\bigoplus$, il est d'une complexité $3 \log N$. Sur le Cray-1, Weiss a estimé le calcul d'adresse à deux cycles en mode pipeline.

Sur les machines tableaux, il est difficile de lui adapter un réseau de réalignement autre que le crossbar.

Les performances ne sont pas meilleures que celles de la méthode d'entrelacement séquentiel (section (1.3.2.1)). Elle est environ 60% moins performante que la méthode séquentielle surtout pour des accès par pas impair. L'auteur pense que si elle est associée avec la méthode des files d'attentes, les performances seront améliorées.

D'après les simulations effectuées par Weiss, même en association avec la méthode des files d'attentes elle sera, au mieux, aussi performante que la méthode d'entrelacement séquentiel.

En conclusion nous dirons que cette méthode apporte un formalisme original, mais qu'elle n'apporte pas de solution au problème de conflits dus à la périodicité de la méthode d'entrelacement séquentiel lorsque le temps de latence est assez grand.

C– Méthode aléatoire

Les méthodes précédentes et plus particulièrement les méthodes pseudo-aléatoires, distribuent les données de manière uniforme sur l'ensemble de la mémoire de façon à favoriser l'accès aux sous-structures qui nous intéressent. Cette distribution n'est efficace que si le temps de cycle de la mémoire est égal à celui des processeurs. Dans le cas contraire, surtout sur les machines pipelines, on ne peut pas référencer un banc qui est déjà en activité, ce qui entraîne des conflits de bancs. Par conséquent la dégradation des performances en utilisant ces techniques est inévitable.

Weiss a proposé une solution en se basant sur la propriété d'apériodicité. Il s'est avéré que cette propriété est insuffisante. La méthode suivante apporte la solution à ce problème [Raghavan et al.90]. Elle combine deux fonctions: **modulo** et **pseudo-aléatoire**.

Soit a une adresse initiale et b l'adresse du banc image de a. On suppose que le nombre de bancs $M = 2^m$ et que la mémoire est adressable en lower-order. L'adresse du banc est donnée par: $b = f(a_{n-m}) + a_m \pmod m$, où f est une fonction pseudo-aléatoire avec une distribution uniforme. a_{n-m} (resp. a_m) est le mot formé par les $n-m$ (resp. m) bits des n bits de l'adresse initiale a.

Principe

On suppose que l'on veuille ranger un vecteur par pas de 2 dans une mémoire à 8 bancs et possédant un temps de latence de 4 cycles. Les éléments d'une période (ensemble d'éléments de la structure qui référencent des bancs différents distancés par le pas d'accès) seront rangés par la fonction modulo, car les $n-m$ bits de poids forts sont inchangés. Comme la fonction f est bijective, la valeur de $f(A_{n-m})$ est la même pour tous les éléments d'une période. En fait la fonction f permet de positionner *"aléatoirement"* les adresses de base d'une période. L'exemple de la figure (1.13) en est une illustration.

M0	M1	M2	M3	M4	M5	M6	M7
a_0			a_1			a_2	
		a_3			a_4		
	a_5			a_6			a_7
a_8			a_9			a_{10}	

Figure 1.13 : Distribution d'adresses suivant un schéma pseudo-aléatoire de Raghavan & al.

Ce schéma est périodique et respecte l'ordre dans lequel les bancs ont été assignés durant les périodes antérieures. Les auteurs ont proposé pour $f(a)$ une fonction de génération aléatoire simple qui est : $f(a) = \lfloor M\ ((\frac{C}{n}a) \pmod 1)\rfloor$. C est constant et choisi de façon qu'il soit premier avec n [Knuth73]. Sachant que (x mod 1) donne la partie fractionnaire de x.

Réflexions

- Ce schéma de rangement donne de très bonnes performances par rapport aux méthodes pseudo-aléatoires de Cydra et des rangements linéaires de IBM RP3 lorsque le temps de latence est assez grand.
- Comparé à la méthode Cydra, elle est moins performante pour des temps de latence assez petits (≤ 8), ce qui prouve que la fonction pseudo-aléatoire de Cydra est plus uniforme que $f(x)$.
- Le calcul d'adresse est complexe car il nécessite deux multiplications et plusieurs opérations de décalages. Son implémentation hardware est un pipeline de 5 étages. Ceci peut être tolérable sur une machine vectorielle pipeline.

- Le réseau de réalignement de données n'est pas étudié. Pour les machines tableaux, il est très difficile d'adapter un réseau de réalignement autre que le crossbar.
- La méthode ne garantit pas l'accès aux sous-structures courantes qui nous intéressent (lignes, colonnes, diagonales, etc.).

1.3.2 Techniques globales

Elles s'intéressent à l'organisation de l'ensemble de l'espace mémoire indépendamment d'une structure de données à laquelle il faut accéder et de son utilisation à chaque instant.

Ces techniques ne sont pas très développées par rapport aux techniques locales. Elles doivent leurs mérites à la méthode d'entrelacement séquentiel. Cette méthode est globale et elle présente des conflits d'accès mémoires. Nous présenterons aussi deux autres types de techniques qui sont l'utilisation d'un nombre de bancs premier et les techniques désordonnées.

1.3.2.1 Principe d'entrelacement

La majorité des architectures à mémoires parallèles utilisent la technique d'entrelacement "classique", ou ce qu'on appelle l'entrelacement séquentiel [Rau et al.89b]. Cette méthode fut l'une des premières à être utilisée pour accéder à des mémoires parallèles. Etant donnée une donnée d'adresse a, celle ci est rangée dans le module mémoire d'adresse (a mod M), où M est le nombre total de bancs. Pour faciliter le calcul de l'adresse du banc, M est toujours choisi parmi les puissances de 2 ($M = 2^m$). L'adresse d'une donnée est composée de deux champs (voir la figure (1.14b)) : un champ de m bits de poids faible représente l'adresse du banc et les bits restant de poids fort indiquent l'adresse de la donnée dans le banc.

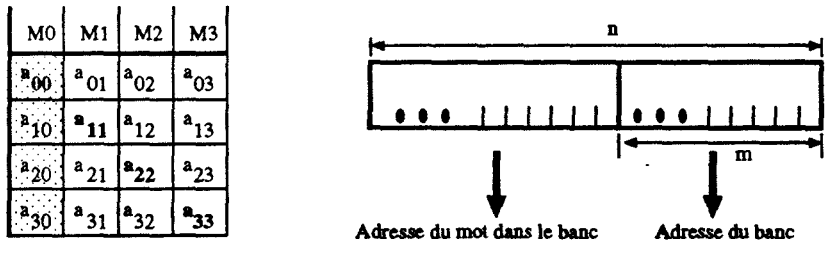


Figure 1.14 : Génération d'adresses en méthode séquentielle.

L'accès à des éléments d'un tableau produit une bande passante maximale lorsque le pas d'accès (S) et M sont premiers entre eux. Dans ce cas, les requêtes mémoires sont uniformément distribuées sur les bancs. Cependant cette technique est sujette à des dégradations de performances lorsque le pas de déplacement est un multiple de 2 et surtout un multiple de M. Dans ce dernier cas, toutes les requêtes concernent le même banc mémoire, et le débit de la mémoire est limité à une donnée par cycle mémoire. Tout accès à une colonne de la matrice de la figure (1.14a) génère un accès par pas multiple de M. L'accès à des sous-tableaux $T[I, J]$ de la structure du tableau principal ($N \times N$) (ligne, colonne, diagonales ou blocs carrés ($\sqrt{N} \times \sqrt{N}$)), a entraîné la mise en oeuvre d'autres méthodes que nous avons appelé ici les techniques locales.

Les performances de cette méthode dépendent fortement de l'habilité du programmeur à développer ses algorithmes en ne générant que des pas de déplacement impairs. A titre d'exemple, pour effectuer la multiplication de deux matrices, on a intérêt à ranger la première matrice ligne par ligne et la seconde colonne par colonne. Les accès à la mémoire seront réalisés sans conflits, d'où un débit maximum. Comme les méthodes précédentes, elle souffre aussi de chutes de performances dues aux accès dispersés contrôlés par un vecteur d'index ou par un bit-vecteur (gather/scatter ou merge/compress). Contrairement à notre intuition qui nous pousse à croire que les techniques locales sont plus efficaces, cette méthode donne de meilleurs résultats que les techniques locales sur les machines vectorielles pipelines à mémoire parallèles [Chen et al.89].

1.3.2.2 Principe du nombre premier de bancs

Si on considère que le nombre de bancs est assez grand par rapport au temps de latence de la mémoire, un banc déjà référencé a de grandes chances d'être libre au moment de la prochaine référence. La bande passante, dans ce cas, est : $B_m = M/pgcd(M,S)$. Si M et S ne sont pas premiers entre eux, les bancs mémoires ne sont pas tous référencés, d'où une diminution du débit. La valeur du pas S doit être judicieusement choisie pour éviter des chutes de performances relativement importantes. En choisissant M premier, cela augmente le nombre de possibilités d'avoir un débit élevé, il est maximum pour toutes les valeurs de S qui ne sont pas multiples de M. Les performances maximales sont régulièrement atteintes [Lawrie et al.82], cependant, cette méthode présente deux inconvénients majeurs :

- La non symétrie du réseau d'interconnexions.
- La complexité de calcul de l'adresse du banc ($A \bmod M$). Néanmoins, si M est de la forme $2^m \pm 1$, les calculs peuvent être simplifiés [dD91].

Ce principe est utilisé pour le système mémoire de la machine BSP [Kuck et al.82], suggérées par Burroughs pour le projet NASF, qui consiste en 17 bancs mémoires et 16 processeurs. C'est une machine multiprocesseur SIMD, successeur d'ILLIAC IV [Barnes et al.68], [Hwang et al.84]. Elle présente la particularité de posséder un réseau de réalignement qui effectue des opérations Compress/Extend. Cependant son problème majeur est la complexité de calcul de l'adresse du banc.

1.3.2.3 Techniques d'accès désordonnés

1.3.2.3.1 Principe des files d'attentes

Aucun schéma de rangement ne peut garantir une bande passante élevée lorsque les requêtes mémoires sont générées par pas irréguliers, comme c'est le cas des opérations gather/scatter ou merge/compress. Le débit est approximativement proportionnel à $\sqrt{M}$ (M étant le nombre de bancs mémoires) [Knuth et al.75]. Cependant, si le système mémoire est pourvu de files d'attentes pour ranger les requêtes qui référencent les bancs occupés, on pourra atteindre la bande passante optimale. Le modèle multiprocesseur idéal (pour garantir le débit optimal lors d'accès aléatoires) est d'installer une file d'attente en chaque point critique d'une ressource comme un banc mémoire ou une ligne du réseau.

Ce principe est utilisé dans Cydra-5 [Rau et al.89a] (figure (1.15)). C'est une machine multiprocesseur hétérogène à fonctionnement Data-flow. Cette machine est composée de deux types

de processeurs : les processeurs interactifs et le processeur numérique. La description de l'architecture de la machine est donnée dans [Rau et al.89b, Rau et al.89a]. Ce qui nous intéresse est non pas la machine elle-même mais l'architecture de son système mémoire.

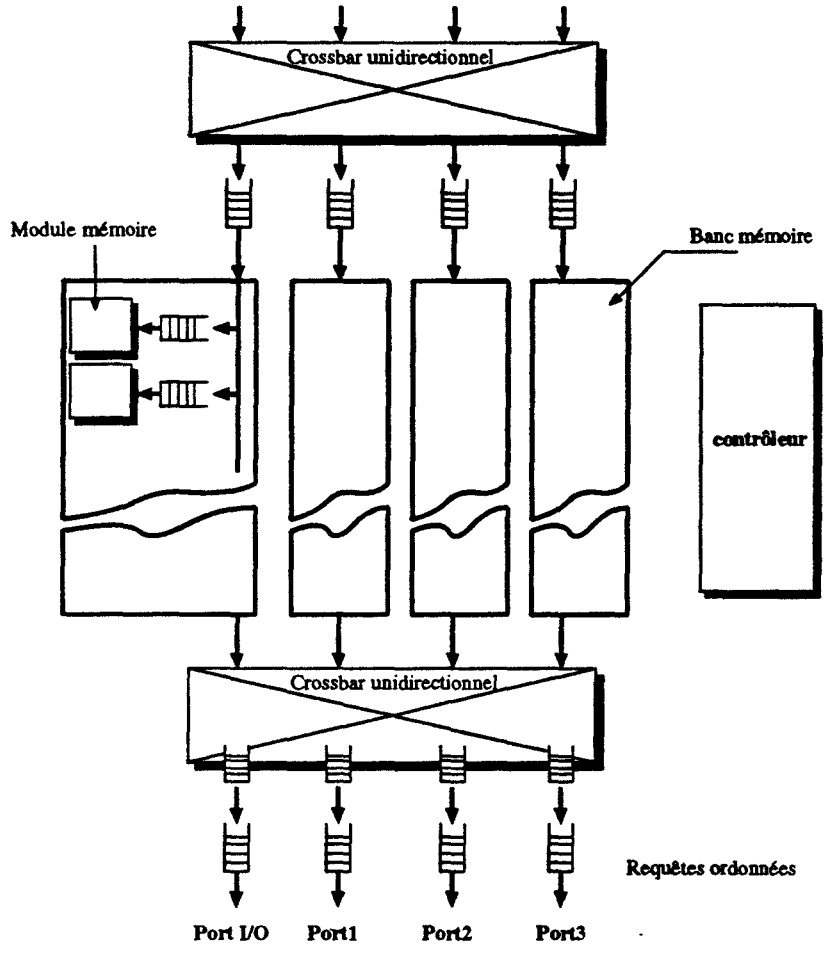


Figure 1.15 : L'organisation mémoire de la machine Cydra-5.

Son système mémoire est presque identique à celui du Cray X-MP. La mémoire est partagée en 4 bancs (appelé section sur le X-MP) et chaque banc peut contenir de 2 à 16 modules (appelé bancs sur le X-MP). Deux crossbars unidirectionnels (l'un en entrée et l'autre en sortie) relient les 4 bancs aux 4 ports (3 ports du processeur numérique et un port pour le processeur interactif). La figure (1.15) montre l'organisation de cette mémoire. Les requêtes mémoires sont rangées dans les files d'attentes si les bancs sont occupés sinon elles arrivent directement dans les files d'attentes des modules associés. Cependant, l'ordre des données n'est pas garanti à la sortie des bancs mémoires, par conséquent, les données sont ordonnées avant d'être envoyées aux processeurs. Ce principe de réordonnancement des requêtes à la sortie de la mémoire est aussi employé dans le processeur DSPA [Jegou86] et dans la machine DAE [Bird et al.91].

Malgré de bonnes performances, cette technique présente les inconvénients suivants :

1. Problème de réordonnancement des données en sortie. La perte de performance est évaluée dans le chapitre suivant.
2. Complexité du système de FiFos dans une architecture multiprocesseur (une FiFo devient une ressource critique si plusieurs processeurs référencent la même ligne du réseau ou le même banc.
3. Complexité de contrôle des FiFos.
4. Problème du choix de la taille des FiFos.
5. Gestion de la saturation des files d'attentes.

1.3.2.3.2 Principe du Cray-2

Nous avons pris l'exemple du Cray-2 [Schonauer87], car, si cette machine est considérée comme une innovation technologique pour son époque (une machine très compacte, multiprocesseur vectoriel pipeline avec une très large capacité mémoire et des processeurs très rapides) elle présente des faiblesses importantes dues à la méthode d'accès mémoire qui utilise le principe des accès désordonnés.

Le C-2 est composé de 4 processeurs (Cpus). Le système mémoire est divisé en 4 quadrants de 32 bancs chacun. Chaque processeur possède un seul port mémoire, ce qui réduit le débit des échanges avec la mémoire et les entrées/sorties. Les quadrants sont reliés aux processeurs via un réseau spécialement conçu pour le C-2. Il est la source de la méthode d'accès mémoire. De manière périodique, chaque Cpu peut accéder un seul quadrant bien précis. La table de la figure (1.16) montre la manière dont les quadrants sont alloués. Par exemple, durant le i^{me} cycle les quadrants q_1, q_2, q_3 et q_4 sont attribués aux processeurs Cpu1, Cpu2, Cpu3 et Cpu4 respectivement, au cycle suivant, q_1, q_2, q_3 et q_4 seront attribués aux Cpu2, Cpu3, Cpu4 et Cpu1 et ainsi de suite. Le Cpu tente de produire une requête à un des bancs mémoires du quadrant qui lui est alloué indépendamment de l'ordre des requêtes en attente d'accès. Les données sont réordonnées à la sortie de la mémoire avant d'entamer tout traitement, car l'ordre d'accès des requêtes n'est pas garanti.

Cycle	Cpu1	Cpu2	Cpu3	Cpu4
i	q_1	q_2	q_3	q_4
i+1	q_4	q_1	q_2	q_3
i+2	q_3	q_4	q_1	q_2
i+3	q_2	q_3	q_4	q_1

Figure 1.16 : Réservation des quadrants par les Cpus du Cray-2.

Cette organisation a deux inconvénients majeurs :

- Le réordonnancement des données à la sortie de la mémoire, qui provoque la rupture du chaînage des pipelines de calcul.
- Les conflits d'accès qui sont de deux types :

- **les conflits de quadrants**: apparaissent lorsque les requêtes d'un processeur ne sont destinées à aucun des bancs qui se trouvent dans le quadrant réservé pendant un cycle donné.
- **les conflits de bancs occupés**: le cycle mémoire du C-2 dure 37 cp (61 cp sur la première version), toute référence à un banc en activité pendant x cycles ($x < 37$) occasionne un conflit de banc occupé. *cp* étant le cycle processeur.

A cause de ces conflits, le C-2 subi de dramatiques dégradations de performances lorsqu'un Cpu accède un vecteur par pas pair ou multiple de 4 (un seul quadrant et 8 bancs parmi 32 sont sollicités) et lors d'accès dispersés (par pas irréguliers).

1.3.2.3.3 Conclusion sur les méthodes désordonnées

La méthode idéale (qui consiste à éviter les conflits mémoires) est certainement la méthode des files d'attentes. Cependant, en plus de la complexité de réalisation et les difficultés de contrôle d'un tel système, elle crée des délais d'attentes devant un réseau de files d'attentes.

La méthode du Cray-2 est en quelque sorte une méthode de files d'attentes centralisée, car il ne dispose pas de queues en chaque point critique. Chaque Cpu dispose de sa propre et unique file d'attente. Effectivement le problème d'implémentation est résolu, mais les conflits introduits par l'architecture persistent. C'est à dire qu'il n'implémente pas de technique de répartition des accès sur l'ensemble des processeurs.

Il reste le problème d'identification des données à la sortie de la mémoire qui est jusqu'à présent résolu en réordonnant ces données. Evidemment cette opération est coûteuse en temps et provoque la rupture des chaînages des pipelines des unités fonctionnelles.

Dans le chapitre suivant nous allons proposer une nouvelle technique désordonnée qui résout le problème de rupture du chaînage et les problèmes des conflits d'accès mémoires.

1.4 Critiques et conclusion

Après avoir fait un aperçu sur les méthodes existantes, nous concluons que la capacité d'échange de données entre les processeurs et les mémoires pose énormément de problèmes aux concepteurs des machines multiprocesseurs.

Nous venons de voir deux classes de méthodes destinées à améliorer les performances des systèmes multiprocesseurs au niveau des échanges de données entre les processeurs et les mémoires. Les techniques locales sont mieux adaptées pour les machines tableaux. Elles apportent une solution au problème de répartition des données en mémoire mais elles causent de nouveaux problèmes tels que l'association du réseau de réalignement, le calcul d'adresses, la perte d'espace mémoire, etc.

Les techniques globales sont plutôt mieux adaptées aux multiprocesseurs vectoriels pipelines. Certaines d'entre elles causent la rupture du chaînage des pipelines de calcul, et leur implémentation n'est pas simple.

Ces deux techniques ne sont pas exclusives. L'une des solutions qui pourrait augmenter la capacité de transfert de données entre les processeurs et les mémoires est l'association d'une méthode locale avec une méthode globale; c'est le cas de Cydra-5. Le tableau (1.17) résume les points forts et les point faibles de chaque méthode étudiée ici.

Technique	Type	Accès				Type de machines	Complexité		
		lignes	colonnes	diagonales	blocs carrés		calcul d'adresses	réseau	autres
Budnick & al.	L (linéaire)	—	—	—	—	Tableaux	Facile	Crossbar	
Harper III	L (linéaire)	—	—	—	—	Tableaux	Facile	Crossbar	
Lawrie	L (linéaire)	oui	oui	oui	oui	Tableaux	Facile	Omega	perte d'espace mémoire
Pseudo-aléatoire	L (non-linéaire)	oui	oui	oui	oui	SIMD	Facile	Crossbar	
Carré Latin	L (non-linéaire)	oui	oui	oui	oui	SIMD	Facile	Crossbar	
Permut. Linéaire	L (non-linéaire)	oui	oui	oui	oui	Tableaux	Facile	Omega	Conflits de réseau
Brouillée	L (non-linéaire)	oui	oui	oui	oui	Tableaux	Facile	Theta	Conflits de réseau
STARAN	L (non-linéaire)	oui	oui	non	—	Tableaux	Facile	spécial	
Apériodique	L (non-linéaire)	oui	oui	oui	oui	Vectorielles	Facile	Crossbar	
Aléatoire	L (non-linéaire)	—	—	—	—	Vectorielles	+/- Facile	Crossbar	
Entrelacement	Globale	oui	non	non	non	SIMD	Facile	Crossbar	
n. bancs premier	Globale	oui	oui	oui	oui	SIMD	Complexe	Crossbar	
Files d'attentes	Globale	—	—	—	—	SIMD	Facile	Crossbar	Réordonnancement
Cray-2	Globale	—	—	—	—	Vectorielles	Facile	Crossbar	Réordonnancement

SIMD = vectoriel pipeline + tableau

Figure 1.17 : Tableau récapitulatif des différentes techniques de résolution des conflits.

Chapitre 2

Le Traitement vectoriel désordonné

Les différentes méthodes de résolution des conflits d'accès mémoires ont été présentées dans le chapitre précédent. Cela nous a permis de dégager les avantages et les inconvénients d'une méthode par rapport à l'autre et, plus précisément, des méthodes à accès désordonné.

Dans ce chapitre nous proposons notre modèle de traitement vectoriel pour les machines vectorielles pipelines multiprocesseurs. Ce modèle apporte des solutions au problème des différents conflits introduit par l'architecture de ces machines, et améliore leurs performances [Dekeyser et al.92b], [Dekeyser et al.92a]. Soulignons que ce modèle n'est pas seulement une méthode d'accès mémoire, mais il est aussi une méthode de calcul dans les unités fonctionnelles du processeur. Ces deux méthodes sont complémentaires. La méthode d'accès produit un gain de performances en évitant les conflits et la méthode de calcul conserve ce gain en permettant le chaînage des pipelines, l'absence de celui-ci étant à l'origine de perte de performances dans les systèmes comme Cydra ou Cray-2.

Nous commencerons par expliquer le mécanisme d'adressage des vecteurs, afin de justifier notre mécanisme. Nous présenterons ensuite le modèle de traitement désordonné et nous montrerons son fonctionnement sur des exemples. Dans les sections IV et V nous verrons comment les différents conflits sont résolus ainsi que les stratégies de choix des requêtes non conflictuelles. Pour des raisons de clarté nous n'abordons pas ici l'implantation du modèle, ceci sera traitée dans le chapitre suivant.

2.1 Architecture de référence

Nous avons vu dans le chapitre précédent qu'il n'existe pas de méthode universelle qui répond au problème de chute de performances des systèmes multiprocesseurs dues aux conflits d'accès mémoires. Les méthodes présentées apportent des améliorations dans certains cas seulement, car elles dépendent intrinsèquement de deux choses :

- de l'architecture de la machine,
- du type d'accès : pas réguliers et pas dispersés.

Comme pour ces méthodes, le traitement vectoriel désordonné vise une architecture bien particulière qui est celle des calculateurs vectoriels pipelines multiprocesseurs à mémoire commune. Ces calculateurs ont les caractéristiques suivantes :

- Les processeurs sont très puissants; ils possèdent essentiellement :

- plusieurs unités fonctionnelles pipelines, vectorielles et scalaires pouvant travailler en parallèle. Ces unités peuvent être chaînées.
- plusieurs ports d'accès mémoires afin d'augmenter leur débit.
- plusieurs registres vectoriels.

- Le système mémoire est un ensemble de bancs mémoires indépendants pour satisfaire les accès parallèles.
- Le réseau d'interconnexions relie tous les processeurs à tous les bancs mémoires de façon uniforme. Ce réseau doit avoir un très haut débit et présenter le moins de conflits possible.

Ce type de "super-processeurs" se doit d'être efficace s'il maintient le flux de données vectorielles entre la mémoire et les unités fonctionnelles continuellement actif. Le flux de données est tout ce qui est mouvement de données à l'intérieur du calculateur. Il se divise en deux (voir la figure (2.1)): un flux d'accès mémoires qui s'occupe du chargement et du déchargement des données vers ou depuis les registres vectoriels. Le second flux, flux de calcul, s'occupe du traitement des données contenues dans les registres vectoriels. Ces deux flux travaillent en mode pipeline. Le flux d'accès est caractérisé par la façon dont les accès sont effectués, donc par la méthode d'accès mémoire. Le flux de calcul est caractérisé par le modèle d'exécution. On retrouve ce mode de traitement sur la majorité des machines vectorielles pipelines (Cray Y-MP, X-MP, Nec SX3, VP200, etc.).

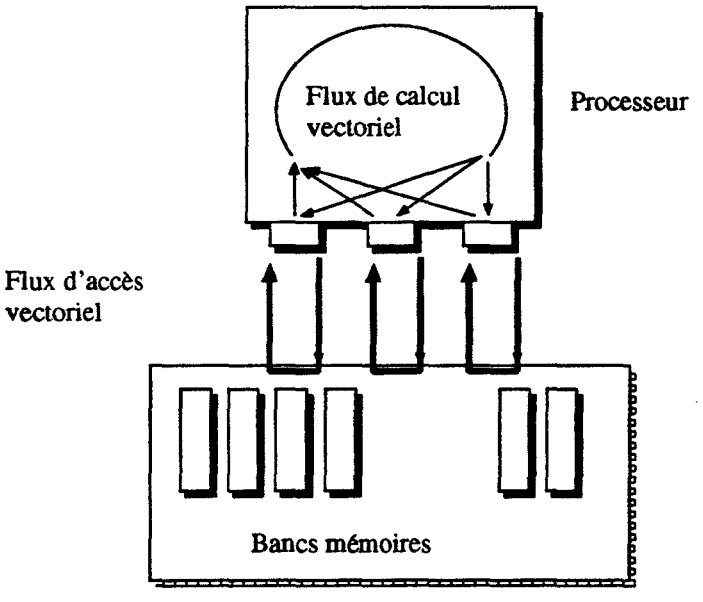


Figure 2.1 : Le flux de données vectorielles

2.1.1 Mécanisme d'adressage des vecteurs

Un vecteur est un n-uple de nombres : $(x_1, x_2, \cdots, x_n)$, avec n le nombre d'éléments du vecteur. A un élément (composante) du vecteur on associe une valeur et une position (rang) unique dans le n-uple. Tout traitement effectué sur un vecteur utilise cette notion de position pour

identifier ses éléments. Une composante d'un vecteur est un couple de données (position, valeur). Cette position est définie de plusieurs manières, suivant le type de machines:

- **Machines mémoire-à-mémoire:** le déplacement par rapport à l'adresse de base du vecteur en mémoire centrale.
- **Machines massivement parallèles:** le numéro du PE et de son adresse dans la mémoire du PE.
- **Machines registre-à-registre:** On distingue deux cas:

  – Lorsque le vecteur est en mémoire, la définition est la même que celle des machines mémoire-à-mémoire.

  – Lorsque le vecteur est dans un registre vectoriel, l'adresse d'un élément est donnée par le nom du registre et l'adresse de l'emplacement de la composante dans le registre. Cette position est appelée **index**. Dans ce qui suit l'index de la composante sera son adresse dans le registre vectoriel.

Durant le traitement, dans un modèle classique pipeline, l'index de la composante est implicite. C'est-à-dire que le calculateur connaît, à l'aide de l'index de la composante courante, l'index de la prochaine composante car le traitement se fait dans l'ordre croissant des index. Par conséquent, on ne manipule que les valeurs des composantes. Ce modèle permet une implémentation simple et un hardware moins coûteux , mais il impose l'ordre de traitement des composantes ($x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n$).

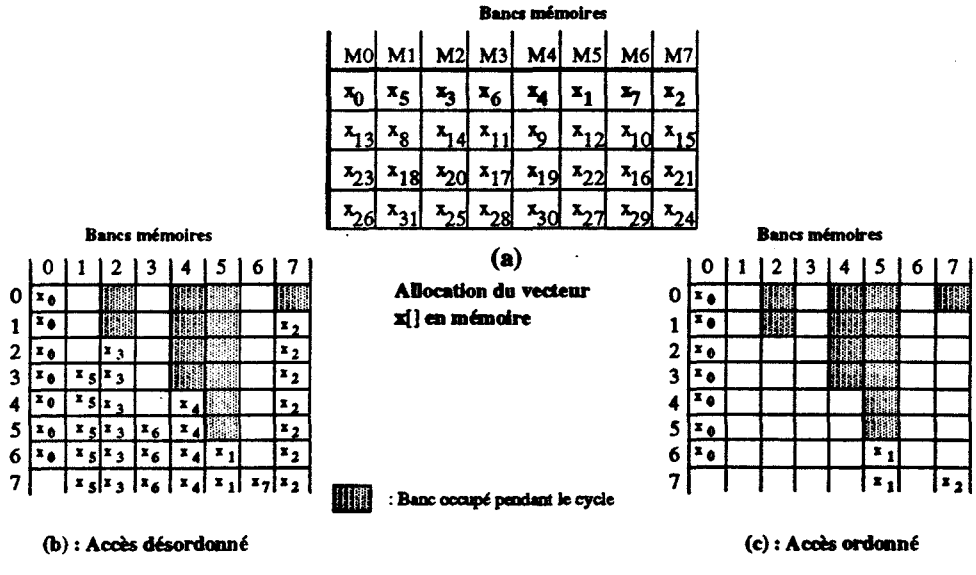


Figure 2.2: Conflits d'accès et leurs conséquences

A partir du moment où il y a conflit (mémoire ou réseau occupé ou accès concurrents) la composante qui a provoqué le conflit est bloquée et elle bloque toutes les composantes qui suivent à cause de la contrainte d'ordre d'accès. Souvent on peut accéder à ces composantes

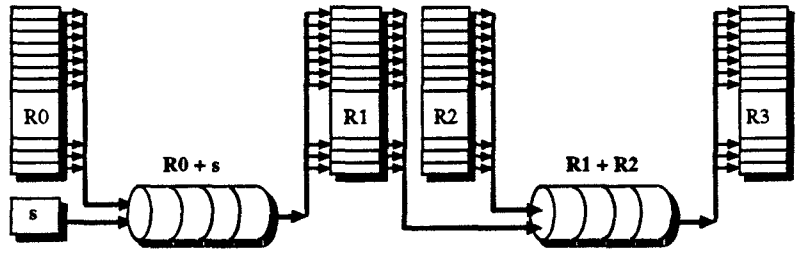
sans conflit et au niveau calcul, il n'y a pas d'inconvénient à traiter une composante plutôt qu'une autre. L'exemple suivant en est une illustration :

Soit $A = (x_0, x_1, \cdots, x_{31})$ un vecteur rangé en mémoire comme le montre la figure (2.2a). On suppose qu'au moment de l'accès certains bancs mémoires sont occupés. On n'accédera qu'aux 7 premières composantes. On accède à x_0 car M_0 est libre. Au cycle suivant M_5 est encore occupé; x_1 reste bloqué pendant 5 cycles. $x_i, (i = 2..7)$ seront bloqués tant que x_1 est en attente d'accès. Ainsi ce conflit a causé une perte de 5 cycles. Cependant si on avait accédé à ces composantes dans cet ordre : $(x_0, x_2, x_3, x_5, x_4, x_6, x_1, x_7)$, (voir la figure (2.2b)), il n'y aurait pas eu de conflits et les performances seraient maximales. Dans les chapitres *"modèle analytique"* et *"simulation"* nous avons évalué cette perte de performance et nous avons montré combien on peut gagner par rapport au modèle classique en appliquant notre modèle.

En traitement vectoriel désordonné, on favorise le traitement en se basant sur le principe suivant : *"traiter une composante comme entité indépendante"*.

2.1.2 Le chaînage

Le chaînage des unités fonctionnelles est une opération capitale dans les calculateurs vectoriels pipelines. Il permet à une unité fonctionnelle d'effectuer des opérations successives aussitôt que le résultat d'une autre unité devient disponible comme opérande. En d'autres termes, le registre vectoriel résultat d'une opération devient le registre source de l'opération de l'instruction suivante. La figure (2.3) illustre le fonctionnement de deux pipelines chaînés. Pour le bon déroulement du chaînage, les pipelines et les registres vectoriels sont proprement réservés.



Calcul de l'expression : (R0 + s) + R2. s: est un scalaire.

Figure 2.3 : Chaînage de deux pipelines

Suite à la définition des deux flux, on peut distinguer deux types de chaînage :

- Le chaînage au sein d'un flux de calcul vectoriel. Par exemple entre deux instructions qui opèrent sur des registres vectoriels :

  $\underline{R_2} \longleftarrow R_1 \; op \; R_0$
  $R_4 \longleftarrow R_3 \; op \; \underline{R_2}.$

  Ce type de chaînage existe sur presque toutes les machines vectorielles pipelines.

- Le chaînage des flux d'accès avec des flux de calcul. Par exemple une instruction qui opère sur des vecteurs opérandes qui sont en mémoire : $R_2 \longleftarrow V_0 + V_1$; c'est à dire

  $R_0 \longleftarrow @V_0$
  $R_1 \longleftarrow @V_1$

$$R_2 \longleftarrow R_0 + R_1$$

Ce type de chaînage se trouve sur les machines mémoire-à-mémoire (Star-100, Cyber-205, ...), Nec SX-3, sur toute la gamme des machines Cray sauf le Cray-2, etc.

Sur les machines à accès désordonné aux éléments d'un vecteur il y a absence totale du deuxième type de chaînage. C'est le cas du Cray-2, Cydra-5, DSPA et DAE. Les machines de ce type effectuent des accès désordonnés à la mémoire puis elles réordonnent les composantes des vecteurs à la sortie de la mémoire, puisque l'ordre d'émission des composantes n'est pas garanti.

Cette manipulation (accès désordonné + "réordonner") est coûteuse en temps, car il faut attendre la fin d'accès au vecteur, réordonner, puis exploiter le vecteur. Par rapport aux machines où les deux types de chaînage existent, une partie du gain en accès est perdu durant l'opération "réordonner". N'est-il pas plus performant et moins coûteux de réaliser des accès classiques (respecter l'ordre d'émission des composantes des vecteurs) et de garder le chaînage avec les unités fonctionnelles internes du processeur?

L'exemple des vecteurs contigus accessibles sans conflits en mémoire répond bien à la question. Car les deux modèles (classique et désordonné) effectuent des accès sans conflit. Dans ce cas là, la méthode d'accès désordonnée ne fait rien gagner. Cependant, en absence du chaînage, le modèle désordonné se révèle moins performant.

Dans notre modèle de traitement désordonné, le chaînage est rendu possible grâce à l'identification de la composante explicitement par son index et par sa prise en compte par le hardware.

La proposition suivante donne la borne inférieure du gain en temps d'exécution des méthodes désordonnées avec chaînage par rapport aux méthodes désordonnées sans chaînage.

Proposition 1 *Soit t_{ch} (resp. t_{sch}) le temps d'exécution d'un flux d'accès et d'un flux de calcul en modèle désordonné avec chaînage (resp. sans chaînage), alors:*

$$t_{sch} - t_{ch} \geq \frac{L}{2} c_p,$$

avec L la taille des vecteurs traités, et c_p, le temps de cycle du processeur.

Preuve :

Soit une instruction vectorielle suivante: $R_2 \longleftarrow V_1 + V_0$ où R_2 est un registre vectoriel et V_0, V_1 sont des vecteurs en mémoire. Cette instruction est composée de deux flux d'accès des deux vecteurs V_0, V_1 suivis d'un flux de calcul d'une opération à deux opérandes. On suppose que les vecteurs sont de taille L. Le cas le plus défavorable pour le modèle de traitement désordonné est le suivant:

Supposant que pendant un temps (t) cycles on n'a pu accéder qu'aux $(\frac{L}{2})$ éléments V_0^i, V_1^j qui ne sont pas conformes (aucun couple (V_0^k, V_1^k) ne s'est formé pour entamer le calcul, voir la figure (2.4)). Pendant ce temps (t) le pipeline de calcul n'est pas activé. Après avoir accédé aux $\frac{L}{2}$ éléments, le pipeline est initialisé à chaque arrivée d'une composante de V_0 et/ou V_1.

On suppose maintenant que pendant les cycles qui suivent (t) on a pu accéder à un élément par vecteur (V_0^i, V_1^j). Par conséquent les couples (V_0^k, V_1^k) se forment et on peut effectuer le calcul. Ainsi les temps d'exécutions pour les deux flux sont:

- **Avec chaînage:** $(t + \frac{L}{2} + \frac{L}{2})cp = (t + L)cp$, (cp est le temps de cycle des processeurs).

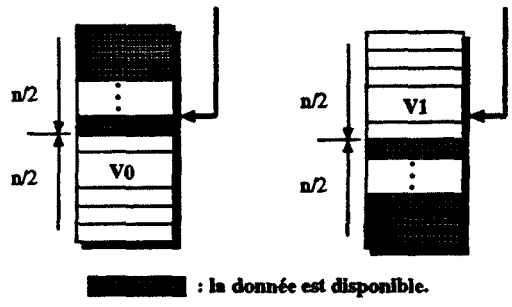


Figure 2.4: L'état des 2 registres opérandes après un temps (t).

- **Sans chaînage:** $(t + L + \frac{L}{2})cp = (t + \frac{3L}{2})cp$.

La différence est bien $\frac{L}{2}cp$. Elle constitue la borne inférieure du gain en temps du modèle désordonné avec chaînage sur le modèle désordonné sans chaînage.

2.2 Présentation du modèle

Le modèle de traitement vectoriel désordonné reste invariant sur le mode de fonctionnement global des deux flux, il agit sur le séquencement des opérations dans les deux flux. Il est une méthode d'accès pour augmenter le débit des flux d'accès, et il est un modèle d'exécution pour maintenir ce débit continuellement à haut niveau.

2.2.1 Méthode d'accès

La méthode d'accès est basée sur les deux règles suivantes:

1. Identification des requêtes auxquelles on peut accéder sans conflits.
2. Accès à un nombre maximum de requêtes par processeur (une requête par port) à chaque cycle, sans tenir compte de l'ordre des index, c'est-à-dire, en respectant le principe d'entité indépendante cité précédemment.

La première règle peut être réalisée en deux étapes pour séparer le traitement purement parallèle (sans communication) du traitement avec communications: a) on identifie les requêtes libres (non conflictuelles) localement à chaque processeur (ce traitement s'effectue en parallèle sur tous les processeurs et sans communication). b) on identifie (si possible) les requêtes qui sont susceptibles de causer des conflits inter-processeur. Cette étape est la plus difficile à réaliser car elle exige des communications entre les processeurs, ce qui est très coûteux en temps d'exécution. Dans la dernière section nous proposons une classe de techniques de sélections plus ou moins efficaces suivant la configuration de la machine.

La deuxième règle optimise les accès, en répartissant de manière uniforme les requêtes sur les ports de chaque processeur afin d'éviter le plus possible de conflits.

2.2.2 Modèle d'exécution

Il présente deux aspects : aspect calcul par disponibilité et aspect chaînage des unités fonctionnelles de calcul.

Premier aspect : Pendant l'accès, les données retournées par la mémoire sont rangées dans les registres vectoriels alloués aux vecteurs. Nous n'allons pas attendre la fin d'accès de tout le vecteur pour entamer le calcul. A la différence des autres machines à accès désordonné, nous commencerons aussitôt que les données sont **disponibles** et **conformes**. Une composante est **disponible** si elle est chargée dans un registre vectoriel. Deux composantes de deux vecteurs distincts sont **conformes** si elles ont le même index. Ceci nous conduit à dégager les propriétés ci-dessous.

Deuxième aspect : Toutes les unités fonctionnelles de calcul peuvent être chaînées à condition qu'elles exécutent des instructions sans dépendances.

2.2.3 Propriétés du modèle

- Les accès sont effectués dans un ordre quelconque. Chaque composante est indépendante des autres, et son accès ne dépend que de l'état de la mémoire et de l'état du réseau d'interconnexions.
- La composante est identifiée explicitement par son index durant tout le traitement. Puisque l'ordre d'accès n'est pas connu à l'avance, tout élément du vecteur doit être identifié par son index afin de le ranger au bon emplacement dans le registre vectoriel cible.
- Maintenir le chaînage des deux flux.
- Le traitement désordonné est présent dans toutes les unités fonctionnelles de la machine. En accès ou en exécution les éléments des vecteurs sources sont traités dans un ordre aléatoire et cela est dû aux trois raisons suivantes :
  - Les composantes sont indépendantes de l'ordre dans le vecteur.
  - Le chaînage de toutes les unités fonctionnelles.
  - Spécification des index des composantes de manière explicite.
- La méthode d'accès produit un gain par rapport au modèle classique, en minimisant les conflits. L'analyse de performances et résultats de simulations confirmeront cette propriété,
- Le modèle d'exécution conserve ce gain, en rendant possible le chaînage du flux d'accès avec le flux de calcul.

Dans ce qui suit, nous allons voir le fonctionnement du modèle sur le flux d'accès et le flux de calcul.

2.2.4 Flux d'accès

D'une manière générale, le flux d'accès vectoriel est une succession d'accès mémoire à des éléments d'un vecteur. Le nombre d'accès est égal au nombre d'éléments dans le vecteur. Ce

flux est déclenché par les instructions d'accès mémoire (Load/Store, Gather/Scatter), et il permet de charger (resp. de décharger) les registres vectoriels en lecture (resp. en écriture). Un flux d'accès est terminé si tous les accès aux éléments du vecteur ont été effectués. En d'autres termes, la terminaison est détectée si tous les éléments sont chargés dans le registre vectoriel cible (en lecture) ou s'ils sont tous rangés en mémoire (en écriture).

En traitement vectoriel désordonné, lorsqu'une instruction d'accès mémoire est exécutée, l'unité de calcul d'adresse reçoit les caractéristiques du vecteur qui sont: l'adresse de base, le type du vecteur (contigu, steppé, ou gather/scatter), la taille du vecteur et le vecteur d'index s'il s'agit d'une opération gather/scatter. Cette unité calcule une adresse par cycle qu'elle range dans un buffer. La figure (2.5) montre les unités et les chemins d'accès concernés par ce flux. La méthode de traitement vectoriel désordonné consiste à explorer les buffers des ports de chaque processeur afin d'extraire les **requêtes** qui ne causent pas de conflits. La résolution des conflits dépend essentiellement de l'architecture et de la topologie de la machine. Evidemment cela suppose que le processeur dispose de quelques informations comme l'état de la mémoire et du réseau et d'un système de priorité entre les ports et les processeurs.

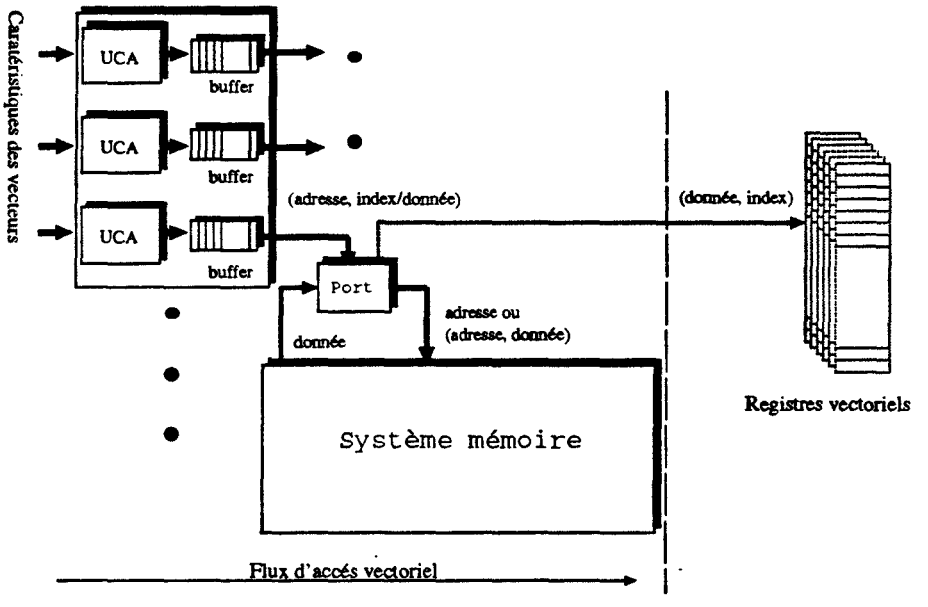


Figure 2.5: Le flux d'accès vectoriel

Ce modèle favorise l'échange de données entre les processeurs et les mémoires car chaque processeur tente le maximum de requêtes par cycle qui ne causent pas de conflits. L'ordre d'accès aux composantes n'étant pas respecté, la composante doit être identifiée explicitement soit par son index dans le registre vectoriel, soit par son adresse mémoire selon les deux cas:

- **Accès en lecture:**

  Lors du chargement des éléments d'un vecteur dans un registre vectoriel, la requête formulée à la mémoire est le couple (adresse, index). Ce couple est composé de l'adresse de l'élément en mémoire pour le chargement de la donnée et de son index pour ranger la donnée au bon emplacement dans le registre vectoriel.

- **Accès en écriture:**

La requête envoyée à la mémoire est le couple (adresse, donnée). La donnée est rangée à l'emplacement désigné par *"adresse"*.

Le registre vectoriel à ranger en mémoire ou le registre d'index pour les opérations gather/scatter peut être la cible d'une unité fonctionnelle. Le rangement des éléments de ce vecteur est effectué au fur et à mesure qu'ils sont disponibles dans le registre vectoriel. L'index de la composante intervient non seulement dans l'identification de son emplacement dans le registre vectoriel mais aussi dans le calcul de l'adresse mémoire.

Dans ce qui suit nous appelons par requête le couple **(adresse, index)** en lecture et le couple **(adresse, donnée)** en écriture.

Le gain en performances par rapport au modèle classique est obtenu durant le flux d'accès car c'est là où s'effectue la résolution des conflits.

2.2.5 Flux de calcul

Le flux de calcul vectoriel est produit durant le traitement effectué sur les composantes des vecteurs opérandes. Ce flux correspond au transfert des éléments des vecteurs opérandes entre les registres vectoriels et les pipelines de calcul ou entre les registres et l'unité de calcul d'adresses s'il s'agit d'une écriture. La figure (2.6) montre les différentes unités concernées par ce flux.

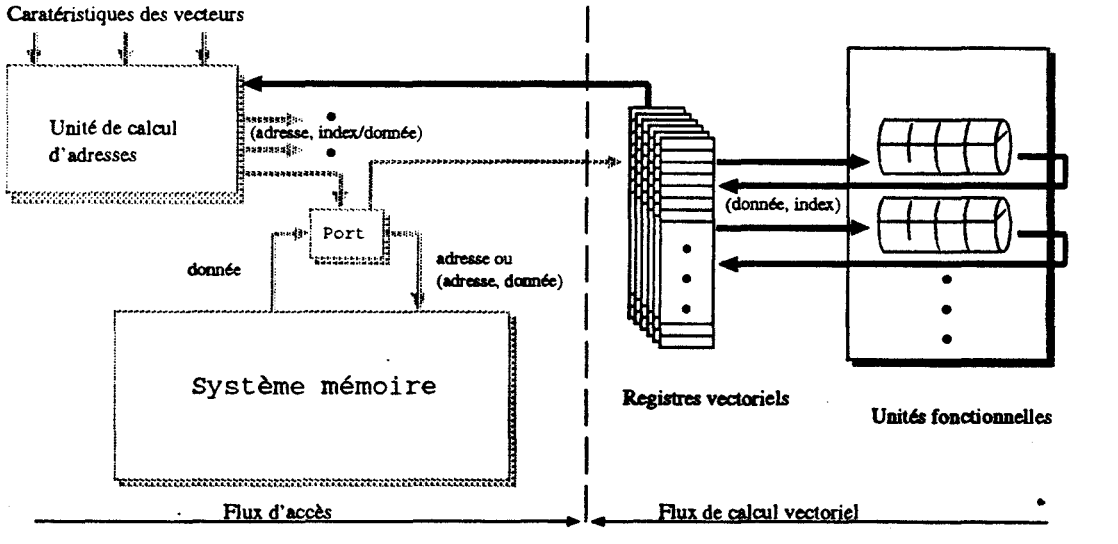


Figure 2.6 : Le flux de calcul vectoriel

Afin de conserver le gain réalisé grâce au flux d'accès, le chaînage de ce flux avec le flux d'accès est nécessaire. Pour cela, le déclenchement des opérations sur les composantes des vecteurs opérandes est réalisé une fois que ces composantes sont disponibles et conformes dans les registres vectoriels sources. Ainsi l'ordre de traitement des données est aussi "aléatoire"; on ne connaît pas à l'avance leur ordre de traitement.

Le déclenchement d'une opération sur les composantes des vecteurs opérandes s'accompagne par la production de leur index. Ce dernier est propagé à travers tous les étages des unités

fonctionnelles, dans le cas d'une opération de calcul, ou simplement il est transmis à l'unité destination dans le cas d'une opération de transfert.

La production de cet index est justifiée par :

- Si le traitement est une opération de calcul, l'index sert à ranger la donnée résultat dans le registre vectoriel cible.
- Si c'est uniquement un transfert d'un registre vectoriel vers l'unité d'accès mémoire, cet index sert à calculer l'adresse mémoire de l'élément.

2.3 Exemples de traitement

Les deux exemples suivants sont choisis pour mettre en évidence le fonctionnement du modèle de traitement désordonné et montrer son intérêt comparé au modèle classique.

Le premier exemple met en évidence deux types de conflits seulement : les conflits de bancs occupés et les conflits de sections. C'est une configuration monoprocesseur. Dans le deuxième exemple le programme est exécuté sur une configuration à deux processeurs.

2.3.1 Accès par pas irréguliers

On considère l'expression vectorielle : $C = A$ "*op*" B, où A, B, C sont des vecteurs indépendants en mémoire. Cette expression est exécutée comme suit :

```
Vload @A,          R0
Vload @B,          R1
"op"  R0, R1,      R2
Vstor R2,          @C
```

Pour simplifier, on suppose que la mémoire est composée de 2 sections et chaque section contient 4 bancs mémoires. La mémoire est adressable en *"Lower-Order"*; les deux bits de poids faible constituent l'adresse du banc et le reste représentent l'adresse de la donnée dans le banc. Le processeur possède 3 ports, chaque port est alloué à une instruction d'accès. Un accès mémoire dure 4 cycles et le temps de latence du pipeline de calcul est de 4 cycles. La taille des vecteurs est de 8 éléments. L'exécution de cette expression nécessite l'allocation de 3 registres vectoriels R0, R1 et R2, 3 ports d'accès à la mémoire (deux pour le chargement de R0 et R1 et un troisième pour le rangement du résultat R2) et un pipeline de calcul, comme le montre la figure (2.7).

On suppose que le processeur exécute une instruction par cycle car il n'y a pas de dépendance:

```
Vload @A,      R0
|------Vload @B,      R1
|------|------"op" R0, R1,  R2
|------|------|------Vstor R2,      @C
```

Les vecteurs A, B, et C sont alloués en mémoire comme indiqué dans la table (2.8) : A0 est dans le banc mémoire 7, A1 est dans le banc 4, etc.

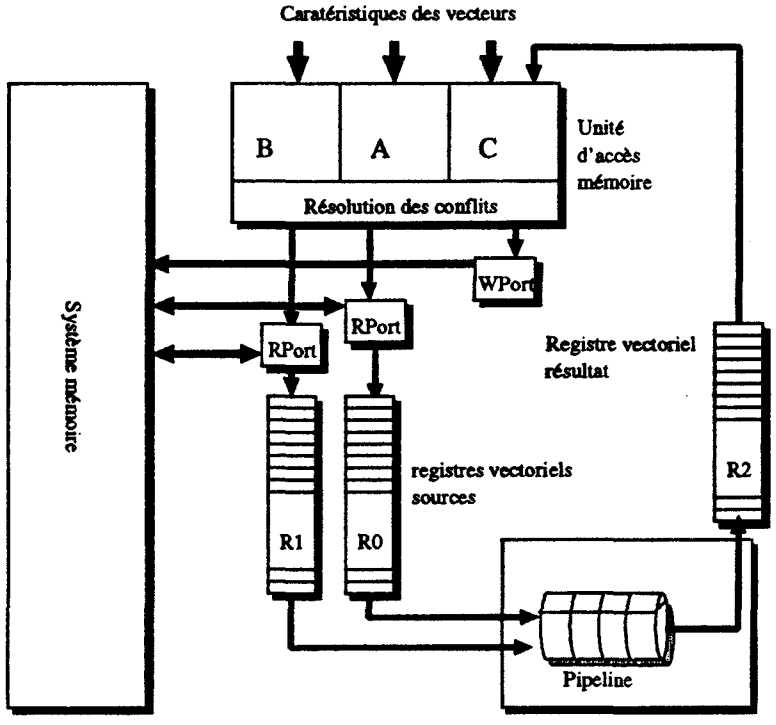


Figure 2.7 : Unités nécessaires pour l'exécution de cet exemple.

La simulation de cet exemple, suivant les paramètres fixés précédemment, est montrée sur la table de réservation de la figure (2.9). La première colonne est le nombre de cycles depuis le déclenchement. La colonne suivante donne l'état de chaque ligne du réseau. La colonne 3 est l'activité des bancs mémoire à chaque instant, et enfin la dernière colonne indique les composantes à la sortie du pipeline. Les ports ont une priorité tournante.

A– Stratégie utilisée

- Les ports sont gérés avec la priorité cyclique d'un cycle de période.

index / vecteurs	0	1	2	3	4	5	6	7
A	7	4	1	6	3	7	1	7
B	7	1	0	5	5	5	4	4
C	5	4	4	2	1	2	3	2

Numéro du banc

Figure 2.8 : table d'allocation des 3 vecteurs A, B et C.

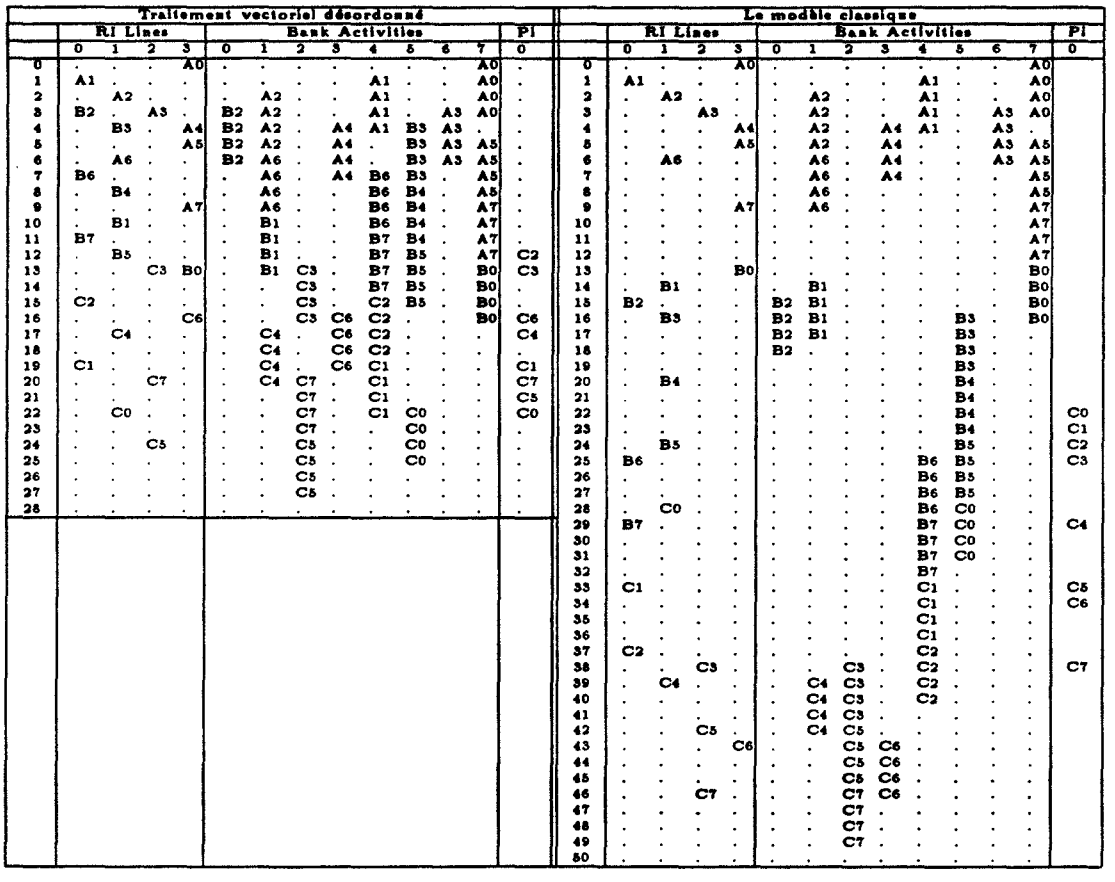


Figure 2.9: Trace d'exécution de l'opération de l'exemple 1.

- On cherche à satisfaire tous les ports actifs, si possible, sinon les ports les plus prioritaires accéderont à la mémoire et les autres seront bloqués.

Cet exemple montre bien comment les conflits de bancs occupés et de sections sont résolus:

Au cycle 0, A_0 est "passé" car le banc mémoire 7 est libre. Au cycle 1, A_1 demande le banc 4 qui est libre et B_0 demande le banc 7 qui est occupé. A_1 accède au banc 4 et B_0 est rangé dans le buffer en attendant que le banc 7 devienne libre. Au cycle 2, on tente A_2 et B_1 qui demandent le même banc. A_2 passe car le port A est plus prioritaire pendant ce cycle. Au cycle 4, le banc 7 devient libre, donc on peut tenter B_0. Le buffer de A n'a pas d'éléments autre que A_4, car le port A n'est pas encore bloqué jusqu'ici. Il ne peut tenter que A_4, alors que le port B a deux possibilités (B_0 et B_3). Or A_4 et B_0 sont en conflit de ligne, d'où le choix de B_3 et A_4 (on optimise les accès).

B– Le gain en performance

La bande passante absolue de m bancs mémoires parallèles de temps de latence t est : $bm_{m,t} = m/t$. Dans ce cas $bm_{8,4} = 2$ requêtes par cycle. Ceci peut se produire si les vecteurs sont indépendants et les accès sont déclenchés en parallèle.

Dans ce cas précis, la bande passante est donnée par le nombre moyen d'accès par cycle. Il est égal à

$$\frac{\text{nombre total d'éléments des trois vecteurs}}{\text{nombre total de cycles nécessaires à l'accès des trois vecteurs}} \qquad (2.1)$$

$b_x(m,t,p,vl) = (24/t_x)$ requêtes/cycle. (p est le temps de latence de pipeline et vl est la taille d'un vecteur).

On note b_{id}, b_{tvd} et b_{cl} respectivement la bande passante théorique (c'est-à-dire l'accès aux $3vl$ données sans conflits), la bande passante du modèle de traitement vectoriel désordonné et la bande passante du modèle classique.

Les temps nécessaires à l'accès des 24 éléments par les modèles désordonné et classique sont obtenus par simulation (voir la table de la figure (2.9)) :

$t_{tvd} = 28cp$ (simulation),
$t_{cl} = 50cp$ (simulation).

Le nombre de cycles nécessaires pour l'accès des trois vecteurs à la mémoire sans conflit est : $t_{id} = 2 + 2t + p + vl$.

La bande passante pour chaque modèle est :
$b_{id}(8,4,4,8) = 24/22 \simeq 1.10$ req/cp.
$b_{tvd}(8,4,4,8) = 24/28 \simeq 0.86$ req/cp.
$b_{cl}(8,4,4,8) = 24/50 = 0.48$ req/cp.

Nous remarquons que la mémoire est beaucoup plus sollicitée en modèle de traitement désordonné. Le gain de performance du modèle de traitement vectoriel par rapport au modèle classique est de 44%. Nous allons montré dans la partie simulation que ce gain augmente en fonction de certains paramètres (vl, m, etc.).

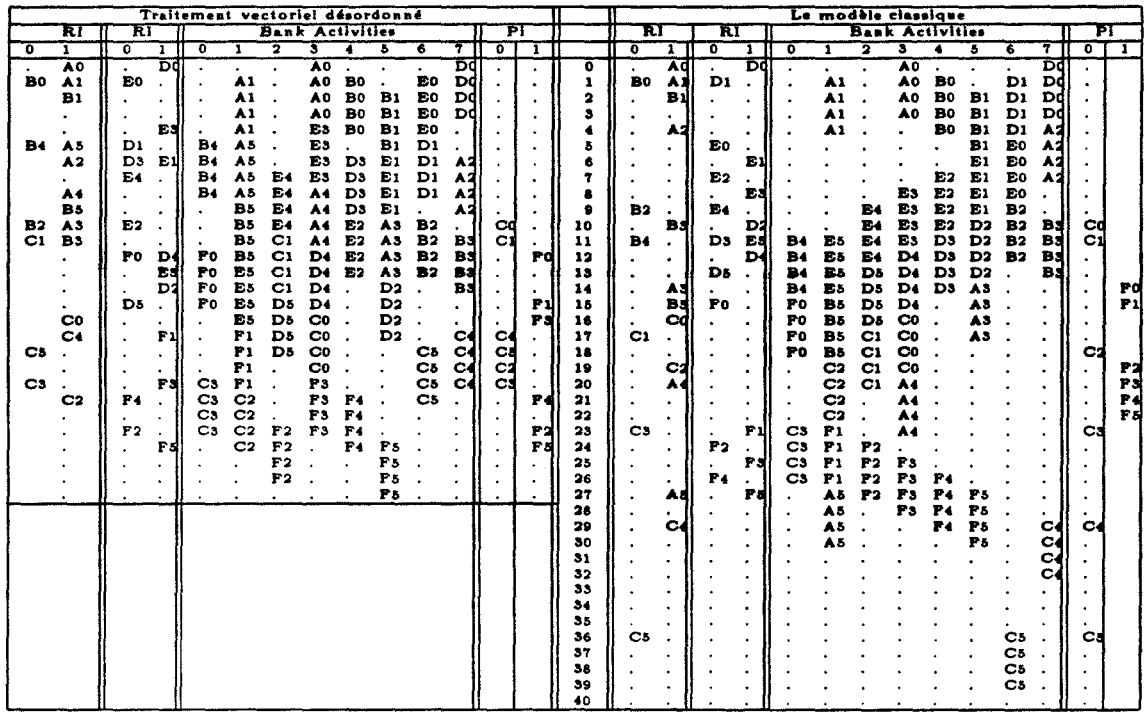


Figure 2.10 : Trace d'exécution de l'opération de l'exemple 2.

2.3.2 Accès par pas réguliers

Le même type d'expression est exécutée sur une configuration à deux processeurs (Cpu1 et Cpu2). Chaque processeur exécute la même expression sur des données différentes. Le Cpu1 exécute l'opération $C = A\ op\ B$ et le Cpu2 exécute l'opération $F = D\ op\ E$. Le système mémoire est composé de 8 bancs répartis en 2 sections de 4 bancs chacune. Les paramètres p et t sont fixés comme dans l'exemple 1. On traite des vecteurs de 6 éléments aux caractéristiques suivantes :

- les adresses de base des 6 vecteurs sont : $(A \rightarrow 3), (B \rightarrow 4), (C \rightarrow 3), (D \rightarrow 7), (E \rightarrow 6), (F \rightarrow 0)$.
- les pas d'accès sont : $(A : 6), (B : 7), (C : 1), (D : 7), (E : 7)$ et $(F : 1)$.

La table de la figure (2.10) nous montre combien les conflits de simultanéité sont pénalisant pour le modèle classique. Tout conflit de ce type est toujours suivi par un conflit de banc occupé, ce qui augmente le temps de réponse de $T + c$ cycles (T est le temps de cycle mémoire et c le temps de réponse du réseau).

Au cycle 8, on a un conflit de simultanéité entre A4 et (D4 ou F3), qu'on ne pouvait pas résoudre, car on n'a pas d'autres choix possibles.

Normalement la méthode séquentielle donne de bons résultats lorsque les données sont uniformément distribuées en mémoire (voir chapitre 1). L'uniformité, dans cet exemple, est justifiée par l'accès de la majorité des vecteurs par pas impairs à l'exception du vecteur A. Nous constatons que le modèle classique, basé essentiellement sur cette méthode, est moins performant

que le modèle désordonné. Ceci est dû aux conflits de simultanéité et au temps de latence de la mémoire. Avec cette méthode tout conflit de simultanéité est suivi directement par un conflit de banc. Par exemple au cycle 5, B_2 et E_0 sont en conflits d'accès au banc 6, ce qui a provoqué le blocage du port B pendant 4 cycles. D'autres conflits de simultanéité se sont produit aux cycles 6, 10, 12 et 23. En modèle de traitement désordonné ces conflits sont détectés et résolus. Par exemple au cycle 6 le conflit de simultanéité entre A_3 et E_1 est résolu en choisissant le triplet (A_2, D_3, E_1). Si nous avions choisi A_3, on était obligé d'élire le couple (A_3, D_3). Le gain en performance par rapport au modèle classique est de 30%.

2.4 Résolution des conflits d'accès

Les architectures multiprocesseurs pipelines vectoriels à mémoire commune souffrent du problème des conflits qui peuvent apparaître à différent niveaux de l'architecture de la machine. D'une manière générale, ces architectures présentent trois types de conflits :

- les conflits de réseau,
- les conflits de simultanéité d'accès au banc mémoire,
- les conflits de bancs occupés.

Certaines techniques de résolution de conflits sont exposées dans le chapitre 1. Elles sont plus ou moins efficaces suivant le type d'application et de l'architecture de la machine. En traitement vectoriel désordonné la résolution des conflits se fait comme suit :

- Eviter le blocage au niveau des bancs et des lignes du réseau d'interconnexions par des schémas de priorités. Ceci permet de choisir parmi plusieurs demandes d'accès à une ressource (lignes du réseau, bancs mémoires), une demande accessible en premier.
- Eviter au maximum qu'un port (et/ou processeur) soit bloqué. Suivant les demandes des ports, on tente de répartir les lignes et les bancs mémoires de façon à ce que tous les ports effectuent des accès (plus ou moins) sans conflit. Ceci permet de minimiser les conflits, donc les temps d'attentes, et d'éviter les dégradations de performances.

Ces deux procédures interviennent pour résoudre les conflits de réseau, les conflits de simultanéité et les conflits de bancs occupés.

2.4.1 Conflits de réseau

Ils apparaissent lorsque le réseau d'interconnexions n'est pas un crossbar complet. A cause du coût très élevé du crossbar, les systèmes multiprocesseurs ont tendance à utiliser des réseaux mixtes (multi-étages). Ces réseaux ne sont pas très coûteux, mais ils engendrent des conflits d'accès. Dans la machine Cray X-MP, par exemple, le réseau est un 2-étages : un crossbar reliant les ports aux sections mémoires et le réseau multi-bus reliant les crossbars à tous les bancs d'une même section. Les conflits générés sont appelés *conflits de sections*. Ils apparaissent lorsque deux ports d'un même processeur tentent l'accès à des bancs mémoires qui se trouvent dans la même section [Cheung et al.86].

Les délais causés par ces conflits dépendent de l'architecture du réseau. (Sur le X-MP ils durent un cycle, sur le Y-MP, de 1 à 2 cycles).

2.4.2 Conflits de simultanéité

Ils apparaissent lorsque deux ou plusieurs processeurs tentent l'accès à un même banc. Lorsqu'un tel conflit se produit, les règles de priorités sont mises en place pour enlever la situation de blocage. En traitement vectoriel désordonné on tentera d'anticiper en envoyant une autre requête à la place de celle qui cause ce conflit, sinon on appliquera la règle de priorité.

2.4.3 Conflits de bancs occupés

Les mémoires ont souvent un temps de réponse (temps d'accès) très long par rapport à celui des processeurs. Ces conflits se produisent lorsqu'un ou plusieurs processeurs référencent un banc mémoire en cours d'activité. En modèle classique, lorsqu'un conflit de simultanéité se produit il est toujours suivi par un conflit de banc occupé.

2.5 Stratégies de sélection

Chaque flux d'accès est associé à un port mémoire. Toutes les requêtes de ce flux sont rangées dans le buffer du port correspondant en attendant l'autorisation d'accès. Le problème est de trouver une "**méthode**" ou une **classe de méthodes** qui permette de choisir les requêtes accessibles à chaque cycle, en tenant compte des trois types de conflits cités précédemment. En d'autres termes, il faut sélectionner des requêtes qui ne causent pas de conflits. Ces méthodes sont appelées **stratégies de sélection**.

De ce point de vue, on peut considérer que le modèle classique est un cas particulier de notre modèle où la stratégie de sélection est : *"tenter la requête en tête de file à chaque cycle"*. C'est une mauvaise stratégie, car elle n'évite pas les conflits.

Nous dirons qu'une stratégie est une **bonne stratégie** si elle vérifie les critères suivants :

1. Efficacité : une bande passante élevée et un bon rapport de performances par rapport à la stratégie classique.
2. Temps de réponse court : elle s'exécute en un temps court et constant.
3. Implémentation facile : elle ne demande pas beaucoup de matériel et supporte éventuellement un fonctionnement pipeline.
4. Elle est globale : elle ne dépend pas d'un type d'accès ou d'une application particulière.
5. Elimination des situations d'inter-blocages : il faut pouvoir décider de la requête à satisfaire lorsque plusieurs se présentent simultanément.

La stratégie classique répond bien aux critères 2, 3 et 5, mais elle ne répond pas aux critères de performances 4 et 1.

Evidemment on peut imaginer une infinité de stratégies, mais sont-elles de bonnes stratégies? Dans notre contexte, la classe de stratégies qui nous intéresse doit résoudre les conflits de bancs occupés, les conflits de sections et les conflits de simultanéité.

Conflits de bancs occupés : Ce sont les plus faciles à résoudre. A chaque cycle, un processeur doit connaître l'état de tous les bancs mémoires, il peut éliminer les requêtes bloquées. Leur résolution est locale à chaque processeur.

Conflits de sections: Le réseau d'interconnexions entre chaque processeur et les sections mémoires est un crossbar ($P \times S$), (P est le nombre de ports par processeur et S est le nombre de sections mémoires). Leur résolution consiste à répartir les ports qui véhiculent un flux d'accès sur les S sections afin d'éviter que deux ports adressent une même section. Ceci peut se faire localement à chaque processeur.

Conflits de simultanéité: Ce sont les plus difficiles à résoudre, ils nécessitent la coopération inter-processeur afin de décider des requêtes accessibles pour chacun des processeurs. Remarquons que ces conflits ne concernent que les ports appartenant à des processeurs distincts. Pour les résoudre nous avons étudié deux techniques possibles que nous appelons les *techniques centralisées* et les *techniques décentralisées.*

La classe de stratégies que nous recherchons se compose:

- d'une méthode de résolution des conflits de sections.
- d'une méthode de résolution des conflits de simultanéité.
- de la méthode de résolution des conflits de bancs occupés, elle est unique à toutes les stratégies.
- d'un système de priorités entre les ports et les processeurs pour éliminer l'inter-blocage.

La résolution des conflits doit pouvoir fonctionner en mode pipeline, car suivant le temps de cycle du processeur, il n'est pas garanti que l'élimination des conflits de sections, de simultanéité et de bancs occupés puisse se faire en un seul cycle. Le but ici est de définir les phases de ce pipeline.

La résolution des conflits de bancs occupés se fait par l'intermédiaire de l'état des bancs mémoires. Le vecteur d'état de la mémoire est reçu à chaque cycle, toute requête, qui se trouve dans le buffer, référençant un banc occupé est ainsi écartée. Seules les requêtes qui référencent les bancs libres sont sélectionnées. Ceci définit la phase de résolution des conflits de bancs occupés. Si cette phase se fait au début du pipeline, les requêtes sélectionnées peuvent être en conflit de bancs lors de l'accès effectif suite au changement d'état de la mémoire (pendant la traversée du pipeline). Par conséquent la phase de résolution des conflits de bancs occupés se fait en dernier étage du pipeline.

D'une manière générale, cette classe de stratégies opère en deux phases:

1. **Phase I:** Elle consiste à résoudre les conflits de sections. Les requêtes sont réparties sur les registres sections. Un registre section doit contenir au plus une requête par banc pour tous les bancs de la même section. Il est inutile de sélectionner deux requêtes qui référencent le même banc en sachant qu'une seule requête pourra accéder. La résolution des conflits de section consiste à affecter une section (valider un registre section) par port. Pour les phases qui suivent chaque port se présente avec son registre section. L'affectation des sections sur les ports tient compte du nombre de requêtes contenues dans chaque registre section et de la priorité du port.
2. **Phase II:** Durant cette phase,
   - Par une technique de résolution des conflits de simultanéité, on identifie pour chaque port, les bancs mémoires sur lesquels le système garantit l'accès mémoire.

- on procède à l'élimination des conflits de bancs occupés pour chaque port. Cette opération n'est pas coûteuse. On effectue un ET logique du registre section avec le vecteur d'état de la mémoire.
- Finalement parmi les requêtes qui référencent les bancs libres on doit élire, si possible, une requête qui ne présentent pas des conflits de simultanéité. On tient compte de la priorité d'accès des processeurs à la mémoire.

La figure (2.11) présente les différentes phases de traitement que doit subir chaque requête pour accéder à la mémoire. Cette classe peut fonctionner en mode pipeline. La phase initiale correspond au calcul d'adresses et bufferisation. La phase I répartit les sections mémoires sur les différents ports. La dernière phase consiste à écarter les requêtes en conflits de bancs occupés et éventuellement les requêtes qui causent des conflits de simultanéité.

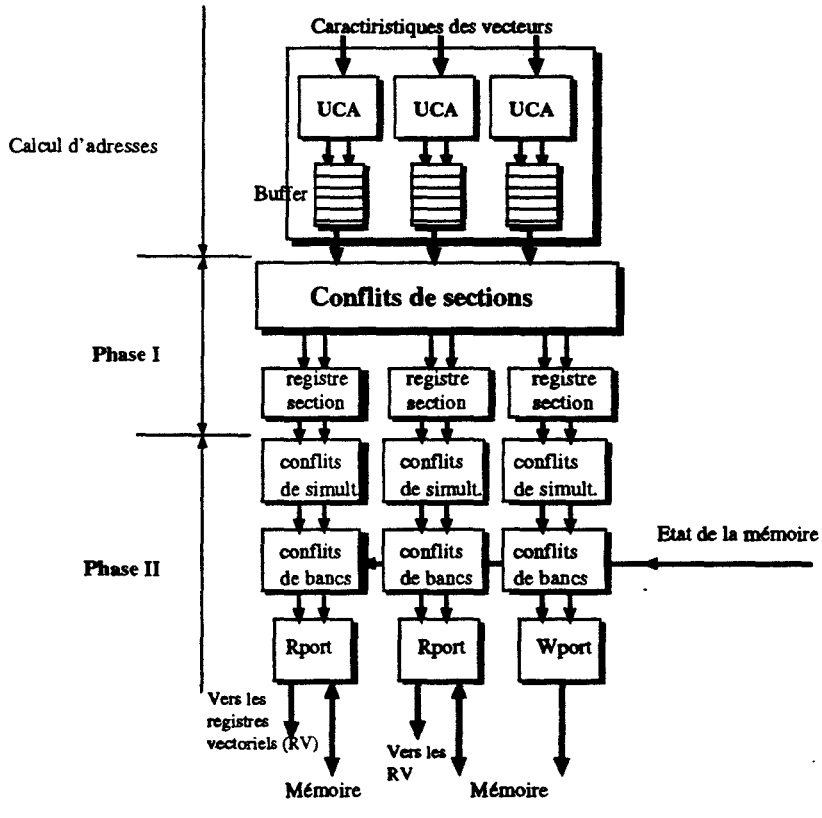


Figure 2.11 : Synoptique des deux phases de stratégie de résolution

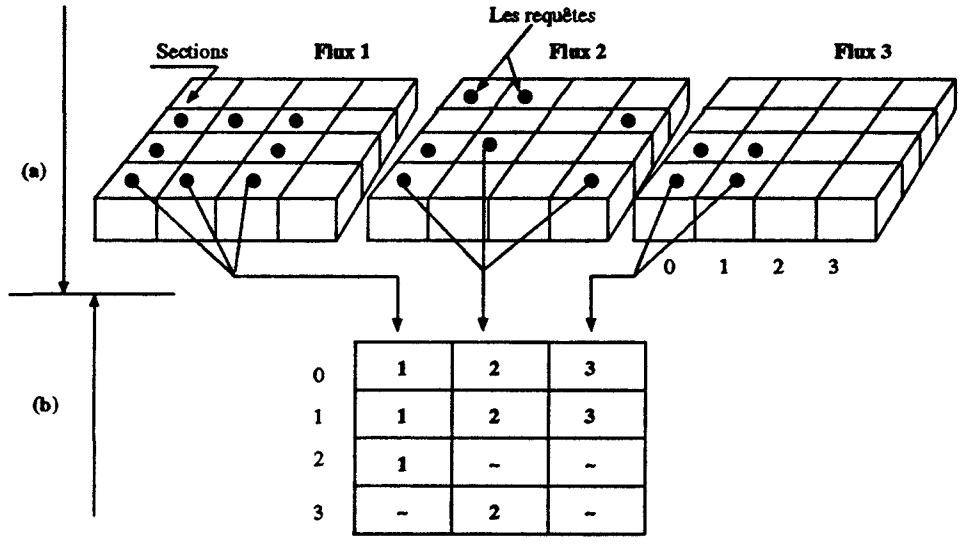
2.5.1 Sélection des requêtes, conflits de sections

Pendant cette phase, tous les buffers sont explorés en parallèle pour définir quels sont les bancs référencés par chaque flux d'accès. Ceci est réalisé en associant à chaque flux une matrice de bits, dont les lignes représentent les différents sections et les colonnes représentent les

bancs appartenant à chaque section. Chaque bit indique, à chaque cycle, les bancs mémoires référencés par les requêtes du buffer (voir figure (2.12). Ceci est fait pour tous les flux actifs. Le but ici est de sélectionner une section non vide par port (c'est-à-dire il y a au moins une requête qui adresse un banc appartenant à cette section).

Pour cela on procède comme suit :

1. On construit les différentes lignes de la matrice de bits. Ceci nous permet d'identifier les différents choix pour chaque port (figure (2.12)).
2. On optimise la répartition des registres sections , en tentant d'affecter à chaque port un registre section qui contient des requêtes. Dans l'exemple de la figure (2.12), les sections suivantes sont attribuées par un mécanisme de priorité. Les sections 2 et 3 ne sont demandés que par les port 1 et 2 respectivement, le port 3 a le choix entre la section 0 et 1. Comme on peut le remarquer sur cette exemple la solution optimale n'est pas unique.



(a) : Requêtes classées en fonction de l'adresse de leur section.
(b) : Différents choix de chaque port .

Figure 2.12 : Le pipeline de résolution des conflits de sections.

L'algorithme d'allocation des requêtes de la figure (2.13) identifie dans une première phase les sections référencées par un seul port. Les sections suivantes sont attribuées par un mécanisme de priorité.

2.5.2 Stratégies pour les conflits de simultanéité

Le traitement des conflits de simultanéité peut être effectué avant ou après l'élimination des conflits de bancs occupés à une condition : les deux types de conflits doivent être résolus en un seul cycle. Sinon on doit résoudre les conflits de simultanéité et en dernière phase on élimine les requêtes en conflits de bancs occupés. La question qu'on peut se poser est comment prendre

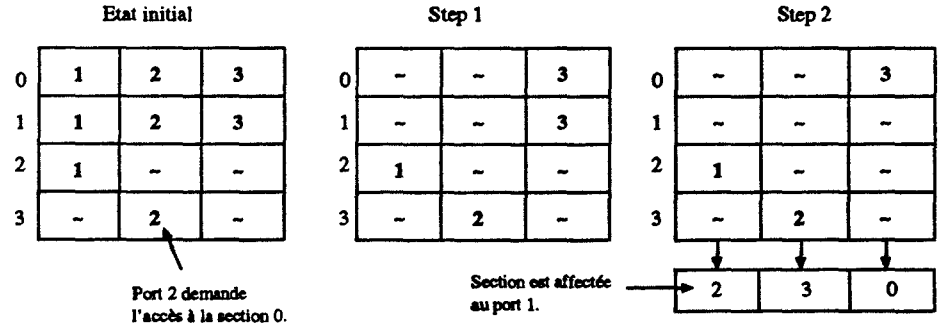


Figure 2.13: Technique d'allocation des sections sur les ports.

en compte les conflits de simultanéité et quelles sont les différentes techniques permettant de les détecter?

La méthode consiste à marquer (positionner le bit correspondant) les bancs qui sont en conflits de simultanéité (figure 2.14). Le marquage des bancs dépend de la technique de détection utilisée. Si la technique utilisée permet de déceler tous les conflits, on procède au marquage des bancs qui peuvent bloquer le processeur. Dans le cas contraire on ne marquera que les bancs dont on est sûr de l'accès. Le résultat de ce marquage peut être considéré comme un vecteur masque indiquant les bancs qui sont en conflits de simultanéité ou ceux qui ne le sont pas.

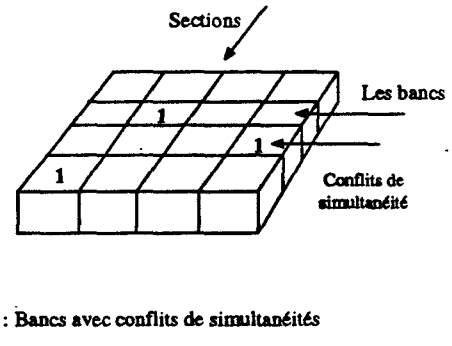


Figure 2.14: Identification des bancs en conflits de simultanéité.

Les bancs marqués sont pris en compte pendant la résolution des conflits de simultanéité, on effectue un ET logique du vecteur masque des bancs en conflits de simultanéité avec le registre section alloué au port.

D'une manière générale, la technique de marquage peut être centralisée; elle nécessite des communications inter-processeurs ou bien décentralisée (il n'y a pas de communications inter-processeurs).

2.5.2.1 Stratégies centralisées

Soit N le nombre de processeurs dans le système. Après avoir exploré tous les buffers et résolu les conflits de sections, chaque processeur connaît tous les bancs qu'il peut référencer. L'ensemble de ces bancs référencés par le processeur p est nommé $E(p)$. Un mécanisme centralisé consiste à consulter toutes les $E(p)$; $(p = 1, \cdots, N)$. Le but de cette stratégie est de marquer, pour chaque $E(p)$, les bancs en conflits de simultanéité en tenant compte du système de priorité des processeurs. Par exemple si un banc mémoire est référencé par plus d'un processeur, il sera attribué au processeur le plus prioritaire, et pour les autres il est considéré comme conflit de simultanéité, donc, il sera marqué. Nous allons voir les différentes règles de priorité plus loin.

Cette stratégie peut être vu comme une boîte aux lettres où les processeurs déposent leurs vecteurs de bancs référencés $E(p)$, (voir la figure (2.15)), puis chacun récupère son $E(p)$ à la fin du traitement de la stratégie, celle-ci ayant marqué les bancs qui peuvent être sujet à un conflit de simultanéité.

Les stratégies centralisées se résument en deux points suivants :

1. Identifier les bancs demandés par plus d'un processeur,

2. Marquer les bancs uniquement pour les processeurs qui ne sont pas les plus prioritaires pour ces bancs.

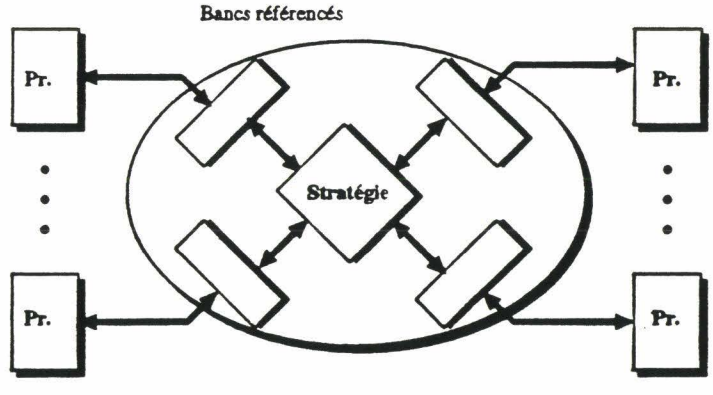


Figure 2.15 : Marquage des conflits de simultanéité par un mécanisme centralisé.

Bien que cette technique détecte toutes les requêtes en conflits de simultanéité, elle présente les inconvénients suivants :

- Elle est coûteuse en temps à cause des communications.

- Elle dépend du nombre de processeurs et du nombre de bancs mémoires: plus le nombre de processeurs et de bancs mémoires sont élevés plus le temps de réponse de l'algorithme de marquage des bancs en conflits de simultanéité est important.

2.5.2.2 Stratégies distribuées

Le but de ces techniques est de supprimer toute communication inter-processeur coûteuses en temps. Chaque processeur procède au marquage de manière locale.

Evidemment on ne peut pas marquer tous les bancs qui peuvent présenter des conflits de simultanéité car chaque processeur ignore les bancs référencés par les autres processeurs. Cependant chaque processeur peut connaître les bancs pour lesquels il est sûr d'accéder en priorité et ceci grâce au système de priorité mis en place. Ainsi le marquage concerne les bancs "les plus prioritaires".

Lors de la résolution des conflits de simultanéité, on tentera, si possible, de sélectionner les requêtes qui référencent les bancs marqués puisqu'elles passeront en priorité, sinon on tentera d'autres requêtes, il n'est pas certain que les autres processeurs déclenchent des accès à ces bancs.

Les règles de priorités à définir concernent les entités suivantes: les processeurs, les ports, et les types d'accès. Elles ont toutes un point commun qui est la priorité d'accès aux bancs mémoires.

2.5.2.3 Priorité fixe

C'est la plus simple à implémenter. Elle peut gérer les processeurs, les ports d'un même processeur et les types d'accès. Elle ne dépend d'aucun paramètre (temps, bancs mémoires, ...). Elle est définie par la fonction $P(n) = n$, $(n = 1, \cdots, N)$. n étant l'entité à gérer par la priorité P, qui peut être le processeur, le port ou le type d'accès, et N est le nombre total d'entités. Elle est fixée, par exemple, suivant les différents types d'accès : un accès contigu est plus prioritaire que les gather/scatter, les accès par pas impair sont plus prioritaires que les accès par pas pair, etc.

2.5.2.4 Priorité cyclique

Il est clair qu'avec la priorité précédente la gestion des conflits de simultanéité favorise le processeur le plus prioritaire (il termine souvent le premier) et pénalise les autres processeurs. La priorité cyclique permet de remédier en partie à ce problème. Elle est définie par : $P(i,t) = (i + \lfloor t/c \rfloor) \bmod N$. i est une entité, processeur et/ou ports. Les deux paramètres t et c sont respectivement le temps et la période de basculement de priorité.

2.5.2.5 Priorité de scrutation

Les deux premières règles ne suffisent pas pour définir une technique qui éliminerait les conflits de simultanéité. Cette technique gère les processeurs et les ports par rapport à la totalité de l'espace mémoire: le processeur qui a la priorité la plus élevée est prioritaire à tous les bancs mémoires. La priorité de scrutation permet de lier les processeurs aux bancs mémoires de la façon suivante : $P(i,t,j) = P(i,t+1,j)+1$. A chaque instant (t) le processeur (i) a la priorité $P(i,t,j)$ pour accéder au banc (j). Cette priorité est tournante pour chaque processeur et pour chaque banc mémoire.

Soit $P(*,t,j)$ le vecteur des priorités de tous les processeurs au banc mémoire (j). A l'initialisation, on attribue à tout processeur une priorité p, (compris entre 1 et N) d'accès au banc (j) différente des priorités des autres processeurs. Ainsi $P(*,0,j) = (p_{1,j}, p_{2,j}, \cdots, p_{N,j})$, avec

$p_{i,j} \neq p_{k,j}$ pour $i \neq k$. Ceci est répété pour tous les bancs mémoires. $P(*,t,*)$ est une matrice $(M \times N)$, M étant le nombre de bancs mémoires.

On peut remarquer facilement que pour les deux règles précédentes le marquage est faible, il ne concerne que le processeur le plus prioritaire, car c'est le seul qui peut accéder sûrement. Avec la priorité de scrutation le marquage est beaucoup plus riche, et la résolution des conflits de simultanéité est meilleure.

2.5.2.6 Priorité de scrutation par zone

La priorité de scrutation telle qu'elle est présentée pose un problème d'initialisation de la matrice $P(*,0,*)$. Si elle est initialisée de manière aléatoire, elle ne favorise aucun type d'accès. Donnons un exemple.

Soit $M = 8$ et $P(i,0,*) = (7,3,6,0,7,6,5,3)$ ce qui signifie que le processeur (i) est plus prioritaire aux bancs $0,4$. On veut accéder à un vecteur contigu qui a pour adresse de base 3. Si le premier élément cause un conflit il est bloqué car il a une priorité minimum, Au cycle suivant, on a $P(i,1,*) = (0,4,5,1,0,7,6,4)$, le second élément d'adresse 4 a la priorité la plus basse, etc. Il faut attendre 8 cycles pour qu'une requête devienne plus prioritaire.

Nous proposons alors d'initialiser la matrice en partitionnant la mémoire sur les processeurs. Chaque processeur est plus prioritaire dès le départ à une zone mémoire. Cette zone contient un nombre de bancs constant. Elle est définie comme suit : les bancs de chaque section sont répartis en groupes de Z bancs (avec $Z = B \bmod N$) où B est le nombre de bancs mémoire par section et N le nombre de processeurs. Chaque groupe de bancs est affecté à un processeur suivant les adresses de base des vecteurs à accéder. Le changement de zone s'effectue de manière cyclique. Le pas de déplacement des zones est le pas de la majorité des vecteurs à accéder.

2.6 Conclusion

Dans ce chapitre nous avons présenté un nouveau modèle de traitement vectoriel désordonné qui répond au problème de perte de performances des systèmes multiprocesseurs pipelines vectoriels. Nous avons aussi présenté ses deux fonctionnalités essentielles et indivisibles, qui sont la méthode d'accès et le modèle d'exécution dans les unités fonctionnelles. Nous avons identifié deux types de flux : le flux d'accès et le flux de calcul. Le flux d'accès produit le gain en performances par rapport au modèle classique et le flux de calcul conserve ce gain par chaînage avec le flux d'accès.

Ce modèle a notamment résolu le problème de distribution des files d'attentes dans un système multiprocesseur et d'autres problèmes liés à cette technique qui sont :

- La rupture du chaînage des pipelines,
- Les opération de réordonnancement des composantes à la sortie de la mémoire.

Nous avons vu que toutes les méthodes présentées dans le chapitre précédent ne sont pas efficaces lors de l'exécution des opérations gather/scatter ou compress/extend. A l'opposé, le modèle de traitement désordonné ne dépend pas d'un type d'accès particulier et il est bien adapté pour l'exécution de ce type d'opérations.

Nous avons défini la classe de stratégies qui résout les différents types de conflits d'accès mémoires. Cette classe est caractérisée par un fonctionnement en mode pipeline, donc, facile à implémenter.

L'implémentation de notre modèle TVD demande quelques modifications au niveau des unités fonctionnelles d'un processeur vectoriel classique. Ceci fait l'objet du chapitre suivant.

Chapitre 3

Implémentation du Modèle

3.1 Introduction

Après avoir expliqué le fonctionnement du modèle, nous allons nous intéresser à son implémentation et plus précisément aux modifications à apporter aux calculateurs vectoriels multiprocesseurs classiques pour supporter le modèle.

Nos premiers travaux ont été consacré à l'implémentation du traitement vectoriel désordonné sur une machine vectorielle monoprocesseur. Son implémentation était simplifiée par l'absence de conflits de simultanéité [Dekeyser90], [Preux91] et [Dekeyser et al.90].

Dans un premier temps, nous allons préciser l'environnement architectural nécessaire à l'implémentation du modèle TVD. Après avoir défini l'architecture globale du calculateur; c'est à dire les unités essentielles que doit comporter un processeur pour supporter le modèle de traitement désordonné, nous spécifions toutes ses unités ainsi que leurs fonctionnements.

3.2 Architecture globale du calculateur

Le modèle de traitement vectoriel désordonné est destiné aux calculateurs multiprocesseurs vectoriels pipelines à mémoire partagée. L'originalité du modèle ne vient pas de l'architecture vectorielle de la machine, mais du fonctionnement des unités vectorielles au sein de chaque processeur. Ce système mulitprocesseur est caractérisé par l'architecture des processeurs, l'architecture du réseau d'interconnexions et son organisation mémoire.

3.2.1 Les processeurs

Tous les processeurs sont identiques; c'est une architecture homogène qui est à la fois SIMD et MIMD. Les communications inter-processeurs se font par l'intermédiaire de la mémoire. L'ensemble des processeurs partagent un ensemble de registres dédiés à la synchronisation.

Chaque processeur renferme plusieurs unités fonctionnelles. Un tel processeur est montré en figure (3.1). Les unités essentielles sont :

- Une unité vectorielle composée de plusieurs unités fonctionnelles de calcul pouvant travailler en concurrence.
- Une unité scalaire et une unité de contrôle et de commande. Cette dernière n'est pas concernée spécialement par le traitement vectoriel désordonné, mis à part quelques modifications qui sont à apporter pour gérer des nouveaux signaux conséquents de l'implémentation de nouvelles unités. Ces deux unités ne sont pas développés ici.
- Une unité d'adressage. Nous considérons cette unité complètement indépendante de l'unité scalaire et des unités fonctionnelles de calcul appartenant à l'unité vectorielle. Elle ne s'occupe que du calcul d'adresses et des accès mémoires. Elle implémente la méthode d'accès désordonnés.
- Les ports mémoires. Chaque processeur possède 3 ports d'accès mémoires qui peuvent supporter deux lectures et une écriture simultanément. La majorité des calculateurs vectoriels fonctionnent ainsi.
- Les registres vectoriels et scalaires. C'est un calculateur registre à registre.

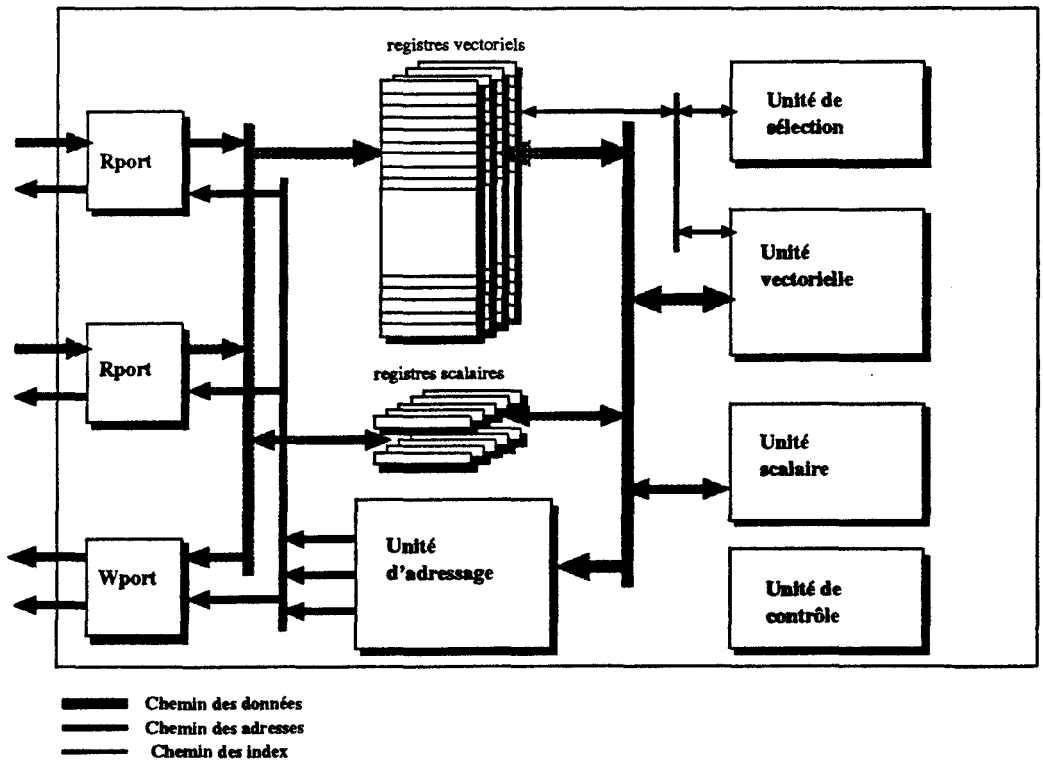


Figure 3.1 : Architecture d'un processeur

3.2.2 Le réseau utilisé

Le réseau d'interconnexions relie les processeurs aux mémoires parallèles de telle façon que tout processeur peut adresser la totalité de l'espace mémoire directement (via le réseau). Ce réseau est un 2-étages : le premier niveau est un ensemble de crossbars ($3 \times S$). Chaque crossbar relie les différents ports d'un processeur aux S super-bancs. Le deuxième niveau est un réseau multi-bus reliant les processeurs aux bancs mémoires. Chaque banc mémoire possède un lien physique avec chacun des crossbars; il y a autant de ports pour un banc mémoire qu'il y a de processeurs.

3.2.3 L'organisation mémoire

Nous avons choisi le système mémoire des calculateurs multiprocesseurs usuels, c'est-à-dire un ensemble de modules mémoires indépendants entrelacés, de telle sorte qu'on puisse effectuer plusieurs accès en parallèle. Ces bancs mémoires sont de même taille et de même temps de cycle. Les mots mémoires sont organisés suivant la méthode séquentielle (*Lower-Order*). En d'autres termes, deux adresses consécutives sont dans deux bancs successifs.

A cause du coût du réseau d'interconnexions, la mémoire est à deux niveaux d'entrelacement. La mémoire est découpée en super-bancs et chaque super-banc est composé de plusieurs bancs mémoires. Les super-bancs sont entrelacés, et au sein d'un super-banc les bancs sont aussi entrelacés. Cette organisation n'est pas propre à la machine TVD, elle est utilisée dans les super-calculateurs existants.

Les bancs mémoires sont multiports. Ces ports ne peuvent pas être actifs en parallèle, on n'autorise pas la lecture simultanée de deux mots dans un même banc. On peut recevoir plusieurs requêtes, une seule requête est servie à la fois. Les ports des processeurs correspondant aux autres requêtes recevront un signal d'échec indiquant qu'ils sont en conflits de simultanéité.

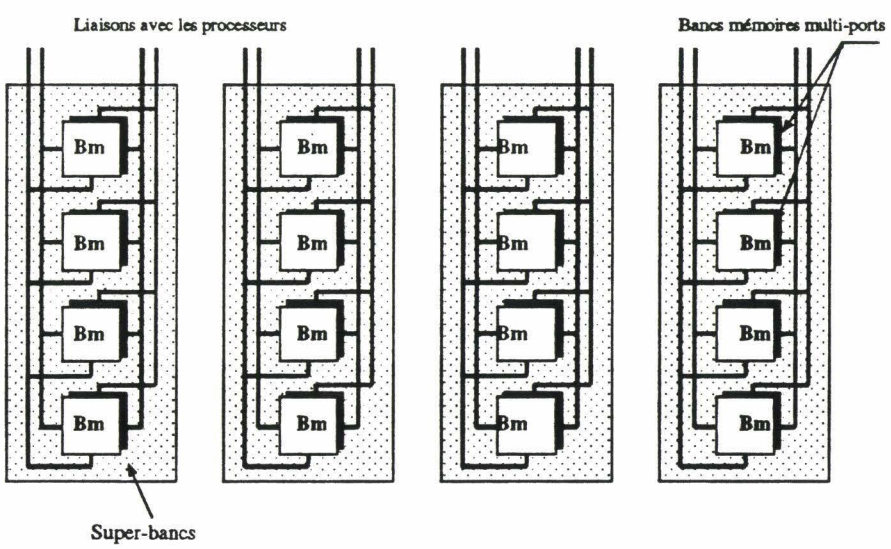


Figure 3.2 : Synoptique de l'organisation mémoire (4 CPUs)

3.2.4 Conséquences du modèle sur l'architecture de la machine

L'une des propriétés originales du modèle de traitement vectoriel désordonné est qu'il est présent dans tout le processeur. Son implémentation implique des modifications de toutes les unités qui doivent piloter le traitement vectoriel. Ces modifications concernent toutes les unités qui véhiculent les flux d'accès et les flux de calcul.

Le flux d'accès est véhiculé par les unités suivantes :

- L'unité d'adressage,
- Les ports,
- Les bancs mémoires,
- Le réseau d'interconnexions.

Le traitement, dans chaque unité à tout instant, est indépendant de l'état des autres unités. Toutes ces unités sont chaînées en formant le pipeline du flux d'accès mémoire. L'unité destination du pipeline est soit la mémoire en cas d'écriture, soit les registres vectoriels en cas de lecture.

Le flux de calcul concernent :

- les registres vectoriels,
- les unités fonctionnelles.

Le pipeline (registres vectoriels)-(unités fonctionnelles) est circulaire. La source et la destination sont les registres vectoriels. Dans ce qui suit nous développons chacune de ces unités.

Avant d'entamer la description de toutes les unités précitées, nous allons montrer comment il est possible d'implémenter le modèle de traitement désordonné. Son implémentation est effectuée à l'aide des registres masques et quelques logiques supplémentaires qui permettent la gestion de ces masques, le contrôle, le traitement et le chaînage des différentes unités. L'ensemble des masques et de la logique de gestion et de contrôle est regroupé sous une même unité appelée : *unité de sélection*.

3.3 L'unité de sélection

Le processeur à traitement vectoriel désordonné (PVD) implémente une nouvelle unité essentielle qui gère l'ensemble du traitement au niveau de toutes les unités fonctionnelles. Elle est appelée "**l'unité de sélection**".

Nous avons montré, dans le chapitre précédent, le fontionnement du TVD sur les deux types de flux. Ce fonctionnement est caractérisé par :

1. La production des couples (adresse, index) ou (adresse, donnée) en flux d'accès en lecture ou en écriture respectivement.
2. La propagation du couple (donnée, index) à travers toutes les unités fonctionnelles,

3. **La disponibilité et la conformité** des composantes. Si les deux conditions ne sont pas vérifiées, le déclenchement du traitement des composantes est impossible.

L'implémentation du modèle se ramène, en quelque sorte, à l'implémentation de ces trois caractéristiques. La première caractéristique concerne l'unité d'adressage, la deuxième concerne les unités fonctionnelles de calcul, et la troisième est au niveau de la sélection des opérandes sources contenus dans les registres vectoriels.

L'implémentation de l'unité d'adressage et des unités fonctionnelles de calcul, qui concernent les deux premières caractéristiques, seront traitées plus loin. Dans ce qui suit nous nous intéresserons à la troisième caractéristique.

3.3.1 Propriétés de disponibilité et de conformité

La propriété de disponibilité est facilement implémentée par un registre masque. On associe à chaque composante un bit masque indiquant si elle est disponible ou non, ce qui revient à attribuer un registre masque pour chaque registre vectoriel. Nous dirons qu'une composante est disponible dans un registre vectoriel si le bit masque correspondant est positionné à 1, il est positionné à 0 sinon.

Le problème de conformité ne se pose pas pour les opérations à un seul opérande. Cette propriété est vérifiée uniquement pour les opérations à deux opérandes. Ceci revient à comparer deux à deux les bits masques de même index des deux registres vectoriels opérandes. Ce traitement est pris en charge par une unité spéciale, appelée *unité de sélection des composantes*.

Ceci n'est évidemment pas suffisant. Non seulement il faut détecter la fin de l'opération, mais il faut aussi identifier les composantes déjà traitées pour éviter de déclencher une seconde fois l'opération sur les mêmes composantes.

A première vue, on peut dire qu'il suffit de tester le masque du registre vectoriel cible, or le résultat n'est pas disponible immédiatement, car les unités fonctionnelles sont pipelinées. On est amené à associer un registre masque à toute unité fonctionnelle de calcul y compris les unités de calcul d'adresses pour identifier les composantes déjà déclenchées.

3.3.2 Gestion des registres masques

On a deux types de registres masques:

- les masques des registres vectoriels,
- les masques des unités de traitements (les unités fonctionnelles de calcul et les unités de calcul d'adresses). Les masques des deux unités de calcul d'adresses en lecture (UCALs) gèrent les opérations gather, et celui de l'unité de calcul d'adresses en écriture (UCAE) gère les écritures et les opérations scatter.

La gestion de ces registres masques se réduit au trois opérations suivantes:

- Initialisation des registres masques.
- Positionnement des registres masques.
- Détection de terminaison d'une opération de calcul.

3.3.2.1 Initialisation des masques

Lorsqu'une instruction vectorielle est déclenchée, les ressources sont allouées: les registres vectoriels sources et le registre vectoriel résultat, l'unité fonctionnelle qui exécute l'opération, les unités de calcul d'adresses (UCA) pour les opérations d'E/S, ainsi que les registres masques associés dans l'unité de sélection.

Les vecteurs traités ne sont pas toujours de même taille que celle des registres vectoriels. Lorsque leur taille est supérieure, on applique le principe du *"strip-mining"*. Dans ce qui suit nous considérons que la taille des vecteurs traités est toujours inférieure ou égale à la taille des registres vectoriels.

Nous considérons aussi que les processeurs disposent d'un registre qui contient la taille des vecteurs traités. Ce registre est appelé `VL` sur les machines Cray, Fujitsu et Nec et `VCT` sur IBM.

L'initialisation des registres masques concerne uniquement les registres vectoriels cibles et les unités de traitement allouées. Elle se fait en fonction de `VL`. On peut distinguer deux cas: le cas où `VL` est égal à la taille des registres vectoriels (`TR`), et le cas où `VL<TR`.

Cas VL = TR:

Au déclenchement de l'instruction le registre masque cible et le masque de l'unité de traitement seront initialisés à 0. Aucune composante n'est ni disponible dans le registre destination ni en cours de traitement dans l'unité de traitement.

Cas VL < TR:

Comme le traitement s'effectuera sur `VL` composantes, les `TR-VL` composantes restantes seront considérées comme étant déjà traitées. Par conséquent les `VL` éléments du registre masque de l'unité de traitement seront initialisés à 0 (ils ne sont pas encore traités) et `TR-VL` éléments restants seront initialisés à 1 (ils sont considérés comme étant déjà traités).

Pour le registre vectoriel cible on applique la même règle en considérant que les `TR-VL` dernières composantes sont disponibles mais indéfinies. Cette technique évite les dead-lock sur l'attente de composante. Supposons que le programmeur ait effectué un traitement d'un vecteur sur une taille L, et que dans l'instruction suivante, le même vecteur soit traité sur une taille supérieure à L; $L+2$ par exemple. Grâce à ce mécanisme le processeur n'attend pas indéfiniment la disponibilité des 2 composantes manquantes, le calcul est toujours effectué mais sur des données erronées.

3.3.2.2 Positionnement des registres masques

Les masques des unités de traitement doivent être mis à jour pour identifier les composantes déjà traitées. A la sortie des unités de traitement, les résultats seront rangés dans les registres vectoriels cibles et leurs masques seront aussi mis à jour pour identifier les composantes disponibles.

A chaque déclenchement des opérandes, les bits masques associés dans l'unité de traitement sont positionnés à 1. L'adresse des bits à positionner est donnée par l'index produit au moment du déclenchement. Cette index est propagé dans les unités fonctionnelles pour identifier l'emplacement de la donnée résultat dans le registre cible et l'adresse du bit masque à mettre à jour.

Notons que lorsque plusieurs couples de composantes sont disponibles et conformes, la priorité

est donnée au couple de plus petit index.

3.3.2.3 Détection de la fin d'exécution

A– Opération de calcul : Lorsque tous les bits du registre masque du pipeline sont à 1, le déclenchement des composantes conformes est terminé. Il ne reste que l'écriture des composantes résultats, qui sont encore dans le pipeline, dans le registre cible. L'opération est complètement terminée lorsque de dernier bit masque du registre destination est mis à 1.

B– Opération d'accès : La détermination de la fin d'un accès diffère selon que l'accès est une lecture ou une écriture. Une lecture d'un vecteur en mémoire est terminée si tous les bits masques du registre vectoriel destination sont à 1. Le port ne peut pas être désalloué avant la réception de la dernière donnée depuis la mémoire.

En écriture mémoire, le mécanisme de détection de fin doit satisfaire les deux conditions suivantes :

- Détecter la fin de déclenchement des accès des composantes dans l'UCAE. Ceci est réalisé lorsque tous les bits masques du vecteur associé à l'UCAE sont à 1. C'est le travail de l'unité de sélection.
- Toutes les requêtes ont été envoyées vers la mémoire. Pour cela l'UCAE envoie un signal vers l'unité de sélection indiquant, à un instant donné, s'il y a des requêtes non encore envoyées.

3.3.3 Fonctionnement et contrôle

Pour expliquer le fonctionnement et le contrôle de l'exécution des unités vectorielles en traitement vectoriel désordonné nous considérons trois exemple d'exécution d'instructions. Le premier exemple met en valeur l'opération de chaînage et le calcul dans les pipelines, les deux autres sont les opérations gather/scatter.

3.3.3.1 Exemple 1

Soit l'exécution de l'expression suivante : $D = (A + B) * C$ sur des vecteurs de 12 éléments. A, B et D sont des vecteurs indépendants rangés en mémoire et C est un vecteur contenu dans le registre vectoriel $R2$. On suppose que la taille des registres vectoriels est de 16 éléments.

Cette expression a pour code :

```
Vmov @VL,          12
Vload @A,          R0
Vload @B,          R1
Vadd  R0, R1,      R3
Vmult R2, R3,      R4
Vstor R4,          @D
```

L'exécution de cette expression nécessite l'allocation de 4 registres vectoriels, deux pipelines de calcul (+ et *) et les trois ports d'accès mémoires (deux en lecture et un en écriture). Cette expression ne présente pas de dépendance des données, donc les opérations d'accès et

les opérations **Vadd** et **Vmult** seront chaînées. Ces 5 instructions de code sont déclenchées à raison d'une instruction par cycle.

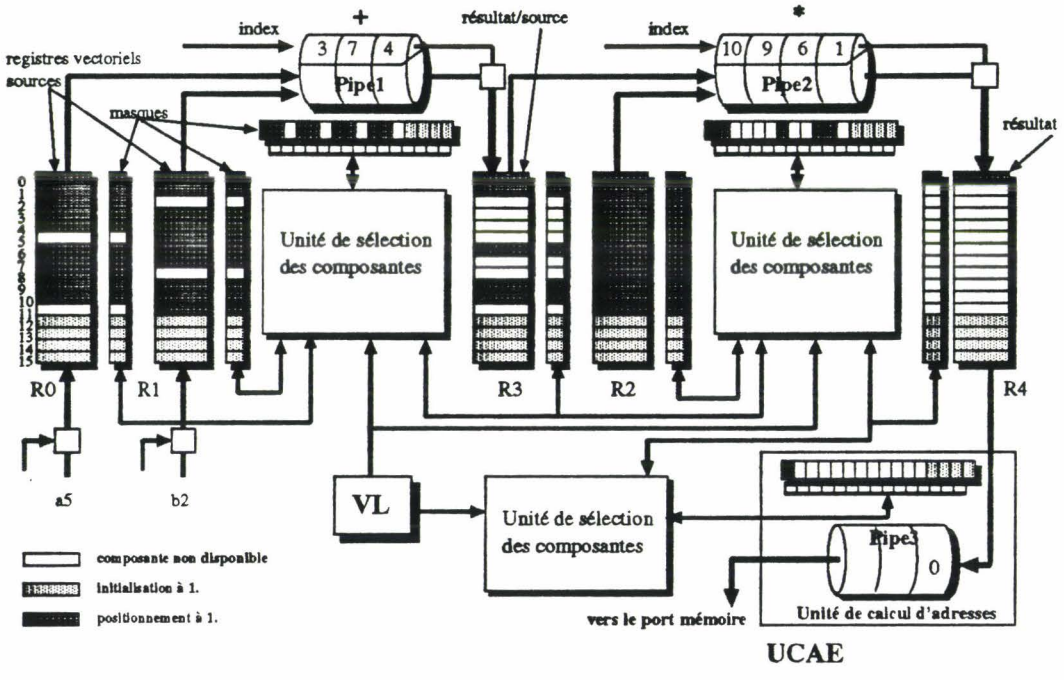


Figure 3.3 : Différentes unités fonctionnelles et leurs chaînage pour l'exécution de l'expression précédente

L'unité de sélection exécute pas à pas les points suivants :

1. Initialiser les masques des pipelines des unités de traitement et ceux des registres vectoriels résultats en fonction de la taille contenue dans **VL**. Les 12 premiers bits des masques des deux pipelines ($+$ et $*$), de celui de l'UCAE et de ceux de $R0, R1, R3$ et $R4$ seront mis à 0. Les 4 bits restant sont à 1; ils sont considérés comme étant déjà traités.

2. La figure (3.3) montre les différentes unités nécessaires à l'exécution de cet expression, leurs masques, ainsi que les différents chaînages. Afin d'éclaircir le principe de fonctionnement, on a schématisé chaque registre vectoriel avec son masque et chaque pipeline avec son masque et son unité de sélection des composantes. En fait tous ces masques et ces unités de sélection des composantes sont regroupés dans l'unité de sélection.

   Dans la configuration du schéma le premier pipeline ($+$) a traité les éléments $(0, 1, 6, 9, 10)$ et les éléments en cours de progression sont $(4, 7, 3)$. Le deuxième pipeline ($*$) a traité l'élément 0 et les éléments $(1, 6, 9, 10)$ sont en cours de traitement. Les masques des registres vectoriels cibles associés aux composantes traitées sont mis à jour. Les masques des pipelines et de l'UCAE sont positionnés pour les composantes traitées et en cours de traitement.

   On suppose qu'au cycle t, **1)** les composantes a_5 et b_2 arrivent dans les registres $R0$ et $R1$ respectivement, elles seront disponibles au cycle $t+1$. **2)** les composantes opérandes

d'index $(3, 10, 0)$ sont déclenchées dans les pipelines $(+)$, $(*)$ et dans l'UCAE respectivement. Les masques de ces unités de traitement pour les index $(3, 10, 0)$ sont mis à 1. Notons que les index de ces composantes opérandes sont propagés à travers tous les étages des pipelines.

Au cycle suivant $(t+1)$, on déclenche le couple de composantes d'index 2 dans le pipeline $(+)$. Les donnée d'index 4 et 1 sont écrites dans $R3$ et $R4$ respectivement. les bits masques correspondants sont mis à 1.

Au cycle $t + 2$, on déclenche les composantes opérandes $(5, 4, 1)$ respectivement dans les pipelines $(+)$, $(*)$ et dans l'UCAE, les masques respectifs sont mis à 1. Les composantes résultats $(7, 6)$ sont rangées dans les registres $(R3, R4)$. Les bits masques $(7, 6)$ des registres $(R3, R4)$ sont positionnés à 1. etc...

3. Lorsque la fin d'exécution dans les pipelines et dans l'unité adressage est détectée, les signaux de libération des ressources sont émis à l'unité de contrôle et de commande.

3.3.3.2 Exemple 2

Cette exemple, ainsi que l'exemple suivant illustrent la manipulation des opérations gather/scatter par l'unité de sélection. Considérons l'instruction qui permet de charger un vecteur dans un registre vectoriel contrôlé par un vecteur d'index: $A = B[C]$. Les vecteurs A est dans le registre vectoriel $R1$ et B et C sont des vecteurs en mémoire. C'est une opération gather qui peut s'écrire sous la forme:

```
Vload  @C,          R0
Gather @B, R0,      R1
```

Les deux instructions sont exécutées par l'unité d'adressage et elles sont chaînées. Le contrôle du gather et la fin de chargement de B dans A est réalisé par l'unité de sélection.

L'allocation des ressources est simple: on a besoin de deux registres vectoriels destination et de deux ports en lecture. Les masques des registres $R0$ et $R1$ et celui de l'UCAL (du gather) seront aussi alloués dans l'unité de sélection.

On commence par initialiser les masques des registres cibles et le masque de l'UCAL. L'instruction gather est déclenchée un cycle après l'instruction de chargement du vecteur d'index, mais son exécution est suspendue tant que les premières composantes de C ne sont pas encore disponibles dans $R0$.

Lorsqu'une composante c_i est disponible dans $R0$ l'opération gather consiste à calculer l'adresse $B[c_i]$. L'unité de sélection déclenche c_i et produit l'index de la composante (i). Le bit masque d'index (i) de l'UCAL est positionné à 1.

La fin de l'opération est réalisée lorsque tous les bits masque de $R1$ sont à 1.

3.3.3.3 Exemple 3

Le traitement de l'opération scatter ressemble au traitement d'une opération de calcul à deux opérandes. Soit l'instruction $B[C] = A$, où B et C sont des vecteurs en mémoire et A est un vecteur contenu dans le registre vectoriel $R0$. L'allocation des ressources et l'initialisation des registres masques sont identiques à l'instruction gather de l'exemple précédent. Cette

instruction peut s'écrire sous la forme :

```
Vload   @C,          R1
Scatter R0, R1,      @B
```

Notons que A (ou $R0$) peut être aussi résultat d'une autre unité fonctionnelle, de toute façon le traitement est le même que dans les unités fonctionnelles de calcul car le traitement est désordonné, en fonction des disponibilités des composantes.

Lorsque deux composantes a_i et c_i de $R0$ et $R1$ sont disponibles et conformes, l'unité de sélection déclenche a_i et c_i. c_i sert à calculer l'adresse mémoire de la donnée a_i. Le bit masque d'index (i) de l'UCAE est mis à 1.

La différence avec l'exemple précédent réside dans la détection de la fin de l'opération scatter. L'unité de sélection détecte la fin de calcul d'adresses, mais elle ignore la fin de réalisation de tous les accès, car on ne sait pas s'il reste encore des composantes bloquées dans le buffer. La fin de l'opération scatter doit satisfaire les deux conditions citées précédemment. C'est à dire tous les bits masques de l'UCAE doivent être à 1 et le buffer doit être vide pour s'assurer que l'accès a été fait à toutes les composantes.

3.3.4 Implémentation de l'unité de sélection

En résumé cette unité :

- initialise et gère les masques,
- détecte la fin d'exécution d'une instruction vectorielle,
- déclenche l'opération sur le couple de composantes et produit l'index correspondant.

Comme le montre la figure (3.4) cette unité est composée des registres masques des registres vectoriels et des unités de traitement, d'une logique de gestion et d'initialisation des masques, des unités de sélection des composantes et d'une unité de détection de la fin d'exécution.

Elle reçoit en entrée :

- les adresses des ressources allouées pour identifier leurs masques respectives,
- les signaux de positionnement des masques des registres cibles. Lorsqu'une donnée est rangée dans le registre cible, le masque associé doit être mis à jour dans l'unité de sélection.
- le signal de fin de l'UCAE.

Elle délivre en sortie :

- l'index des composantes disponibles et conformes sélectionnées pour chaque unité de traitement active,
- les signaux de fin des opérations pour la libération des ressources.

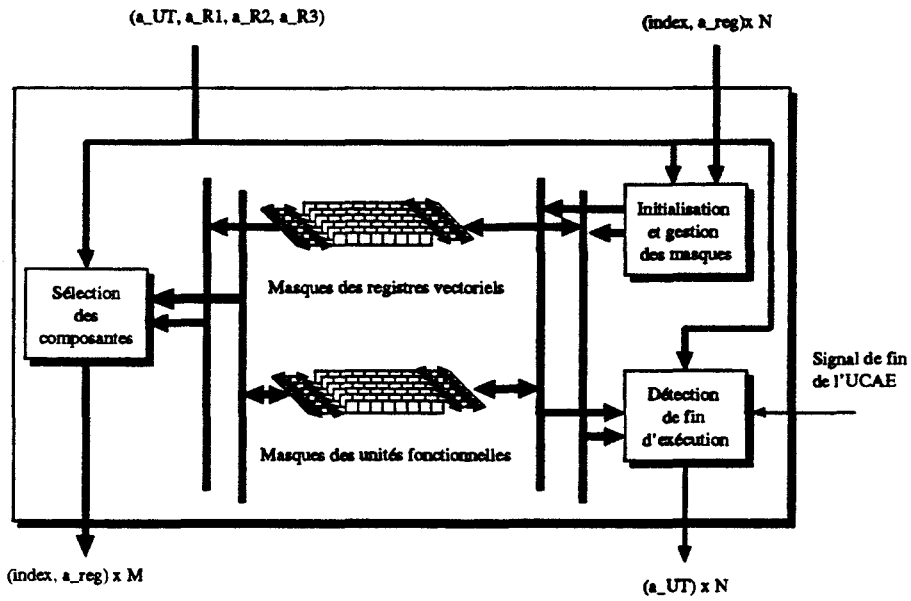


Figure 3.4 : Architecture de l'unité de sélection

3.4 L'unité d'adressage

L'unité d'accès mémoire est une unité indépendante des autres unités. On peut la considérer comme un processeur autonome dédié aux calculs d'adresses et à la gestion des accès mémoires. Elle peut piloter trois flux d'accès (un flux par port) : deux en lecture et un en écriture. Elle calcule les adresses de chaque flux et implémente une stratégie de sélection pour optimiser les accès sans conflit. Cette unité est composée de :

- trois unités de calcul d'adresses. Chaque unité prend en charge un flux d'accès.
- trois buffers; un pour chaque unité de calcul d'adresses.
- une unité de résolution des conflits qui est une implémentation d'une stratégie de sélection.

Toutes ces sous-unités travaillent en pipeline. L'ensemble forme un pipeline de calcul d'adresses et de résolution des conflits, appelé pipeline d'accès mémoire. Chacune des unités de calcul d'adresses associée à un buffer forment les premiers niveaux du pipeline d'accès mémoires. Ils travaillent en parallèle et servent d'entrée pour le niveau suivant qui est la résolution des conflits (voir la figure (3.5)).

3.4.1 Unité de calcul d'adresses

Les trois unités sont spécialisées : deux en lecture et une en écriture. Ainsi on peut activer simultanément deux vecteurs en lecture et un en écriture.

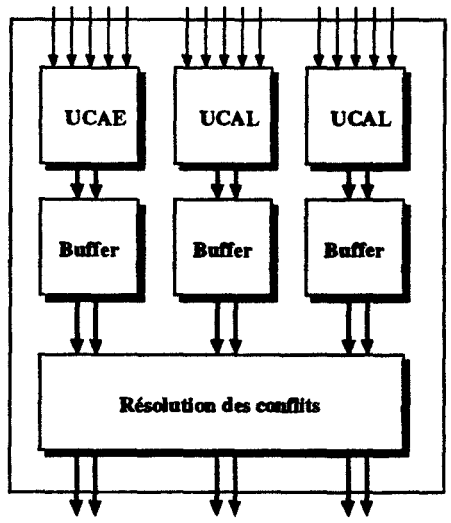


Figure 3.5 : Unité d'adressage

Les opérations Gather/Scatter sont prise en charge totalement par le hardware. Les deux unités spécialisées en lecture implémentent l'opération Gather et la troisième implémente l'opération Scatter.

Ces unités reçoivent en entrée certaines des caractéristiques des vecteurs qui sont:

- La taille du vecteur,
- L'adresse de base du vecteur,
- Le type d'accès : par pas régulier, gather/scatter,
- La valeur du pas d'accès pour les accès par pas régulier ou la valeur de l'index pour les opérations gather/scatter.
- La valeur de la composante à écrire en mémoire, uniquement pour l'unité de calcul d'adresse en écriture (**UCAE**).
- Pour l'écriture et gather/scatter, on reçoit, pour chaque accès, l'index de la composante à lire ou à écrire.

Chacune des deux unités de chargement (**UCAL**) produit en sortie l'adresse et l'index d'un élément du vecteur. L'adresse est calculée dans un pipeline d'addition, l'index est fourni par l'unité *"compteur/registre"*. Le fonctionnement de cette unité est contrôlé par le signal **Ta** : *"type d'accès"*. Si **Ta** est un accès régulier, l'index est calculé par incrémentation de la valeur du pas, sinon l'index est reçu depuis l'unité de sélection puis est transmis tel que en sortie. La figure (3.6) montre les différents composants de cette unité et les caractéristiques de chaque accès en lecture.

L'UCAE transfère, en plus de l'adresse et de son index, la donnée de la composante vers la sortie, elle sera rangée dans le buffer.

Le calcul d'adresses est réalisé suivant les deux cas de figure: les accès par pas régulier en lecture d'une part et les accès irréguliers (gather/scatter) et les écritures d'autre part.

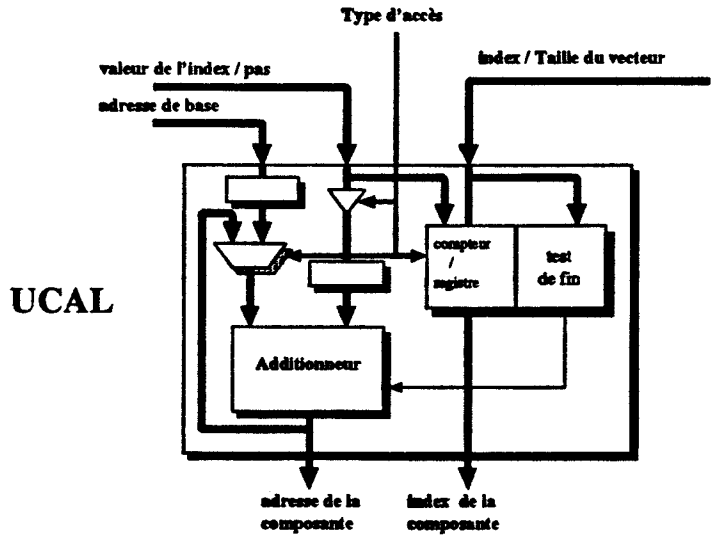


Figure 3.6 : Deux unités de calcul d'adresses en lecture

a- Accès par pas en lecture : Plus précisément l'UCAL reçoit :

- La taille du vecteur, elle sert à stopper le calcul des adresses localement.
- L'adresse de base : ab.
- Le pas d'accès : pas.

Soit $ad(i)$: l'adresse de la composante (i). L'adresse de la composante en cours est donnée par : $ad(i) = ab + i.pas = ad(i-1) + pas$. On peut remarquer que l'on a pas besoin de (i).

Un signal de fin est déclenché une fois que toutes les adresses des éléments du vecteur sont calculées afin d'arrêter le calcul des adresses dans l'UCAL.

b- Opération gather/scatter et écriture : L'UCA reçoit en entrée :

- L'adresse de base : ab.
- La valeur de l'index : $vind$, ou le pas d'accès : pas.
- L'index de la composante : i.

On peut remarquer que la taille du vecteur est inutile car le contrôle est réalisé par l'unité de sélection. L'adresse de la composante pour un gather/scatter en cours est donnée par : $ad(i) = ab + vind$.

En écriture, le calcul d'adresse est simplement $ad(i) = ab + i.pas$. La composante précédente n'est pas forcément le prédécesseur de (i). On a effectivement besoin de la valeur de (i) pour chaque composante. C'est le cas de l'exemple 1 où les données à écrire arrivent dans un ordre quelconque dans $R4$.

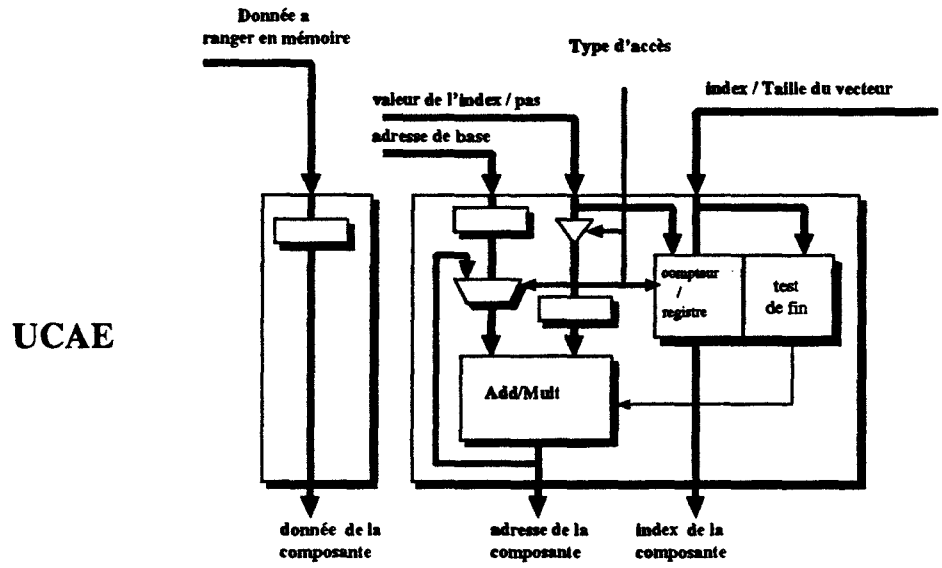


Figure 3.7 : Deux unités de calcul d'adresses en écriture.

Le registre d'index ou le registre à ranger en mémoire peut être résultat d'un pipeline d'une unité fontionnelle. L'ordre de disponibilité de ses composantes étant aléatoire, l'index de chaque élément est nécessaire pour calculer l'adresse effective de la donnée. Celui-ci est fourni en entrée et il est transmis en sortie.

Les valeurs des index et les données pour l'UCAE sont choisies par l'unité de sélection en fonction de leur disponibilité et de leur conformité.

3.4.2 Le buffer

Il a pour rôle de stocker les requêtes (adresse, index) en lecture ou (adresse, donnée, index) en écriture. De cette façon, l'unité de calcul d'adresses n'attend pas l'accès de la requête précédente pour calculer la suivante.

Le buffer est une file de registres qui permet des accès rapides et parallèles, et il est de la même taille qu'un registre vectoriel. Le buffer est exploré en parallèle à chaque cycle, pour identifier des bancs référencés pendant le cycle. Il permet l'écriture des nouvelles adresses envoyées par l'UCAL ou l'UCAE. Les adresses et données sont rangées aux l'emplacements désignés par leurs index [0 $\longrightarrow$VL]. Il n'y a pas de risque de collisions car l'index d'une composante est unique.

Un registre masque est associé à chaque buffer pour indiquer les adresses disponibles. Les adresses non disponibles sont soit déjà accédées en mémoire, soit elles ne sont pas encore calculées. Lorsqu'une adresse est accédée en mémoire le registre masque est aussitôt mis à jour pour éviter de la traiter une seconde fois. Ainsi un signal de retour est reçu depuis le port correspondant pour mettre à jour ce masque.

La figure (3.8) montre les différents éléments d'un buffer en lecture et en écriture.

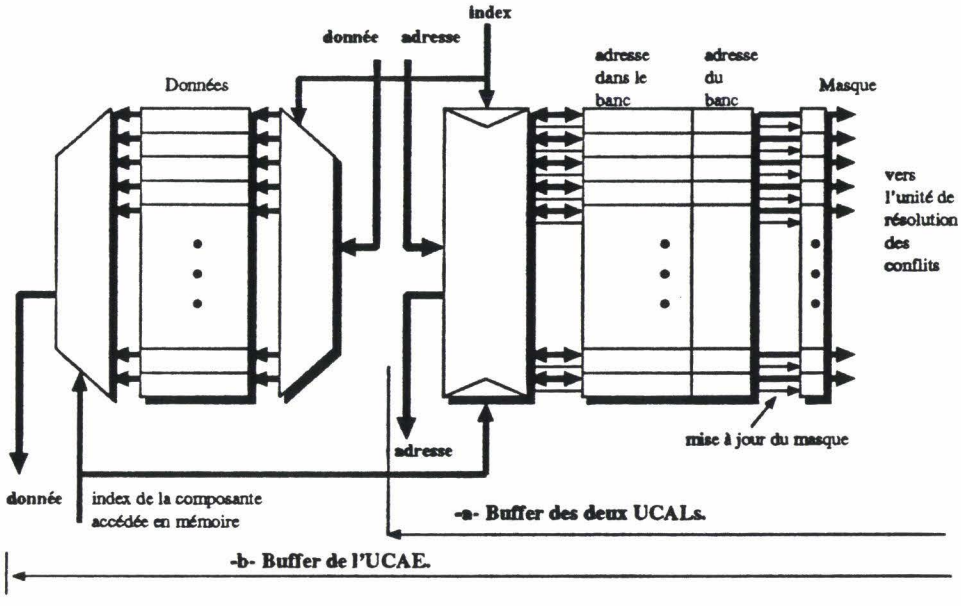


Figure 3.8 : Le buffer qui contient les requêtes bloquées.

3.4.3 Unité de résolution des conflits

Nous allons nous intéresser uniquement à la classe de stratégies décrites dans le chapitre précédent.

Cette unité est composée de deux sous-unités, chacune implémente une phase de cette classe de stratégies : *résolution des conflits de sections* et *résolution des conflits de simultanéité et de bancs occupés*. Les résolutions des conflits de sections et de simultanéité sont liées directement à l'implémentation propre des algorithmes utilisés.

L'unité de résolution des conflits reçoit en entrée les requêtes présentes dans le buffer. Les éléments du buffer sont explorés en parallèle afin de mettre à jour le vecteur de bits (un bit par banc mémoire) qui permet d'identifier les bancs référencés par des requêtes présentes dans le buffer. Ce vecteur est appelé *vecteur de référence*. La solution de file de registres pour représenter le buffer facilite la recherche et l'extraction des requêtes à accéder. Les actions exécutées par cette unité sont :

1. Pour chaque adresse disponible dans le buffer extraire les bits de poids faibles qui représentent l'adresse du banc.
2. Mettre à jour le vecteur de référence. L'index de la requête (ou d'une des requêtes) est aussi sauvegardé dans un registre associé au vecteur de référence qu'on peut appeler *registre de référence*. Ceci permet de retrouver l'entrée de la requête dans le buffer si la requête est candidate pour un accès mémoire, voir la figure (3.9).
3. Si plusieurs requêtes référencent le même banc, on garde l'index le plus petit.
4. L'action de résolution des conflits de sections, certainement la plus conséquente, consiste à répartir les sections sur les ports selon l'algorithme utilisé.

5. La dernière action constitue la deuxième phase consacrée à la résolution des conflits de bancs occupés et de simultanéité.

Dans ce qui suit nous présentons une implémentation pour chaque type de conflits séparément.

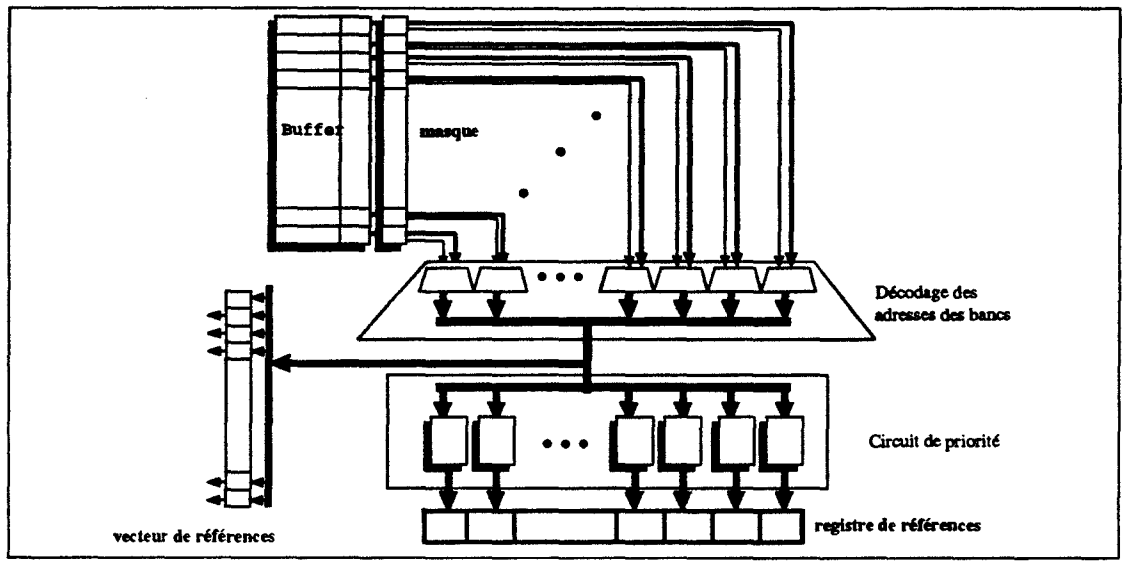


Figure 3.9: Implémentation de l'unité de résolution des conflits

3.4.3.1 Unité de résolution des conflits de sections

La résolution des conflits de section est implémentée directement comme elle a été décrite dans le chapitre précédent. Cette unité reçoit en entrée les trois vecteurs de références correspondants aux trois unités de calcul d'adresses et elle délivre en sortie un vecteur de bits (pour chaque port), appelé *registre section*, indiquant la section sur laquelle un port est autorisé à effectuer des accès. Les différentes actions effectuées sont montrées en figure (3.10).

Elle opère de la manière suivante :

- former la matrice d'optimisation. Son implémentation consiste à effectuer un `OU` logique sur les bancs d'une même section.
- optimiser la matrice. Son implémentation est fonction de l'algorithme d'optimisation, par exemple l'algorithme exposé dans le chapitre précédent (figure 2.13).
- produire les registres sections. Il suffit d'effectuer un `ET` logique avec les vecteurs de références reçus en entrée.

3.4.3.2 Les conflits de simultanéité et de bancs occupés

La résolution des conflits de simultanéité et de bancs occupés se fait en trois étapes :

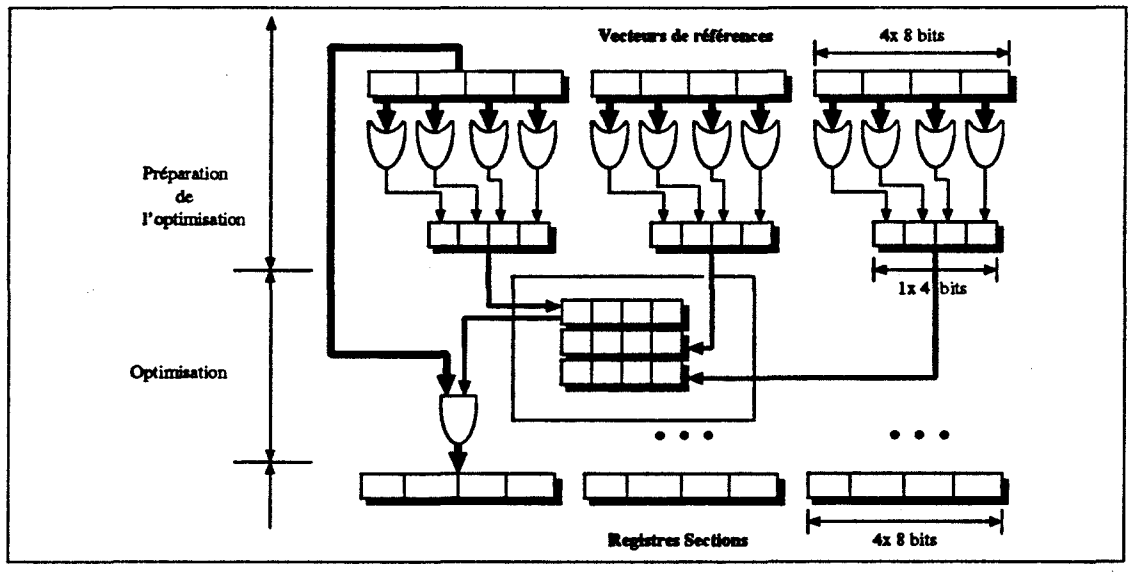


Figure 3.10 : Implémentation de l'unité de résolution des conflits de section.

- produire le vecteur masque des bancs pour lesquels le processeur est plus prioritaire. Ceci est en fonction de la stratégie mise en place.

- élimination des requêtes qui référencent des bancs non prioritaires. Elle est réalisée par un ET logique entre les registres sections et le vecteur masque des bancs en conflits de simultanéité.

- élimination des requêtes qui référencent les bancs occupés. Elle est obtenue par un ET logique entre les registres sections et le vecteur masque des bancs occupés produit par la mémoire à chaque cycle. Si plusieurs requêtes sont valides pour un accès, on prend celle d'index le plus petit.

Dans le cas où aucune requête n'est pas sélectionnée lors de cette opération on tentera quand même une requête indépendamment des conflits de simultanéité, sachant qu'elle n'est pas en conflits de bancs occupés. Cette unité est montrée dans la figure (3.11).

Une stratégie centralisée reçoit les vecteurs de bits indiquant les bancs référencés par chaque processeur. Elle marque les bancs en conflits de simultanéité en produisant un vecteur de bits correspondant à chaque processeur en tenant compte du système de priorité d'accès à la mémoire (figure 3.12).

Cette stratégie ne peut pas être implémentée en un seul étage du pipeline, elle nécessite au minimum trois cycles (émission, traitement, réception).

La stratégie décentralisée implémente un système de priorité d'accès à la mémoire. Les bits associés aux bancs indiqués par la règle de priorité sont positionnés à 1. Ainsi le vecteur masque est formé localement. On peut directement passer à l'élimination des requêtes en conflits de simultanéité. Cette stratégie est la plus simple à implémenter, (figure (3.13)) :

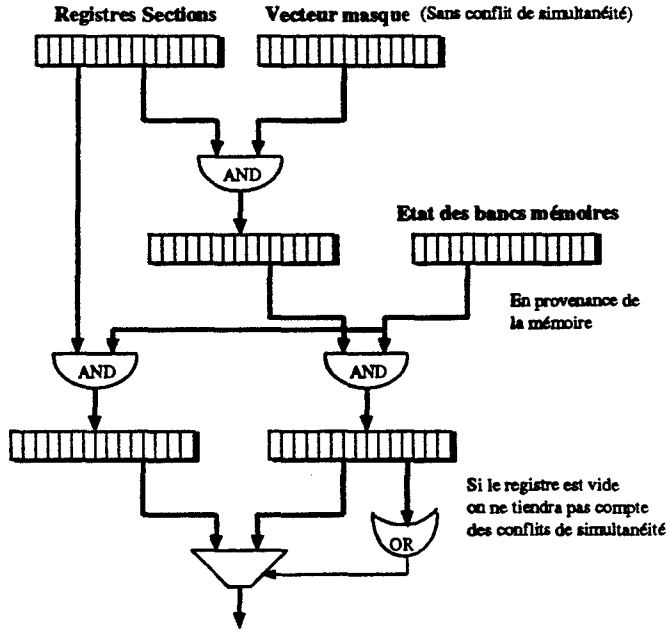


Figure 3.11: Mécanisme de sélection des requêtes qui ne sont en conflit de bancs et qui ne éventuellement pas en conflits de simultanéité.

Réalisation de l'accès d'une requête

Les conflits de simultanéité ne sont pas complètement résolus localement à un processeur; certaines requêtes sont tentées et peuvent causer des conflits de simultanéité. Par conséquent lorsqu'une requête est envoyée à la mémoire le masque du buffer n'est pas aussitôt mis à jour. Le bit masque associé à la requête n'est mis à jour que lorsqu'on reçoit la confirmation de son accès depuis le port au cycle suivant. Elle sera donc sélectionnée dans les étages inférieurs du pipeline mais lors de la dernière étape (le nombre d'étages du pipeline est inférieur au temps de latence de la mémoire) elle sera en conflit de banc occupé au prochain cycle.

Pour les accès en écriture nous avons vu précédemment que le test du buffer vide s'impose. Le buffer est considéré vide que si tous les bits masques sont positionnés à 0 (aucune requête n'est disponible dans le buffer). Si c'est le cas, un signal (buffer vide) est envoyé à l'unité de sélection.

3.5 Les ports

Les ports reçoivent les composantes susceptibles de réaliser un accès mémoire. En flux d'écriture, le couple (adresse, donnée) est envoyée au banc mémoire correspondant. En flux de lecture, l'index de la composante est rangé dans un registre à décalage dans le port et l'adresse est envoyée à la mémoire. La taille du registre à décalage est égal au temps de latence de la mémoire. Après un temps de cycle mémoire, la donnée est renvoyée de la mémoire. On forme les couples (donnée, index) dans chaque port de lecture, ils seront envoyés aux registres vectoriels destinataires, (voir la figure (3.14)).

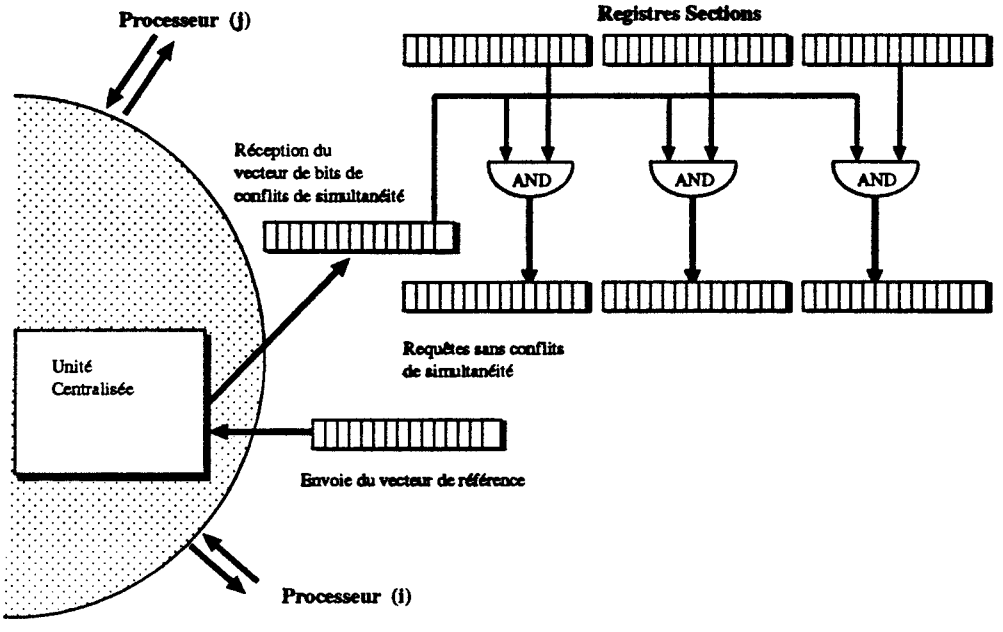


Figure 3.12: L'unité de résolution centralisée des conflits de simultanéité.

Les unités de résolution des conflits reçoivent depuis les ports mémoires une confirmation des accès ou des échecs. Cette information est gérée par le port. Si le banc mémoire renvoie la requête, le port émet un échec à l'unité d'adressage, sinon au bout d'un temps Te, qui est le temps de traversée du réseau dans les deux sens, le port émet un signal de succès qui avec l'index permet d'invalider l'entrée du buffer correspondant.

3.6 Les registres vectoriels

A cause des accès mémoire désordonnés, une composante d'un vecteur doit être identifiée de manière explicite par son index. Cet index, par définition, est l'adresse de la composante dans le registre vectoriel. Pour permettre le chaînage d'une opération d'accès mémoire avec une opération de calcul, on associe à chaque registre vectoriel un registre masque dans l'unité de sélection (figure (3.15)). Ce masque identifie les données disponibles.

Le flux d'accès et le flux de calcul sont complètement chaînés. Les registres vectoriels servent à la fois de source et destination pour les deux flux.

3.7 Les pipelines

La figure (3.16) montre le fonctionnement d'un pipeline. Il reçoit en entrée un couple composantes ainsi que leur index. A la sortie le couple (donnée, index) est produit. Le résultat (la donnée) est rangé dans le registre cible à l'emplacement adressé par son index.

Pendant que le calcul sur les deux opérandes progresse dans le pipeline, l'index, à son tour,

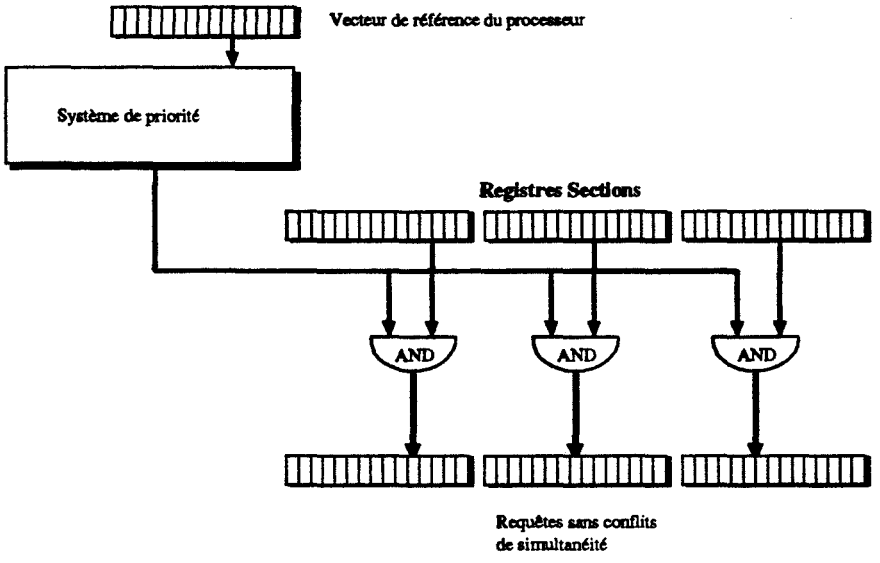


Figure 3.13 : L'unité de résolution décentralisée des conflits de simultanéité.

progresse dans un pipeline parallèle. L'index ne subit aucune opération de modification, il est uniquement décalé d'un étage à un autre. Ce pipeline parallèle peut être représenté par un registre à décalage. Le nombre et le temps de décalage sont respectivement égaux au nombre d'étages et le temps de cycle d'un étage (figure (3.16)).

3.8 Le réseau d'interconnexions

Le système de contrôle du réseau d'interconnexions est simplifié car les conflits de sections et de bancs occupés sont résolus dans les processeurs. Les conflits de simultanéité sont résolus par les bancs mémoires ou dans les processeurs. Le rôle du réseau est réduit :

- au cheminement des données entre les processeurs et les mémoires.
- à l'alignement des données. Ici l'alignement consiste à renvoyer la donnée au processeur et au port qui l'a envoyé.

Au niveau du réseau, la largeur des chemins de données et d'adresses restent inchangées car le réseau ne véhicule pas les index. Ces derniers sont gardés dans le port en attendant le retour des données. Néanmoins un bus de contrôle est créé, reliant les mémoires aux processeurs. Il permet aux bancs mémoires d'informer les processeurs de leurs l'état (libres ou occupés) à chaque cycle.

3.9 Les bancs mémoires

Un banc mémoire doit pouvoir informer les processeurs de son état à chaque cycle. L'ensemble de la mémoire produit un vecteur d'état, celui ci est diffusé à tous les processeurs via le réseau

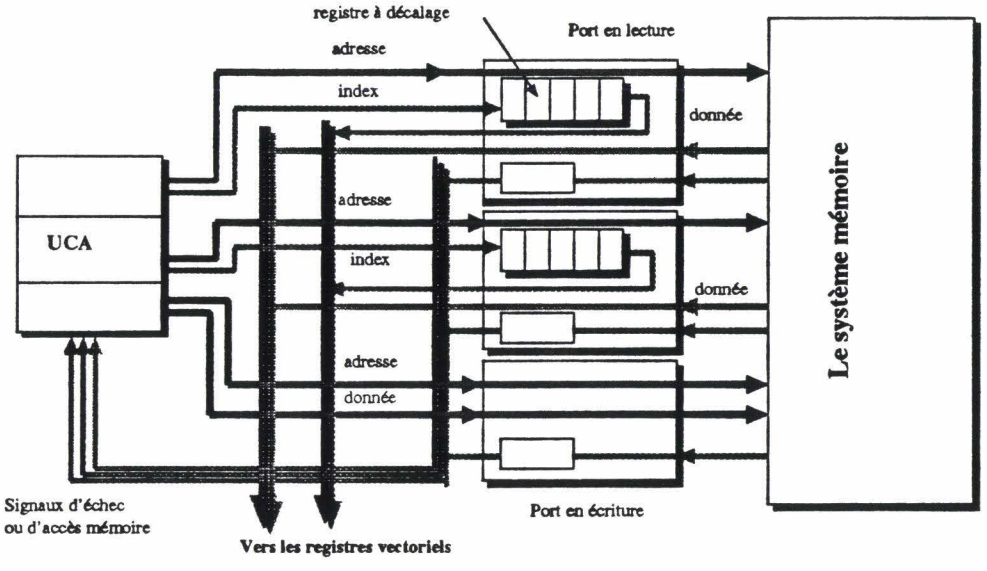


Figure 3.14 : Architecture des ports

d'interconnexions. Lorsqu'un conflit de simultanéité se produit, le banc envoie un signal d'échec aux ports propriétaires de ces requêtes.

3.10 Conclusion

Ce troisième chapitre présente l'implémentation du modèle de traitement vectoriel désordonné dans un environnement multiprocesseurs vectoriels pipelines à mémoire partagée.

L'implémentation de ce modèle est caractérisée par l'introduction d'une nouvelle unité qui est l'unité de sélection. Elle gère et contrôle toute instruction vectorielle de son déclenchement jusqu'à sa terminaison. Nous avons également apporté des modifications à toutes les unités concernées par le traitement vectoriel, notamment l'unité d'adressage et les pipelines de calcul.

D'une manière générale, l'unité de sélection implémente les propriétés de disponibilité et de conformité et quelques mécanismes pour le bon déroulement de l'exécution d'une instruction, le chaînage des pipelines, et la détection de la fin d'exécution d'une opération. Cette unité gère toutes les unités fonctionnelles vectorielles y compris l'unité d'adressage et les registres vectoriels.

Les pipelines de calcul et les ports d'accès en lecture doivent conserver l'index des composantes opérandes en cours de traitement (calcul ou accès) afin d'identifier l'emplacement de la donnée résultat dans le registre vectoriel destination. Ceci est faisable par l'association à chaque pipeline et au port d'accès en lecture d'un simple registre à décalage dont le nombre de décalage est égal au temps de latence du pipeline et de la mémoire respectivement.

L'unité d'adressage implémente le principe d'accès désordonné qui est la résolution des conflits de bancs occupés, les conflits de section et les conflits de simultanéité. Elle est caractérisée par une implémentation pipeline de chaque phase de traitement : phase de calcul d'adresses, phase d'écriture dans le buffer et phase de résolution des conflits (conflits de sections, conflits

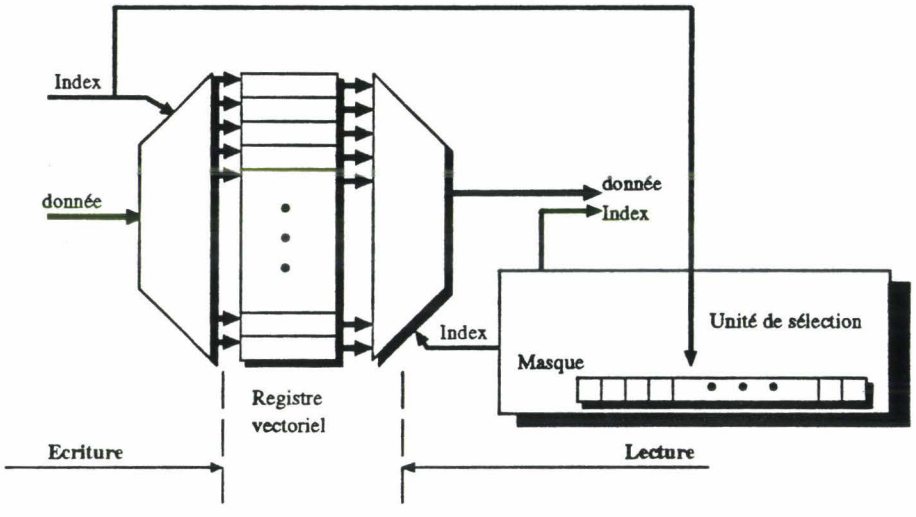


Figure 3.15 : architecture d'un registre vectoriel

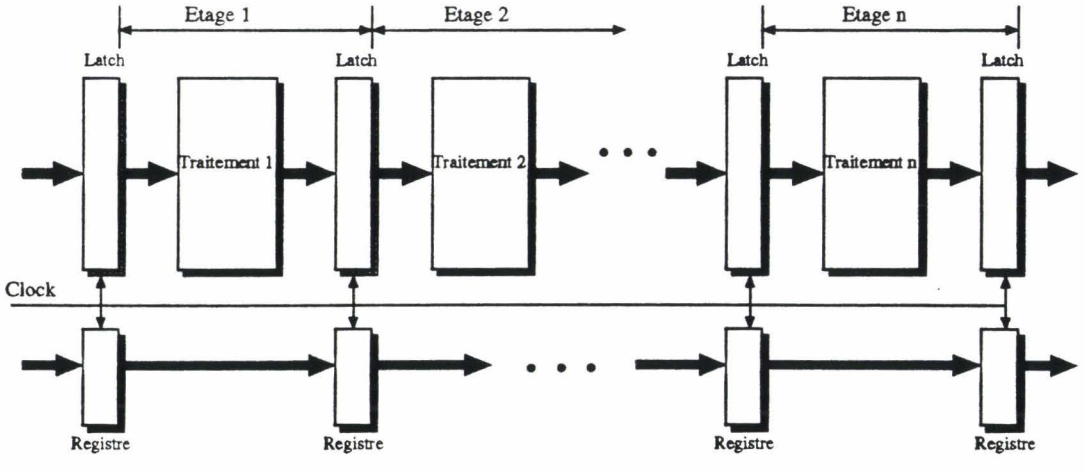


Figure 3.16 : Architecture d'un pipeline de calcul

de simultanéité et conflits de bancs occupés).

Nous avons montré que la résolution des conflits peut être faite de manière pipeline. La phase de résolution des conflits de sections peut être implémentée en un ou plusieurs étages suivant l'algorithme d'optimisation utilisé. A l'opposé les conflits de bancs occupés doivent être résolus en un seul cycle, il utilisent le vecteur d'état de la mémoire qui varie d'un cycle à un autre. Le circuit de résolution de ce type de conflits se réduit à un **ET** logique entre les registres sections et le vecteurs d'état de la mémoire qui nécessite juste le temps de traversée d'une porte **AND** à deux entrées.

L'implémentation de cette classe de résolution peut être améliorée en éliminant certaines requêtes qui sont en conflits de bancs occupés avant la phase de résolution des conflits de sections. Supposons que le pipeline de résolution est un 2-étages et que le temps de latence de la mémoire est de 4 cycles processeur. Si à un instant t un banc est occupé pendant 4 cycles par exemple, on peut anticiper l'élimination des requêtes qui référencent ce banc parce qu'à l'instant $t + 2$ le banc reste encore occupé.

Nous avons implémenté une classe de stratégies de résolution des conflits de section et de simultanéité que nous avions défini dans le chapitre précédent. Cette classe a l'avantage de supporter une implémentation pipeline. L'implémentation de cette classe de stratégies a permis de réaliser un prototype du processeur vectoriel désordonné (PVD) [Dekeyser et al.91]. Les unités essentielles sont présentées en Annexes.

Chapitre 4

Modèle Analytique et Mesure de Performances

4.1 Introduction

La modélisation est un moyen rapide et économique pour étudier un système en explorant un large espace de contraintes sur ce système. Elle évite de construire le système, et d'observer son comportement durant une assez longue période. D'une manière générale la modélisation d'un système nous permet d'étudier son comportement, par conséquent ses performances, en fonction d'un certain nombre de contraintes matérielles ou fonctionnelles. Ici les contraintes matérielles représentent le hardware de la machine et les contraintes fonctionnelles représentent les communications entre différentes unités et leurs fonctionnements internes. Nous n'étendrons pas cette étude au fonctionnement global de la machine. Nous nous intéresserons uniquement au problème d'échange d'informations entre les processeurs et la mémoire en mode vectoriel.

L'évaluation des performances de la machine à traitement vectoriel désordonné est effectuée avec deux techniques différentes qui sont la technique de simulation et la technique des files d'attentes. Cette dernière est caractérisée par son meilleur rapport complexité/coût des efforts déployés pour développer les modèles des calculateurs [Allen80], [Sauer et al.80]. Cependant, les calculateurs actuels sont souvent très complexes et pour simplifier le modèle on est amené à poser quelques hypothèses d'approximations. Par exemple, dans un système multiprocesseur, on peut ignorer les interférences du réseau d'interconnexions; on peut aussi supposer que les accès à la mémoires sont uniformément distribués sur l'espace mémoire, ce qui n'est pas toujours vérifié, comme l'accès aux éléments d'un vecteur contigu ou à des instructions d'un programme qui sont dans un espace contigu de la mémoire, etc. Ces hypothèses peuvent invalider le modèle si elles sont très sévères. De ce fait, l'évaluation des performances doit être faite sous différentes méthodes et on compare leurs résultats, car les résultats d'une technique peuvent contester ceux d'une autre. Pour cela nous avons simulé notre modèle, en tenant compte de toutes les contraintes posées sur des machines réelles, pour conforter nos résultats. Les résultats de simulation seront présentés dans le chapitre qui suit.

Nous commençons par présenter le phénomène d'attente et le processus de Markov, puis les différents modèles construits autour des systèmes multiprocesseurs. Les sections 4 et 5 présentent la modélisation du TVD et celle du modèle classique. Comparaison et mesures de performances sont discutées en section 6.

4.2 Phénomène d'attente, processus de Markov

Le phénomène d'attente est la technique de modélisation la plus utilisée à cause de la facilité de construction et de la simplicité de modification du modèle [Spragins80], [Pack77, Pack78], [Coffman jr et al.78]. Cette technique étudie le système en terme de files d'attentes ("queues") et des stations de services (figure 4.1). Les entités qui circulent dans le système sont appelées "clients". Chaque client doit passer dans une station pour recevoir un service ou subir une certaine opération. Le service est en général aléatoire de sorte que les unités peuvent avoir à attendre avant qu'une station soit libre. Elles séjourneront dans le centre d'attente constitué d'une ou plusieurs files d'attentes. Un tel système est caractérisé par :

- le nombre de stations de services et leurs disposition (parallèle ou série),
- la méthode de choix des unités dans le centre d'attente (FiFo, aléatoire, ...),
- le processus de distribution des arrivées (processus de Poisson, processus de Gauss, ...)
- le processus de distribution des durées de service des clients.

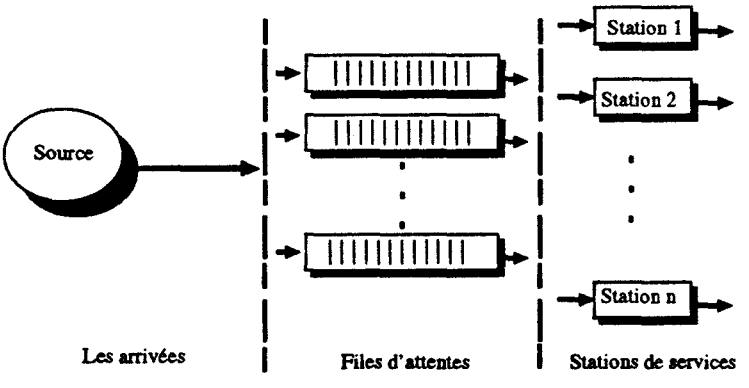


Figure 4.1 : Synoptique d'un système de files d'attentes

Cette technique est souvent utilisée pour un caractère économique. L'attente est coûteuse et pour l'éviter il faut augmenter la capacité du service qui demande des investissements importants et qui risquent d'être peu rentables. Alors on cherche à minimiser la somme des coûts de l'attente des clients et du coût de l'inactivité des stations en agissant sur quelques paramètres (nombre de stations, nombre de files, ...). Elle est aussi utilisée pour comprendre le comportement d'un système sous un certain nombre de contraintes : le calcul des débits, la taille des files d'attentes, etc.

L'étude du système de files d'attentes est envisagée en recherchant comment l'aléatoire se manifeste sur le comportement global du système. Elle consiste à observer l'état du centre de service c'est-à-dire le nombre de clients dans le centre de service à un instant t. Ainsi, le système réel est représenté par l'ensemble des états $\Gamma = \{s^t ; \quad s^t$: *le nombre de clients dans l'installation à l'instant* $t\}$ décrivant le centre de service à chaque instant. Le phénomène aléatoire intervient dans la détermination des instants t_i de changement d'état et dans la détermination de l'état $s^{t_{i+1}}$ qui succède s^{t_i}. On peut remarquer que l'état $s^{t_{i+1}}$ dépend également des processus des arrivées et des fins de service.

Soit $P_s(t)$ la probabilité que le système soit dans l'état s^i à l'instant t. L'état du système sera défini par les probabilités: $(P_0(t), P_1(t), \cdots, P_n(t))$, qui caractérisent l'ensemble des états possibles du système à un instant donné t. Cet ensemble est appelé *vecteur aléatoire ou vecteur des probabilités.*

Le passage d'un état s^{t_i} à un autre état $s^{t_{i+1}}$ (ou *transition*) est exprimé par la probabilité de transition $P_{s^{t_i}, s^{t_{i+1}}}$. D'une manière générale, cette probabilité dépend de l'évolution antérieure du système.

Un phénomène d'attente est dit **"processus de Markov"** si l'état actuel du système et la probabilité de transition vers les états suivants permettent de décrire complètement le comportement futur du système [Deitel84]. Cette propriété est dite *propriété sans mémoire.*

4.3 Les travaux antérieurs sur la modélisation des calculateurs

Une analyse complète des performances des systèmes multiprocesseurs doit être faite sur la base d'un modèle qui tient compte des facteurs suivants:

- la structure des processeurs,
- la structure de la mémoire,
- la structure du réseau d'interconnexions entre les processeurs et les mémoires,
- le comportement des programmes sur chaque processeur.

4.3.1 Structure des processeurs

La description de la structure des processeurs renferme:

1. le nombre de processeurs,
2. la vitesse de chaque processeur,
3. les communications entre les processeurs,
4. l'organisation multiprocesseur,
5. le nombre de requêtes en attente et qui peuvent être bufferisées dans chaque processeur
6. le nombre de requêtes qu'un processeur est capable d'envoyer à la mémoire simultanément.

La construction d'un modèle qui tient compte de tous ces facteurs est complexe et certainement difficile à analyser. Pour simplifier, on fixe quelques paramètres, par exemple, absence de communications entre les processeurs, processeurs identiques et indépendants, etc.

En général, les performances des systèmes multiprocesseurs se mesurent en terme de débit de données et/ou d'efficacité d'exécution. Les travaux effectués jusqu'à présent s'intéressaient plutôt au problème de transfert de données entre les processeurs et les mémoires, car le problème majeur de ces systèmes réside dans le réseau d'interconnexions, très coûteux et très complexe

à réaliser, qui constitue un goulot d'étranglement entre les processeurs et les mémoires. Sur ce genre de système, on ne se soucie pas vraiment de l'architecture des processeurs, car leurs performances sont avant tout limitées par le débit de la mémoire.

On peut classer en quatre groupes les structures des processeurs étudiées jusqu'à présent. Ces groupes sont définis suivant le nombre de requêtes qu'un processeur est capable d'envoyer à la mémoire simultanément à chaque instant :

1. Les modèles de structure monoprocesseur monoport (SPSP).
2. Les modèles de structure monoprocesseur multiport (SPMP).
3. Les modèles de structure multiprocesseur monoport (MPSP).
4. Les modèles de structure multiprocesseur multiport (MPMP).

La différence entre ces systèmes réside dans le type de conflits générés et dans le débit effectif de requêtes émis à la mémoire en concurrence. Le système SPSP ne présente pas de parallélisme. C'est une architecture classique de Von-Newman.

4.3.1.1 Les systèmes SPMP

Les systèmes SPMP sont limités par le débit effectif d'un processeur et par les dépendances entre les requêtes successives. Hellerman [Hellerman66] a proposé un modèle où il a exprimé la bande passante du système. Dans son modèle il suppose qu'il n'y a pas de dépendances entre deux requêtes successives et que le processeur ne dispose pas de file d'attente pour stocker les requêtes en attente d'accès. Ainsi le processeur reste bloqué sur la requête en cours jusqu'à la libération de la mémoire. La modélisation des dépendances entre les requêtes fut introduite par Burnett & Coffman [Burnet et al.70, Burnett et al.75] qui ont construit deux modèles séparés pour les instructions et les données. Le modèle des données constitue une généralisation du modèle d'Hellerman. Flores [Flores64] a étudié le modèle en considérant que le processeur dispose d'une file d'attente. Les requêtes sont générées suivant une loi de Poisson et la discipline FiFo est appliquée sur la file d'attente.

4.3.1.2 Les systèmes MPSP

Les premiers travaux sur les systèmes MPSP furent ceux de Skinner et Asher [Skinner et al.69] qui ont développé un modèle exact, en utilisant le processus de Markov, mais leur étude est limitée à 2 processeurs. Il est beaucoup plus complexe d'en déduire des équations générales pour un système ($N \times N$). Pour résoudre ce problème, Baskett & Smith [Baskett et al.76] ont développé un modèle asymptotiquement exact, en supposant que les requêtes sont émises à la mémoire suivant une loi binomiale. Ce modèle donne de bons résultats aussi bien pour des grandes valeurs que pour des petites valeurs de M et N, mais cette hypothèse est très restrictive. En procédant d'une autre manière Rau [Rau79] a proposé un autre modèle approximatif et symétrique pour d'assez grandes valeurs de M et N et il a retrouvé exactement l'expression de Baskett & Smith sans supposer que les arrivées suivent une loi binomiale. Bhandarkar [Bhandarkar75] a utilisé un modèle de chaînes de Markov discret pour évaluer de manière exacte les performances (la bande passante) des systèmes multiprocesseurs. Ce modèle, étudié aussi par Ravi [Ravi72], est plus proche des systèmes multiprocesseurs réels. Cependant, son étude s'est restreinte au cas où le processeur émet une seule requête par cycle en considérant

que $t_w = t_p$ (t_p est le temps d'exécution d'une instruction et t_w est le temps de rafraîchissement de la mémoire). Il a déduit que le résultat de Streker peut être amélioré en montrant que la bande passante est symétrique par rapport à M et N. Ce type de processeur a un débit de données par unité de temps inférieur à 1, puisqu'il exécute aussi des instructions qui ne produisent pas de requêtes mémoires. Yen & al. [Yen et al.82] a généralisé l'expression de la bande passante en considérant aussi cette hypothèse. Une extension du modèle pour $t_w > t_p$ est décrite dans [Smilauer85] et [Hoogendoor77].

4.3.1.3 Les systèmes MPMP

Les systèmes MPMP doivent leurs mérites avant tout au progrès technologique qui a fait qu'un processeur intègre plusieurs unités fonctionnelles qui peuvent travailler en concurrence. Par conséquent, celles-ci demandent le chargement de plusieurs données depuis la mémoire en parallèle. Si beaucoup de modèles ont été construis autour des systèmes MPSP et SPMP, il n'en est pas de même pour les systèmes MPMP.

Sastry & al. [Sastry et al.75] ont proposé un modèle MPMP à deux ports. Ils n'ont modélisé que des instructions à une seule adresse de sorte qu'à chaque accès, un processeur puisse charger l'instruction suivante et l'opérande de l'instruction en cours d'exécution. Leur modèle présente l'originalité d'anticiper le chargement de l'instruction suivante, mais il ne supporte pas les instructions sans opérande. En principe le modèle des instructions sans opérande est MPSP.

Il faut attendre l'apparition des machines vectorielles pour voir de vrais processeurs multiports. Tang & Mendez [Tang et al.89] ont proposé un modèle probabiliste des calculateurs multiprocesseurs vectoriels. Ils ont évalué l'efficacité de transfert de données entre la mémoire et une unité de calcul. Ils ont montré que la dégradation des performances due aux conflits d'accès à la mémoire est supérieure à 30% quand plus de trois ports sont actifs simultanément. Bucher & al. [Bucher et al.90] ont montré que le temps d'attente d'une requête est le produit du temps de réservation par le taux d'utilisation de la mémoire sur un système MPMP à P flux d'accès mémoires. On a aussi montré qu'à efficacité constante, le nombre de processeurs et le nombre de bancs mémoires doivent varier d'un même facteur k [Bailey87]. Ceci est validé par des simulations des opérations vectorielles sur le Cray-2 [Calahan et al.88].

Ces modèles décrivent uniquement les requêtes uniformément distribuées sur les bancs mémoires. Une partie simulation complète le modèle pour des requêtes non uniformes, par exemple l'accès aux vecteurs contigus ou par pas constant.

4.3.2 Structure de la mémoire

La structure de la mémoire est influencée par les facteurs suivants:

1. l'organisation mémoire (des bancs et des données),
2. le nombre de modules,
3. le temps de cycle de chaque module et sa taille,
4. la taille d'un banc et d'un mot mémoire,
5. type d'instructions mémoires,
6. les priorités d'accès etc.

La majorité des littératures sur le système mémoire des multiprocesseurs ont modélisé sa structure de manière raisonnable: utilisation des mémoires parallèles identiques et indépendantes accessibles par tous les processeurs.

Un autre type d'organisation, appelé ***organisation* L-M** [Briggs et al.77], est introduit dans le but de simplifier le réseau d'interconnexions. La mémoire est un tableau à deux dimensions. Les éléments du tableau sont les bancs mémoires tels que tous les bancs d'une même ligne sont reliés par un seul bus d'accès. (voir la figure 4.2). Nous l'avions utilisée pour implémenter le modèle de traitement vectoriel désordonné en remplaçant le bus par le multi-bus et les bancs monoport par les bancs multiports.

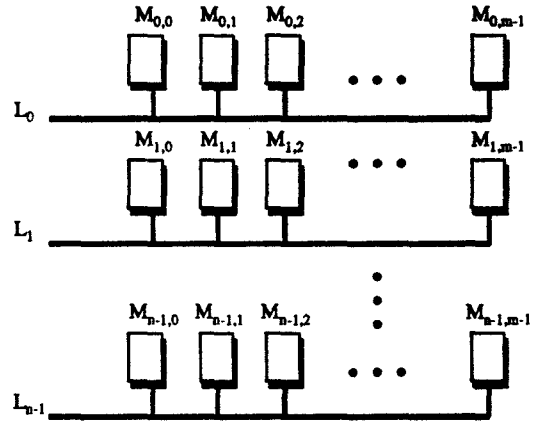


Figure 4.2: L'organisation **L-M** de la mémoire

L'organisation **L-M** introduit la notion d'entrelacement à deux niveaux. Dans cet esprit Burnett & al. [Coffman jr et al.71] ont modélisé les accès aux super-mots; c'est-à-dire l'accès est réalisé sur tous les bancs d'une même colonne et à la même adresse. Cette organisation est proposée et étudiée par Briggs & al. Leur modèle est basé sur le processus de Markov. Les auteurs ont évalué les performances suivant les deux types de conflits; les conflits de lignes occupées et les conflits d'accès simultanés à la même ligne. Dans leurs hypothèses, ils ont considéré que le temps de traversée d'une ligne est beaucoup plus grand que le temps de cycle mémoire. Cette notion de liaison des bancs d'une même ligne peut être étendue aux colonnes. Un système multiprocesseur muni de cette organisation (liaison des bancs en ligne et en colonnes) est appelé système multiprocesseur orthogonal (OMP) [Hwang et al.89].

Toutes les études référencées dans cette section ont montré l'influence du facteur 2 (le nombre de bancs) sur le débit de la mémoire. Le facteur 3 (le temps de cycle de la mémoire) partitionne cette littérature en deux: Celles qui considèrent que le temps de cycle de la mémoire est égal à celui du processeur et celles qui considèrent que le cycle mémoire est un multiple de celui du processeur. La deuxième hypothèse est beaucoup plus proche de la réalité, puisque les mémoires sont souvent plus lentes que les processeurs. Avec la dernière hypothèse, on a montré que si le temps de réservation d'un banc mémoire est augmenté d'un facteur k, il faut augmenter d'un facteur k^2 le nombre de bancs mémoires pour garder l'efficacité du système constante [Bailey87]. Le nombre de bancs mémoires doit être suffisamment grand par rapport au temps de latence de la mémoire pour obtenir des performances "raisonnables" du processeur [Creange et al.88].

4.3.3 Comportement des programmes

Très peu de tentatives ont été investi dans la modélisation des comportements de programmes dans un environnement multiprocesseur. A cause de la complexité du modèle et des difficultés d'analyse des résultats, la majorité des auteurs construisent des modèles simples en supposant que les requêtes sont indépendantes et uniformément distribuées sur la mémoire, puis ils procèdent à des simulations pour évaluer l'erreur commise. Souvent une part importante de cette erreur émane de l'effet des comportements des programmes.

Nous pensons que pour étudier le comportement d'un programme on doit tenir compte des 3 facteurs suivants :

1. l'ordre dans lequel les modules mémoires sont référencés,
2. le timing du comportement des requêtes
3. les dépendances logiques qui existent entre les requêtes mémoires.

Le comportement des programmes peut se modéliser en terme de dépendances entre les requêtes mémoires de chaque processeur. Nous dirons que deux requêtes sont en dépendance logique si l'une des deux empêche l'émission de l'autre lorsqu'elle n'a pas encore été émise ou qu'elle est en cours d'accès mémoire.

On distingue jusqu'à présent deux types de tentatives de modélisation des comportements des programmes. Celles qui supposent que le processus d'émission des requêtes suit une loi de probabilité : (loi de Poisson ou loi exponentielle) [Flores64], [Coffman jr68], et celles qui affectent des probabilités d'accès à des requêtes [Burnet et al.70, Burnett et al.73]. Par exemple, le modèle défini dans [Sethi et al.79] considère que si la k^{eme} requête est dans le banc (i), la $(k+1)^{eme}$ référencera le banc (i) avec une probabilité α et le banc $j, (j \neq i)$ avec une probabilité $(1-\alpha)/(m-1)$ (m est le nombre de bancs). La première requête référence un banc mémoire avec la probabilité $(1/m)$. De cette façon, on identifie les dépendances entre deux requêtes par la probabilité α.

Chang & al. [Chang et al.77] ont identifié 4 classes de dépendances, représentées à la figure (4.3). Cependant, ils n'ont fait aucune tentative pour les modéliser.

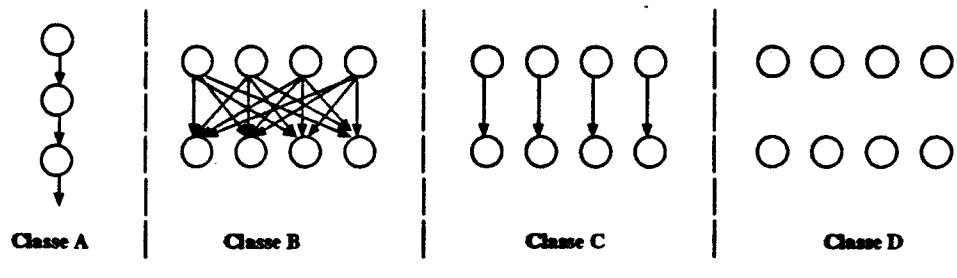


Figure 4.3 : Différentes classes de dépendances de Chang

La classe A suppose qu'on ne peut émettre une adresse que si la précédente a réalisé un accès mémoire. Cette exécution est identique à celle du modèle classique de Von-Newman. Cette classe est l'opposée de la classe D qui considère que toutes les requêtes sont indépendantes. L'exécution du code vectoriel sur un calculateur multiprocesseur multiport en traitement vectoriel désordonné est un modèle de classe D.

La classe B est une dépendance entre deux ensembles de requêtes indépendantes E_1 et E_2. Les requêtes de l'ensemble E_1 dépendent de plus de deux requêtes dans E_2. C'est une dépendance forte. Cette classe peut être identifiée à un traitement SIMD sur une machine massivement parallèle.

Dans la classe C une requête de E_1 dépend d'une et une seule requête de E_2. L'exécution des accès à des vecteurs indépendants sur une machine multiprocesseur (le C-90, par exemple) est de classe C.

Dans les sections qui suivent nous avons modélisé le modèle classique qui est un modèle de classe C et notre modèle de traitement vectoriel désordonné qui est de classe D.

4.4 Modélisation du traitement vectoriel désordonné

Le modèle de traitement vectoriel désordonné est implémenté sur un système multiprocesseur multiport à mémoire commune et entrelacée. C'est un système MPMP. Les deux modèles que nous allons présenter ne considèrent que l'échange de données vectorielles entre les processeurs et la mémoire. Nous supposons, pendant l'exécution d'un programme pour les deux modèles, que les accès scalaires (aux instructions d'un programme ou aux données scalaires) et les accès vectoriels sont supposés être faits de manière séparée. Seules les performances du traitement vectoriel sont évaluées.

4.4.1 Modèle et Hypothèses

Le système considéré est multiprocesseur multi-ports partageant une mémoire entrelacée. Il présente les caractéristiques suivantes :

1. Le système a N processeurs et M bancs mémoires identiques et indépendants. Chaque processeur est doté de L ports pouvant émettre ou recevoir plusieurs données en parallèle. Les ports sont identiques et indépendants au niveau fonctionnement, c'est-à-dire qu'ils ne sont pas spécialisés pour un type d'opérations donnée (lecture, écriture, gather/scatter) et il n'existe pas d'interaction entre eux. Ainsi chaque port a la même probabilité de référencer la mémoire. Pour des raisons matérielles cette hypothèse n'est pas vérifiée sur toutes les machines existantes.

2. L'échange de données entre les processeurs et les mémoires transite par un réseau d'interconnexions. Ce réseau assure la liaison de tout port d'un processeur à tous les bancs mémoires de façon indépendantes. Cette hypothèse assure une distribution équitable des lignes du réseau pour accéder à la mémoire.

3. l'unité de temps est le cycle processeur (*cp*). le système ne change pas d'état pendant le même cycle.

4. On suppose que les requêtes produites à chaque cycle sont uniformément distribuées sur l'ensemble des bancs mémoires. Ainsi à chaque accès, avec une probabilité $\frac{1}{M}$, un banc mémoire est référencé indépendamment de l'accès précédent.

5. Un processeur produit L requêtes par cycle qui sont rangées dans les files d'attentes des ports respectifs. Chaque port tente une requête à chaque cycle vers la mémoire. Si elle n'est pas acceptée, elle est renvoyée au port pour une tentative ultérieure. La file

d'attente est gérée de manière aléatoire; le port tente les requêtes qui référencent les bancs libres indépendamment de leur ordre d'arrivée.

6. Chaque banc mémoire est muni d'un système d'arbitrage permettant de choisir de manière aléatoire un parmi les ports quel'ont référencés. Cette hypothèse garantit l'équiprobabilité d'accès à la mémoire entre tous les processeurs du système.

Les hypothèses 1, 2 et 6 nous permettent d'affirmer que les comportements des ports (et des processeurs) sont semblables; les ports sont identiques et indépendants (d'après les deux premières hypothèses) et ils sont en équiprobabilité d'accès à la mémoire (d'après l'hypothèse 5). Par conséquent l'étude du phénomène d'échange de données entre les processeurs et la mémoire dans le système revient à étudier le sous-système *port↔mémoire* .

De part ces hypothèses ce système ainsi défini, peut être modélisé par un processus de files d'attentes. La source des arrivées est le processeur, (plus précisément l'unité de calcul d'adresses), le port est une file d'attente et les bancs mémoires sont des stations de service, comme le montre la figure 4.4. La file d'attente est supposée être de taille infinie afin que toutes les arrivées soient contenues dans la file d'attente.

pour résoudre le problème d'ergodicité (saturation de la file). Dans la pratique le problème d'ergodicité dans le processeur vectoriel désordonné ne se pose pas; la taille de la file est égale à celle des registres vectoriels.

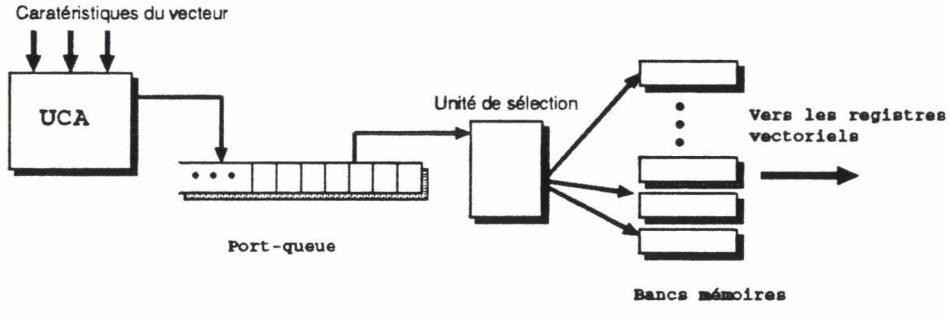


Figure 4.4: Système avec une file d'attente de taille infinie

4.4.2 Les paramètres

En traitement vectoriel nous allons nous intéresser uniquement à l'activité du port; c'est-à-dire à l'échange de requêtes du port avec la mémoire. L'étude du système *port↔mémoire* consiste à observer l'état des requêtes en attente ou en service à tout instant. Pour pouvoir définir ces états décrivant le comportement du système *port↔mémoire* , la définition des probabilités suivantes est nécessaire :

Soit p la probabilité qu'un port accède à un banc mémoire. Ceci implique qu'il existe au moins une requête dans la file d'attente qui peut accéder à la mémoire sans qu'elle soit bloquée suite à un conflit d'accès. Cette probabilité englobe les effets des trois types de conflits qui sont:

1. le taux d'occupation des bancs mémoire (le rapport du nombre de bancs occupés sur le nombre total de bancs). Dans le chapitre "simulation" nous allons voir le comportement de ce taux d'occupation des bancs en fonction de certains paramètres.

2. l'architecture du réseau d'interconnexions. Cette architecture est choisie de manière à minimiser le rapport coût/performance. Par conséquent, le taux de conflits réseau est plus ou moins important suivant le réseau utilisé.

3. le nombre de ports actifs (et/ou processeurs) dans le système. Ce facteur est la cause des conflits de simultanéité. Le nombre de conflits de simultanéité augmente avec le nombre de processeurs.

Soit λ la probabilité que la requête qui arrive dans la file référence un banc libre. Cette probabilité dépend uniquement du nombre de bancs occupés. Elle est égale à (1 - ***probabilité d'occupation de la mémoire***).

On suppose que le temps de cycle (c_m) de la mémoire est un multiple de celui du processeur (c_p): $c_m = Tc_p$. Le facteur T est dit ***temps de latence de la mémoire.***

4.4.3 Variable d'états

Vu de l'extérieur, le système *port↔mémoire* peut être dans deux états possibles: ***actif ou bloqué.*** Il est actif si le port tente un accès à la mémoire, il est bloqué sinon. Si l'état d'un port (j) est $s_j^{t_i}$ à l'instant t_i, l'état du système global est donné par le $(N \times L) - uple$: $(s_1^{t_i}, s_2^{t_i}, \cdots, s_{N\times L}^{t_i})$ des états des ports à l'instant t_i car les ports sont indépendants.

L'état d'un port est donné par l'état de toutes les requêtes dans sa file, donc un port est bloqué si toutes les requêtes dans la file d'attente sont bloquées. Une requête est dite bloquée si elle référence un banc occupé. Un banc mémoire sert une requête pendant T cycles (T est un paramètre entier) et toute autre requête référençant ce banc reste bloquée jusqu'à la fin du service de la précédente. Ainsi on peut définir l'état d'une requête par le nombre de cycles restants r_i pour que le banc mémoire associé soit libre. r_i est compris entre 0 et T. Notons que si $r_i = 0$, la requête est dite ***libre.***

On peut définir complètement l'ensemble des états du système *port↔mémoire* comme $(r_1, r_2, \cdots, r_{K_t})$, où K_t est le nombre de requêtes dans la file à l'instant t. Cependant cet ensemble des états est infini car il d'epend du nombre de requêtes dans la file qui est de taille infinie. Par conséquent l'étude de ce système est complexe.

Un port est actif s'il existe au moins une requête dans sa file qui référence un banc libre, sinon il est bloqué pendant un certain nombre de cycles (i). (i) est le nombre de cycles restants pour qu'au moins une requête de sa file soit libre. Ce nombre (i) est donné par le minimum des états des requêtes dans la file. Par agrégation, cet état sera noté S_i. Par conséquent, le système *port↔mémoire* est représenté par T états bloquants $S_{i,1\leq i\leq T}$ et un état actif S_0, appelé aussi état libre du système. En d'autres termes on ne tient pas compte de l'historique du port. Ainsi on peut écrire:

- Un port est dans l'état $S_i \iff i = min(r_1, r_2, \cdots, r_{K_t})$, où K_t est le nombre de requêtes dans la file à l'instant t.

- Un port est bloqué $\iff \forall i_{(1\leq i\leq K_t)},\ r_i \geq 1$.

- Un port est actif $\iff$ il existe au moins une requête qui référence un banc libre, et le port peut tenter des accès.

4.4.4 Les transitions

Soit q_{ij} la probabilité de transition de l'état S_i à l'état S_j. Elle exprime, donc, la probabilité que le port soit dans l'état S_j sachant qu'il est dans l'état S_i : $Pr[S_j|S_i]$.

a- $S_0 \xrightarrow{q_{0i}} S_i$ pour $(i \geq 1)$: $q_{0i} = Pr[S_i|S_0]$. Nous savons que $\sum_{n=1}^{T} Pr[S_n|S_0] = 1 - p$: le port ne peut pas réaliser un accès, par définition de la probabilité p. On suppose que les T états bloquants sont équiprobables. Comme les requêtes sont aléatoirement distribués sur la mémoire (Hypothèse 5), par conséquent le port peut se trouver dans un état S_i parmi T avec une probabilité $\frac{1}{T}$. D'où alors $Pr[S_i|S_0]$ est égale à la probabilité d'être dans un état $S_i, i \neq 0$ multiplié par la probabilité que le port ne peut pas réaliser un accès : $q_{0i} = \frac{q}{T}$.

b- $S_0 \xrightarrow{q_{00}} S_0$: On sait que : $Pr[S_0|S_0] + \sum_{n=1}^{T} Pr[S_n|S_0] = 1 \Longrightarrow Pr[S_0|S_0] = 1 - \sum_{n=1}^{T} Pr[S_n|S_0]$ $\Longrightarrow Pr[S_0|S_0] = 1 - (1 - p) = p$.

En posant $q = 1 - p$, on a $q_{00} = 1 - q$.

c- $S_i \xrightarrow{q_{ij}} S_j$ pour $(i \neq 0)$: par définition, le port est dans l'état S_i à l'instant t $\Longrightarrow i = \min(r_1, r_2, \cdots, r_{K_t})$. Au cycle suivant (t+1) le port sera dans un état S_j, tel que : $j = \min(r_1 - 1, r_2 - 1, \cdots, r_{K_t} - 1, r_{K_{t+1}})$, ce qui donne :

$$j = \min(i - 1, r_{K_{t+1}}). \tag{4.1}$$

La probabilité q_{ij} est calculée en fonction des cas suivants :

1. **Cas** $j > i$: ce cas ne se produira pas, d'après l'équation (4.1) j est toujours inférieur ou égal à $(i - 1)$; $q_{ij} = 0$.

2. **Cas** $i = 1$ *et* $j = 0$: $q_{10} = 1$ car $j = \min(0, r_{K_{t+1}}) = 0, \forall r_{K_{t+1}}$.

3. **Cas** $i > 1$ *et* $j = 0$: pour que $j = 0$ il faut que $r_{K_{t+1}} = 0$. Cet évènement est réalisé avec une probabilité λ; $q_{i0} = \lambda = 1 - u$.

4. **Cas** $j \neq 0$: $j = \min(i - 1, r_{K_{t+1}}) \Rightarrow \quad j = i - 1$ *ou* $j = r_{K_{t+1}}$. Cherchons les probabilités $Pr[S_{i-1}|S_i]$ et $Pr[S_{r_{K_{t+1}}}|S_i]$.

   On suppose que la requête $r_{K_{t+1}}$ peut référencer un banc mémoire occupé pendant x cycles, $(1 \leq x \leq T)$, de manière aléatoire avec une probabilité $\frac{u}{T}$. D'où $Pr[S_{r_{K_{t+1}}}|S_i] = \frac{u}{T}$, $r_{K_{t+1}} \neq i - 1$.

   $Pr[S_{i-1}|S_i] = Pr[r_{K_{t+1}} \geq i - 1] = u(1 - Pr[r_{K_{t+1}} < i - 1] = u\left(1 - \frac{i-2}{T}\right)$.

Le modèle ainsi défini est bien un processus de Markov. Les transitions des états du système *port↔mémoire* dépendent de l'état courant S_i et de la nouvelle requête arrivée pendant le cycle en cours.

En résumé, les probabilités de transitions sont données par :

$$q_{ij} = \begin{cases} 1-q & \text{si } i = j = 0, \\ \frac{q}{T} & \text{si } i = 0 \text{ et } j > 0, \\ 1 & \text{si } i = 1 \text{ et } j = 0, \\ \lambda & \text{si } i > 0 \text{ et } j = 0, \\ \frac{u}{T} & \text{si } 1 \leq j < i, \\ u\left(1 - \frac{i-2}{T}\right) & \text{si } i = i-1. \end{cases} \tag{4.2}$$

Elles sont aussi illustrées sur le graphe de transitions de la figure 4.5.

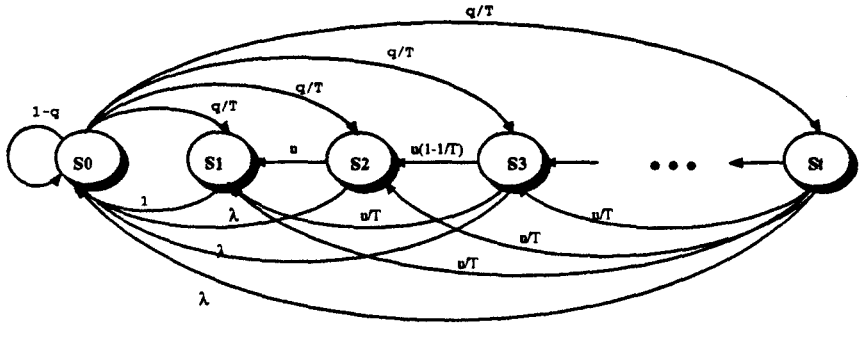


Figure 4.5: Graphe de transition du modèle de traitement désordonné du système *port↔mémoire* .

Soit P le vecteur des probabilités des états $S_0, S_1, \ldots, S_T$.

$P = (P_0, P_1, \ldots, P_T)$.

Lorsque le système est en régime stationnaire on a l'équation d'invariance suivante :

$$PM = P. \tag{4.3}$$

Où M est la matrice des transitions :

$$M = \begin{pmatrix} 1-q & \frac{q}{T} & \frac{q}{T} & \cdots & \frac{q}{T} & \frac{q}{T} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ \lambda & u & 0 & \cdots & 0 & 0 \\ \lambda & \frac{u}{T} & u\left(1-\frac{u}{T}\right) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \lambda & \frac{u}{T} & \frac{u}{T} & \cdots & u\left(1-\frac{T-2}{T}\right) & 0 \end{pmatrix} \tag{4.4}$$

L'équation (4.3) peut aussi s'écrire sous la forme d'un système d'équations :

$$\begin{cases} P_0 &= (1-q)P_0 + P_1 + \lambda \sum_{i=2}^{T} P_i \\ P_1 &= \frac{q}{T}P_0 + uP_2 + \frac{u}{T}\sum_{i=3}^{T} P_i \\ \vdots & \vdots \quad \vdots \\ P_i &= \frac{q}{T}P_0 + u\left(1 - \frac{i-1}{T}\right)P_{i+1} + \frac{u}{T}\sum_{j=i+2}^{T} P_j \\ \vdots & \vdots \quad \vdots \\ P_{T-1} &= \frac{q}{T}P_0 + u\left(1 - \frac{T-2}{T}\right)P_T \\ P_T &= \frac{q}{T}P_0 \end{cases} \tag{4.5}$$

4.4.5 Résolution du système

La résolution de ce système s'effectue en deux étapes :

1. Exprimer $P_{i,(1\leq i\leq T)}$ en fonction de P_0, car le déterminant de la matrice M est nul.
2. Exprimer P_0 en fonction de u, q et T, moyennant la contrainte supplémentaire suivantes : $\sum_{i=0}^{T} P_i = 1$.

4.4.5.1 Expression des $P_{i,(1\leq i\leq T)}$

Dans un premier temps, on a constaté qu'on peut exprimer P_i en fonction de P_0 et de P_j avec $(i < j \leq T)$:

$P_0 = (1-q)P_0 + P_1 + \lambda \sum_{i=2}^{T} P_i \quad \Rightarrow \quad P_1 = qP_0 - \lambda \sum_{i=2}^{T} P_i$

$P_1 = \frac{q}{T}P_0 + uP_2 + \frac{u}{T}\sum_{i=2}^{T} P_i \quad \Rightarrow \quad P_2 = \left(q - \frac{q}{T}\right)P_0 - \left(\frac{u}{T} + \lambda\right)\sum_{i=3}^{T} P_i$

$P_2 = \frac{q}{T}P_0 + u(1-\frac{1}{T})P_3 + \frac{u}{T}\sum_{i=3}^{T} P_i \quad \Rightarrow \quad P_3 = \left(q - \frac{2q}{T}\right)P_0 - \left(\frac{2u}{T} + \lambda\right)\sum_{i=4}^{T} P_i$

On peut montrer par récurrence que :

$$\boxed{P_i = q\left(1 - \frac{i-1}{T}\right)P_0 - \left(\frac{u(i-1)}{T} + \lambda\right)\sum_{j=i+1}^{T} P_j, \quad 1 \leq i \leq T} \tag{4.6}$$

Soient $C_i = u(1 - \frac{i}{T})$, et $S_i = \sum_{j=i}^{T} P_j$. L'équation (4.6) peut s'écrire sous la forme:
$P_i = (C_{i-1} - 1)\,S_{i+1} + \frac{q}{u}P_0 C_{i-1}, \quad 1 \leq i \leq T.$

En substituant P_i dans l'équation $S_i = P_i + S_{i+1}$, on obtient :

$$S_i = C_{i-1}\left(S_{i+1} + \frac{q}{u}P_0\right), \quad 1 \leq i < T. \tag{4.7}$$

En remplaçant $S_{j,(i<j\leq T)}$ de manière récursive en procédant dans l'ordre décroissant des indices, on arrive à l'équation suivante :

$$S_i = S_T m_{i-1} + \frac{q}{u}P_0 n_{i-1}, \quad 1 \leq i < T. \tag{4.8}$$

avec $m_i = \prod_{j=1}^{T-2} C_j$ et $n_i = \sum_{k=i}^{T-2}\prod_{j=i}^{k} C_j$.

Comme $S_T = \frac{q}{T}P_0$, en posant $y_i = \frac{u}{T}m_i + n_i$, $(1 \leq i \leq T-2)$, les équations (4.8) et (4.6) peuvent s'écrire respectivement sous la forme :
$S_i = \frac{q}{u}P_0 y_{i-1}$ et $P_i = \frac{q}{u}P_0\left[(C_{i-1}-1)\,y_i + C_{i-1}\right]$, $(1 \leq i \leq T-2)$.

m_i et n_i ne sont pas définis pour $i = T-1$ et $i = T$, pour étendre l'équation précédente pour $i = T-1$ et $i = T$, on pose : $m_{T-1} = 1$ et $m_T = n_T = n_{T-1} = 0$. Ainsi on peut écrire :

$$\boxed{P_i = \frac{q}{u}P_0\left[(C_{i-1}-1)\,y_i + C_{i-1}\right] \quad 1 \leq i \leq T} \tag{4.9}$$

Cette équation ne dépend que de q, P_0, u et T.

4.4.5.2 Calcul de P_0 en fonction de u, q et T

$$\sum_{i=0}^{T} P_i = 1 = P_0 + \sum_{i=1}^{T} P_i \Rightarrow \quad P_0 = 1 - \sum_{i=1}^{T} P_i = \frac{1}{1 + \frac{q}{u}\sum_{i=1}^{T}\left((C_{i-1}-1)\,y_i + C_{i-1}\right)}$$

Le terme $\sum_{i=1}^{T} C_{i-1} = u\sum_{i=1}^{T}\left(1 - \frac{i-1}{T}\right) = u\left(\frac{T+1}{2}\right)$.

On pose $W = \sum_{i=1}^{T}(C_{i-1}-1)\,y_i$ et $Y = \frac{T+1}{2}$, alors :

$$\boxed{P_0 = \frac{1}{1 + \frac{q}{u}(W + uY)}} \tag{4.10}$$

W est donné par l'expression suivante[1] :

[1] **Calcul de W** :

$$\boxed{W = -\frac{T(T-1)}{2}x + T!\sum_{k=1}^{T-1}\frac{x^{k+1}}{(T-k-1)!} \quad T \geq 1 \text{ avec } x = \frac{u}{T}} \tag{4.11}$$

D'où alors P_0 ne dépend que de u, q et T. Il en est de même pour les probabilités $P_{i,(1\leq i\leq T)}$.

La probabilité p, telle qu'elle est définie, dépend de la proportion de bancs occupés, de la proportion des conflits de réseau d'interconnexions et des conflits de simultanéité.

Soit α le taux de conflits de réseau et de conflits de simultanéité. Sous l'hypothèse 5 (les requêtes sont uniformément distribuées sur la mémoire), α est la probabilité qu'un port est en

---

$$W = \sum_{i=1}^{T} y_i\left(C_{i-1} - 1\right) = \sum_{i=1}^{T-1} y_i\left(C_{i-1} - 1\right), \text{ car } y_T = 0$$

avec $y_i = T_i!\left(x^{T_i} + \sum_{k=1}^{T_i-1}\frac{x^k}{(T_i-k)!}\right)$, et $C_{i-1} = u\left(1 - \frac{i-1}{T}\right) = \frac{u}{T}(T_i+1) = x(T_i+1)$.

$$\begin{aligned}
W &= \sum_{i=1}^{T-1} C_{i-1}y_i - \sum_{i=1}^{T-1} y_i = \sum_{i=1}^{T-1} x(T_i+1)y_i - \sum_{i=1}^{T-1} y_i \\
&= \sum_{i=1}^{T-1}(T_i+1)!\left(x^{T_i+1} + \sum_{k=1}^{T_i-1}\frac{x^{k+1}}{(T_i-k)!}\right) - \sum_{i=1}^{T-1} T_i!\left(x^{T_i} + \sum_{k=1}^{T_i-1}\frac{x^k}{(T_i-k)!}\right) \\
&= \sum_{i=1}^{T-1}(T_i+1)!x^{T_i+1} - \sum_{i=1}^{T-1} T_i!x^{T_i} + \left(\sum_{i=1}^{T-1}(T_i+1)!\sum_{k=1}^{T_i-1}\frac{x^{k+1}}{(T_i-k)!} - \sum_{i=1}^{T-1} T_i!\sum_{k=1}^{T_i-1}\frac{x^k}{(T_i-k)!}\right)
\end{aligned}$$

On a : $\sum_{i=1}^{T-1}(T_i+1)!x^{T_i+1} - \sum_{i=1}^{T-1} T_i!x^{T_i} = \sum_{n=2}^{T} n!x^n - \sum_{n=1}^{T-1} n!x^n = T!x^T - x.$

D'où alors :

$$\begin{aligned}
W &= T!x^T - x + \sum_{i=1}^{T-1}(T_i+1)!\sum_{k=1}^{T_i-1}\frac{x^{k+1}}{(T_i-k)!} - \sum_{i=1}^{T-1} T!\sum_{k=1}^{T_i-1}\frac{x^k}{(T_i-k)!} \\
&= -x + T!x^T + \sum_{i=1}^{T-1}\left((T_i+1)!\sum_{k=1}^{T_i-1}\frac{x^{k+1}}{(T_i-k)!} - T_i!\sum_{k=1}^{T_i-1}\frac{x^k}{(T_i-k)!}\right) \\
&= -x + T!x^T + \sum_{n=2}^{T-1}\left((n+1)!\sum_{k=1}^{n-1}\frac{x^{k+1}}{(n-k)!} - n!\sum_{k=1}^{n-1}\frac{x^k}{(n-k)!}\right) \\
&= -x + T!x^T + \sum_{n=2}^{T-1}\left((n+1)!\sum_{k=1}^{n-1}\frac{x^{k+1}}{(n-k)!} - n!\sum_{k=2}^{n-1}\frac{x^k}{(n-k)!} - nx\right) \\
&= -x + T!x^T + T!\sum_{k=1}^{T-2}\frac{x^{k+1}}{(T-k-1)!} - \sum_{n=2}^{T-1} nx.
\end{aligned}$$

d'où :

$$\boxed{W = -\frac{T(T-1)}{2}x + T!\sum_{n=1}^{T-1}\frac{x^{n+1}}{(T-n-1)!} + T!x^T}$$

conflit de réseau ou en conflit de simultanéité. La probabilité complémentaire de α sera notée $\theta = 1 - \alpha$.

Toujours sous la même hypothèse, le taux de conflits de bancs occupés peut être considéré comme la probabilité qu'une requête, uniformément distribuée sur les bancs, référence un banc occupé. Elle est égale à la probabilité complémentaire de λ; $u = 1 - \lambda$.

Pour réaliser un accès à la mémoire, il faut qu'il ait une requête qui référence un banc libre et que celle-ci ne provoque ni conflit de réseau ni conflit de simultanéité. Cet évènement est réalisé avec la probabilité $p = \theta\lambda$.

$p = \theta\lambda \Rightarrow q = 1 - \theta\lambda = \alpha + \theta u$. Ainsi P_0 devient :

$$\boxed{P_0 = \frac{1}{1 + \left(\theta + \frac{\alpha}{u}\right)(W + uY)}} \tag{4.12}$$

On peut remarquer que P_0^{-1} est un polynôme de variable u de degré T.

4.4.6 Equation des débits

Ce qui nous intéresse, ici, est d'exprimer u, le taux d'occupation des bancs en fonction des paramètres "*contrôlables*" du système qui sont : T, N, L, M et θ. Connaissant u, on peut calculer l'efficacité et la bande passante, ainsi que d'autres fonctions de mesure de performances. Ceci peut se faire en considérant une équation supplémentaire qui est : l'équation des débits en régime stationnaire.

L'équation des débits dit que : ***quand le système est dans la phase stationnaire, le débit des entrées (des processeurs) est égal au débit des sorties (des bancs mémoires).***

- Le débit des entrées est : $NL\theta P_0$; (N et L sont respectivement le nombre de processeurs et le nombre de ports par processeur).
- Le débit des sorties est : $\frac{Mu}{T}$; (M est le nombre de bancs et T est le temps de latence de la mémoire).

Ainsi :

$$\boxed{NL\theta P_0 = \frac{Mu}{T}} \tag{4.13}$$

On suppose que la portion de bancs occupés est constante en régime stationnaire. Quand le système est en équilibre, le débit de tous les ports est égal au débit de la mémoire (*équation des débits*). On a :

$$\boxed{NL\theta P_0 = \tfrac{Mu}{T} \implies u + \left(\tfrac{\alpha}{u} + \theta\right)(W + uY) = \tfrac{LN\theta T}{M}} \tag{4.14}$$

Après résolution de cette équation, on substitue u dans l'équation de P_0; P_0 ne dépend que de L, N, M, θ et T: $P_0 = f(L, N, M, T, \theta)$. L'équation (4.14) est de degré T, il est très difficile de la résoudre quelque soit T.

4.5 Modélisation du traitement vectoriel classique

Le traitement vectoriel classique respecte l'ordre d'émission des composantes. Cet ordre est donné par l'ordre croissant des index des composantes. La composante d'index $(i+1)$, par exemple, ne peut accéder à la mémoire que si la composante (i) a déjà accédé. Une seule composante est envoyée au port à un moment donné. Le port tente un accès à la mémoire, s'il le réalise avec succès, alors il tente une nouvelle composante au cycle suivant sinon il reste bloqué sur cette composante jusqu'à ce qu'il y ait succès. A un moment donné, il y a une seule requête qui est visible pour le port, donc le modèle peut être décrit par un processus de Markov où la file d'attente est de taille $L = 1$ (figure 4.6). Lorsque le port est bloqué il ne peut pas y avoir d'arrivée dans la file. Dès que le port effectue un accès, une nouvelle arrivée vient s'insérer dans la file d'attente. Ce modèle peut être considéré comme un cas particulier du modèle de traitement vectoriel désordonné. La probabilité p sera nommée p' pour le modèle classique et la probabilité λ n'existe pas, cela est du au fait que la file contient au plus une requête à un instant donné.

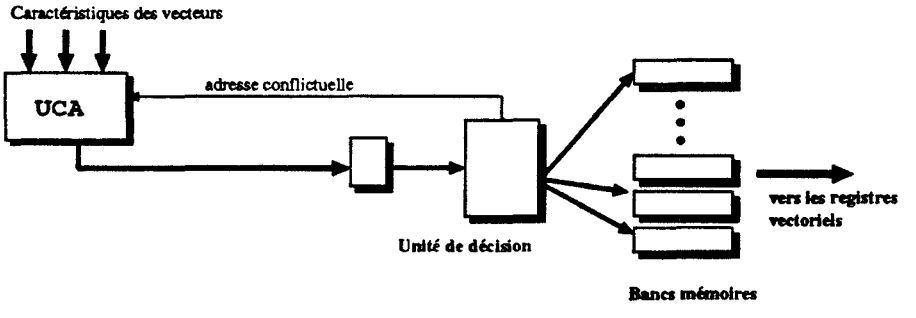


Figure 4.6 : Un système avec une file d'attente de taille un

La définition de l'état du système global est identique et les états du système *port↔mémoire* sont les mêmes que ceux du modèle de traitement vectoriel désordonné. Le port est actif si la requête référence un banc libre, il est bloqué sinon. Le port peut être dans $(T+1)$ états dont T états bloquants. L'état bloquant est donné par le nombre de cycles restants pour que la requête soit libre.

4.5.1 Transitions :

a- $S_0 \xrightarrow{q'_{0i}} S_0$: en étant dans S_0, le port peut tenter un accès avec une probabilité p' de réaliser un succès. $q_{00} = 1 - q'$.

b- $S_0 \xrightarrow{q'_{0i}} S_i$, $(i \geq 1)$: le port a une probabilité $\frac{q'}{T}$ pour qu'elle soit bloquée dans un état quelconque parmi T (pour les mêmes raisons évoquées précédemment). $q'_{0i} = \frac{q'}{T}$.

c- $S_i \xrightarrow{q'_{ij}} S_j$, $(i \neq j+1, i, j \geq 1)$: pas de transitions.

d- $S_i \xrightarrow{q'_{i,i-1}} S_{i-1}$, $(i \geq 1)$: si le port est bloqué pendant (i) cycles, il est certain qu'au cycle

suivant, il n'est bloqué que pendant $(i-1)$ cycles. $q'_{i,i-1} = 1$.

Ces transitions sont représentées sur le graphe de la figure (4.7) et par la matrice suivante:

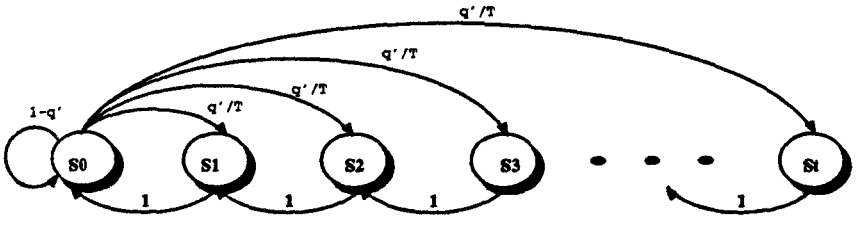


Figure 4.7: graphe de transition du modèle classique

$$M' = \begin{pmatrix} 1-q' & \frac{q'}{T} & \frac{q'}{T} & \cdots & \frac{q'}{T} & \frac{q'}{T} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{4.15}$$

Soit P' le vecteur de probabilités des états du système:

$P' = \left(P'_0, P'_1, \cdots, P'_T\right)$

Lorsque le système est en régime stationnaire, on a: $P'M' = P'$. Ce système matriciel peut s'écrire comme suit:

$$\begin{cases} P'_0 &= p'P'_0 + P'_1 \\ P'_1 &= \frac{q'}{T}P'_0 + P'_2 \\ \vdots & \vdots \\ P'_i &= \frac{q'}{T}P'_0 + P'_{i+1} \\ \vdots & \vdots \\ P'_{T-1} &= \frac{q'}{T}P'_0 + P'_T \\ P'_T &= \frac{q'}{T}P'_0 \end{cases} \tag{4.16}$$

4.5.2 Résolution du système

La résolution de ce système se fera de la même manière que dans le modèle précédent:

1– Expression des $P'_{i,1\leq i\leq T}$ en fonction de P'_0:

$P'_1 = q'P'_0$.

$$P_2' = P_1' - \frac{q'}{T}P_0' = q'\left(1 - \frac{1}{T}\right)P_0'.$$

$$P_3' = P_2' - \frac{q'}{T}P_0' = q'\left(1 - \frac{2}{T}\right)P_0' \quad \text{etc...}$$

On peut vérifier par récurrence que :

$$\boxed{P_i' = \left(1 - \frac{i-1}{T}\right)q'P_0', \quad 1 \leq i \leq T} \tag{4.17}$$

2- Calcul de P_0' en fonction de q' et T :

$$\sum_{i=0}^{T} P_i' = 1 = P_0' + \sum_{i=1}^{T} P_i', \Rightarrow P_0' = 1 - q'P_0' \sum_{i=1}^{T}\left(1 - \frac{i-1}{T}\right) = 1 - q'P_0'\left(\frac{T+1}{2}\right)$$

d'où alors :

$$\boxed{P_0' = \frac{1}{1+q'Y}} \tag{4.18}$$

4.5.3 Equation des débits

Pour les mêmes raisons évoquées précédemment, p' peut s'écrire sous la forme $p' = \theta'\lambda'$. En considérant que λ', taux de bancs libres, est constant lorsque le système est en équilibre (dans l'état stationnaire), l'équation des débits peut s'écrire : $\theta' NLP_0' = \frac{Mu'}{T}$.

La résolution de cette équation en u' donne :

$$u' = \frac{-(1+Y\alpha') + \sqrt{(1+Y\alpha')^2 + \frac{2\theta'^2T(T+1)NL}{M}}}{\theta'(T+1)}$$

En substituant u' dans l'expression de P_0', on obtient :

$$\boxed{P_0' = \frac{2}{1+\alpha'\frac{(T+1)}{2} + \sqrt{(1+\alpha'\frac{T+1}{2})^2 + \frac{2\theta'^2T(T+1)NL}{M}}}} \tag{4.19}$$

$\alpha' et u'$ sont les probabilités complémentaires de $\theta' et \lambda'$ respectivement.

Notons que pour $T = 1$, $P_0' = \frac{1}{1+q'}$, et $P_0 = \frac{1}{1+q}$. Comme pour le traitement vectoriel désordonné ce modèle ne dépend que de l'état du réseau d'interconnexions et des interactions entre les ports et les mémoires.

4.6 Mesure de performances

Le débit des processeurs est certainement maximum lorsqu'ils exécutent des accès vectoriels. Il est d'une requête par port et par cycle. Ce qui nous intéresse ici est d'évaluer la capacité de transfert de données moyenne entre les processeurs et les mémoires en fonction des paramètres

du système. Cette capacité dépend du débit du réseau, la bande passante effective de la mémoire et du comportement des requêtes de chaque processeur.

La capacité d'échange de données entre le port et les mémoires est donnée par le nombre d'accès effectués par le port par unité de temps. Elle est aussi appelée **efficacité du système** ou taux d'accès mémoire.

Pour pouvoir comparer le modèle TVD au modèle classique, et de pouvoir décider sous quelles conditions le modèle TVD est nettement plus rentable. Nous avons défini la fonction **speed-up**. Les deux fonctions (**efficacité** et le **speed-up**) sont définies dans ce qui suit.

4.6.1 Efficacité du système

La capacité de transfert de données est l'ensemble de communications entre les processeurs et la mémoire. Cette capacité est donnée, pour un transfert de données vectorielles, par l'efficacité E_x d'un système multiprocesseur. Il est égal au rapport entre le nombre moyen d'accès et la somme du nombre moyen d'accès et du nombre moyen d'unités de temps passées dans la file d'attente.

Soit N_{ma} (resp. N'_{ma}) le nombre moyen d'accès par unité de temps du modèle de traitement désordonné (resp. du modèle classique), et soit N_{mt} (resp. N'_{mt}) le nombre moyen d'unités de temps passés en attente dans le buffer.

On a : $N_{ma} = pP_0$, et $N'_{ma} = p'P'_0$.
$N_{mt} = 1 - P_0$, et $N'_{mt} = 1 - P'_0$.

L'efficacité pour les deux modèles de traitement E_{tvd} et E_{cl} respectivement pour le modèle de traitement vectoriel désordonné et pour le modèle classique sont :

$$E_{tvd} = \frac{N_{ma}}{N_{ma} + N_{mt}} = \frac{pP_0}{1 - qP_0} = \frac{p}{\frac{1}{P_0} - q} = \frac{p}{1 + \frac{q}{u}W + q(Y-1)} = \frac{(1-u)\theta}{1 + (\frac{\alpha}{u} + \theta)(W + uY + u)}$$

$$E_{cl} = \frac{N'_{ma}}{N'_{ma} + N'_{mt}} = \frac{p'P'_0}{1 - q'P'_0} = \frac{p'}{\frac{1}{P'_0} - q'} = \frac{p'}{1 + q'(Y-1)} = \frac{u'(1-u')\theta'}{1 + (\alpha' + u'\theta')Y}.$$

4.6.2 Le Speed-up du système

Pour pouvoir comparer les temps d'exécution des deux modèles sur une application et une architecture données, nous avons introduit la fonction speed-up qui est définie par:

$$G_{tvd} = \frac{T_{cl} - T_{tvd}}{T_{cl}}$$

où T_{tvd} et T_{cl} sont respectivement les temps d'exécution du code vectoriel d'une application par le modèle de traitement désordonné et par le modèle classique.

Les temps d'exécution sur du code vectoriel d'une application par chacun des modèles sont :

$T_{tvd} = V_l(1 + N_{mt})cp$,

$T_{cl} = V_l(1 + N'_{mt})cp$.

Où V_l est le nombre d'accès effectués.

Le speed-up du modèle de traitement désordonné par rapport au classique est :

$$G_{tvd} = \frac{T_{cl} - T_{tvd}}{T_{cl}} = \frac{N'_{mt} - N_{mt}}{1 + N'_{mt}} = \frac{P_0 - P'_0}{2 - P'_0} = \frac{(q' - q)Y - \frac{q}{u}W}{(1 + \frac{q}{u}(W + uY))(1 + q'Y)}$$

4.6.3 Comparaison et discussion

Afin de valider ce modèle nous l'avons comparé à deux modèles : *classique* et *idéal*. La comparaison avec le modèle classique permet d'évaluer l'amélioration des performances des systèmes multiprocesseurs vectoriels pipelines. Quant à la comparaison avec le modèle idéal, elle permet de fixer certains paramètres architecturaux (comme le nombre de bancs mémoires, nombre de ports, ...) suivant les performances qu'on veut atteindre.

Le modèle idéal est un système qui ne présente aucun conflit d'accès à la mémoire. Dans le cas de l'architecture que nous avons considéré, le réseau est un crossbar complet et le nombre de bancs mémoires doit être infini et aucun banc ne doit contenir deux informations qui peuvent être référencées au même instant. Ce modèle permet de supposer qu'on peut accéder à n'importe quel ensemble de mots mémoire en parallèle. Cela n'est évidemment possible que si chaque banc mémoire est composé d'un seul mot mémoire. Notre but est non pas de voir si le modèle idéal est réalisable ou comment le réaliser, mais de comparer les performances d'un tel modèle avec le nôtre, et ainsi de voir si les performances du modèle TVD sont proches de celles qu'on obtient avec le modèle idéal.

Dans le cas d'un flux d'accès vectoriel, le système idéal possède un seul état S_0 (figure 4.8).

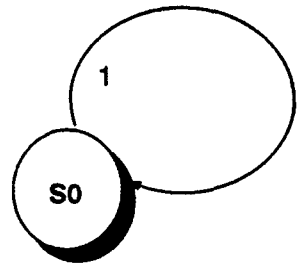


Figure 4.8 : graphe de transition du modèle idéal

Lemme 1 *Par définition du modèle TVD et du modèle classique, de p et p',on a : $p \geq p'$.*

En modèle TVD une requête qui a subit un échec est rangée dans la file. Au cycle suivant, contrairement au modèle classique, le port n'est pas certain qu'il soit bloqué, car il reçoit une nouvelle requête à chaque cycle. A un instant donné le nombre moyen de requêtes susceptibles de réaliser un accès est plus important en TVD, donc on a beaucoup plus de chance de trouver une requête qui effectue un accès avec succès. En modèle classique, on a au plus une requête dans la file, et on a moins de chance de réaliser un succès.

Lemme 2 *Le temps moyen d'attente dans le port pour le modèle classique N'_{mt} est supérieur au temps moyen d'attente dans le port pour le modèle TVD :*

$$N'_{mt} \geq N_{mt}.$$

Preuve :

$N'_{mt} - N_{mt} = (1 - P'_0) - (1 - P_0) = P_0 - P'_0$.

Montrons que $P_0 - P_0' \geq 0$.

On a $P_0 = \dfrac{1}{1+\frac{q}{u}(W+uY)}$ et $P_0' = \dfrac{1}{1+q'Y}$

$P_0 \geq P_0' \quad \Rightarrow \quad q'Y \geq \frac{q}{u}(W+uY)$ (*). Or $q' \geq q \quad \Rightarrow \quad q'Y \geq qY$.

Pour montrer l'inégalité (*) il suffit de montrer que : $qY \geq \frac{q}{u}W + qY$ (car $q < q'$), $\Rightarrow \frac{W}{u} \leq 0$.

$$\frac{W}{u} = -\frac{(t-1)}{2} + (t-1)! \sum_{k=1}^{T-1} \frac{u^k}{T^k(T-k-1)!}.$$

$$(T-1)! \sum_{k=1}^{T-1} \frac{u^k}{T^k(T-k-1)!} \leq \sum_{k=1}^{T-1} \frac{(T-1)!}{T^k(T-k-1)!} \leq \sum_{k=1}^{T-1} \frac{(T-2)^k}{T^k} = \sum_{k=1}^{T-1} \left(1-\frac{2}{T}\right)^k \quad (**)$$

(**) est la somme des $(T-1)$ premiers termes d'une suite géométrique de raison $\left(1-\dfrac{2}{T}\right)$ et de premier terme $\left(1-\dfrac{2}{T}\right)$. D'où :

$$\sum_{k=1}^{T-1} \left(1-\frac{2}{T}\right)^k = \frac{(T-2)\left(1-\left(1-\frac{2}{T}\right)^{T-1}\right)}{2} \leq \frac{T-1}{2}$$

Par conséquent $\dfrac{W}{u} \leq 0$. D'où le résultat : $P_0 \geq P_0'$.

Proposition 2 *Pour un système multiprocesseur donné, on a toujours :*

$-1-$ $E_{tvd} > E_{cl}$

$-2-$ $G_{tvd} \geq 0$

Preuve :

1. $E_{tvd} = \dfrac{p}{\frac{1}{P_0} - q} = \dfrac{p}{1+\frac{q}{u}W + q(Y-1)}$ et $E_{cl} = \dfrac{p'}{\frac{1}{P_0'} - q'} = \dfrac{p'}{1+q'(Y-1)}$

   On a : $1 + \dfrac{q}{u}W + q(Y-1) \leq 1 + q(Y-1) \leq 1 + q'(Y-1)$

   $\Rightarrow \quad \dfrac{1}{1+\frac{q}{u}W + q(Y-1)} \geq \dfrac{1}{1+q'(Y-1)}$

   $\Rightarrow \quad \dfrac{p}{1+\frac{q}{u}W + q(Y-1)} \geq \dfrac{p'}{1+q'(Y-1)}$

   $\Rightarrow$ $E_{tvd} \geq E_{cl}$, quelque soit la variations des paramètres N, M, T, L et θ.

2. La preuve de (2) est une conséquence directe du lemme 1.

Nous venons de montrer que le modèle désordonné est plus performant que le modèle classique lorsque le système est en régime stationnaire et en supposant dès le départ que les requêtes sont uniformément distribuées sur l'ensemble des bancs mémoires.

Les deux probabilités p et p' respectivement du modèle désordonné et du modèle classique (et par conséquent les performances du système) dépendent de la répartition des requêtes sur les bancs. Dans la réalité cette répartition n'est pas vraiment uniforme, comme c'est le cas des

vecteurs contigus ou des vecteurs définis par pas régulier qui représentent souvent une partie importante des accès en vectoriel dans un programme. Ce paramètre sera considéré dans les simulations.

Les hypothèse considérées ici simplifient l'étude et la construction de ces modèles. Ce qui conduit à une large estimation des performances du fonctionnement du modèle de traitement vectoriel désordonné et du modèle classique. Les modèles étudient explicitement les conflits de bancs occupés qui sont les plus pénalisant dans ce type d'architectures à cause des temps de latence de la mémoire élevés. Cette approche ne modélise pas de manière exacte les conflits de simultanéité et les conflits de réseau qui dépendent des interactions entre les processeurs et les ports d'un même processeur respectivement. Dans ce type d'architecture les conflits de simultanéité n'apparaissent pas entre les ports d'un même processeur et les conflits de réseau n'apparaissent pas entre les ports de deux processeurs distincts. Il serait intéressant de modéliser cette architecture en modifiant l'hypothèse d'indépendance totale des ports en considérant un groupe de ports appartenant à un même processeur.

Les caractéristiques du réseau d'interconnexions ne sont pas considérées. Dans le cas du TVD, les performances dépendent aussi de la stratégie de sélection mise en place. Cette stratégie permet de choisir les requêtes qui peuvent accéder à la mémoire sans conflits. Pour cela, on a simulé le système et on a pris en compte tous les paramètres dont dépendent les performances du modèle. Le chapitre suivant présente la simulation des deux modèles, d'une part afin de valider cette étude théorique et de la completer par simulation du modèle de fonctionnement exact.

4.7 Validation du modèle

La construction du modèle analytique nous permet de connaître la performance approximative a laquelle on peut s'attendre pour des valeurs données de certains paramètres architecturaux de la machine. les deux modèles que nous avons construit ne sont qu'approximatifs, ils ne reproduisent pas le comportement exacte de la machine multiprocesseur TVD ou de la machine classique. Les hypothèses simplificatrices sont de deux types : celles qui permettent la modélisation du système *port↔mémoire* telles que indépendance totale des ports, supposition que les requêtes sont uniformment distribuées sur la mémoire et surtout, l'hypothèse qui nous permet de passer d'un système à un nombre d'états infini à un système à un nombre d'état fini T, et celles qui facilitent la définition des probabilités et la résolution des équations qui en découlent. On peut citer l'hypothèse d'équiprobabilité des états bloquants, la résolution du système à l'état stationnaire etc. Pour cela les modèles ont besoin d'être valider. Validation et confortation des résultats théoriques sont effectuées à l'aide de la simulation du modèle exact (c'est à dire le modèle réel). Les résultats pratiques sont obtenus à l'aide de la méthode de simulation traitée dans le chapitre suivant.

Les résultats théoriques sont conformes aux résultats simulés. La différence de variation entre les deux courbes représentant l'efficacité du TVD simulée et théorique des figures (4.9, 4.10, 4.11) est due aux choix de θ qui est de 90% pour les courbes théoriques et aux effets des phases de transition (la taille des vecteurs est de 64 éléments) puisque le modèle analytique traite uniquement le comportement du système en phase stationnaire.

On peut conclure que l'allure des courbes des résultats de simulation coïncide avec les courbes théoriques. L'écart entre les deux courbes n'est pas très significatif. Le modèle analyitque est acceptable.

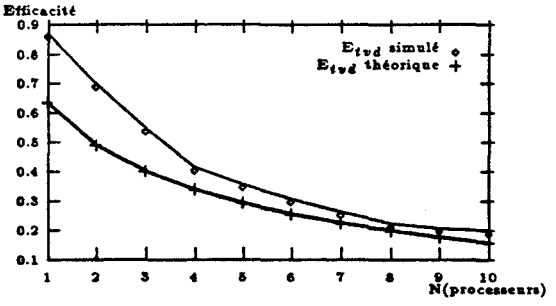


Figure 4.9 : Courbes de variation de l'efficacité en fonction du nombre de processeurs.

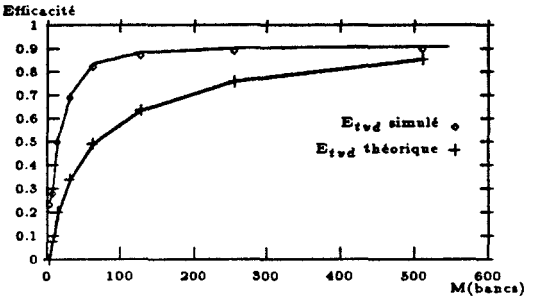


Figure 4.10 : Courbes de variation de l'efficacité en fonction du nombre de bancs.

4.8 Conclusion

Le problème de modélisation et d'évaluation de performances a deux objectifs :

1. Trouver un modèle qui exprime le comportement, dans le temps, du système multiprocesseur de manière exacte, en tenant compte de tous les paramètres du système.
2. Le modèle doit être simple, pour permettre d'en estimer facilement les performances et de pouvoir le comparer aux modèles existants.

Ces deux objectifs sont en contradiction. Le premier but complique le modèle et le rend très difficile à analyser. Concernant le deuxième but, le modèle n'est qu'approximatif et il n'exprime pas tous les paramètres du système et toutes les contraintes qui agissent sur ce système. Une autre étude complémentaire est nécessaire pour valider le modèle.

Nos modèles sont simples. Ils sont réduits à l'analyse du comportement du système sur du code vectoriel. Les modèles sont simulés afin d'estimer les performances sous certains paramètres exprimés de manière implicite ou qui ne sont pas considérés dans le modèle analytique.

Certains résultats, déjà démontrés en utilisant des modèles simples, sont vérifiés dans les deux modèles analytiques :

A efficacité constante,

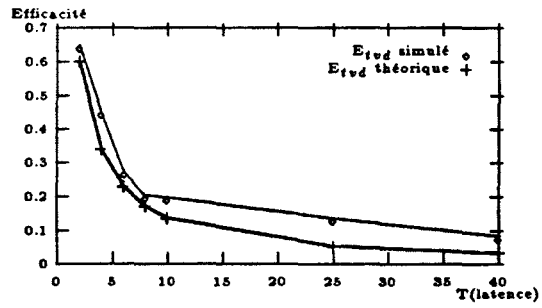


Figure 4.11 : Courbes de variation de l'efficacité en fonction du temps de latence de la mémoire.

- les bancs mémoires et les processeurs augmentent d'un même facteur k.
- le temps de latence de la mémoire augmente d'un facteur k^2 si le nombre de bancs mémoire augmente d'un facteur k.

Chapitre 5

Simulation du modèle de traitement vectoriel désordonné

Ce chapitre présente l'ensemble des résultats de simulations du modèle de fonctionnement du traitement vectoriel désordonné. L'analyse de ces résultats et les résultats théoriques permet d'une part de valider et de conforter le modèle analytique et d'autre part elle permet une évaluation des performances du modèle TVD comparé aux modèles classique et idéal.

Nous commençons par exposer le comportement du modèle en fonction des contraintes imposées lors de sa construction (hypothèse de définition ou de simplification pour de la résolution des systèmes d'équations). Nous verrons, par la suite, l'influence des paramètres architecturaux sur le modèle TVD et notamment la validation du modèle théorique. Pour finir nous comparerons le TVD aux deux modèles classique et idéal par l'intermédiaire de la fonction speed-up pour les différentes variations des paramètres. Ceci nous permettra d'estimer le gain en performance et de trouver une configuration d'une machine (les valeurs des paramètres) performante et moins coûteuse (en hardware).

5.1 Introduction

Comme nous venons de le voir le modèle analytique est construit sous certaines hypothèses qui ne reflètent pas exactement le fonctionnement réel de la machine. Ces hypothèses sont celles qui ne sont pas faciles à exprimer de manière formelle (comportement des programmes ou répartition des données en mémoire) et celles qui permettent de faciliter la résolution des équations (indépendance des ports et la supposition que le système est en régime stationnaire). Dans ce cas précis ces hypothèses sont :

- comportement des requêtes mémoires. La répartition des requêtes en mémoire n'est pas un problème formel. Nous avons supposé que les requêtes sont uniformément distribuées sur l'ensemble de l'espace mémoire. Cette hypothèse permet aussi de simplifier le modèle.
- Indépendance et dépendance des ports. Nous avons supposé que les ports sont complètement indépendants, ce qui n'est pas toujours de cas. Supposons l'exécution de l'expression $A = B + C$, où A, B et C sont des vecteurs en mémoire. Les opérations de chargement de B et C, l'opération "+" et de rangement de A en mémoire sont totalement chaînées. Donc on peut commencer le rangement de A avant la terminaison des deux chargements. Or l'arrivée des composantes du vecteur résultat dépend des évènements précédents déclenchés par les accès des deux vecteurs sources (A et B). Par conséquent l'écriture du résultat en mémoire est lié aux deux flux de lecture.
- Fonctionnement en régime stationnaire : Cette hypothèse permet de se ramener à la résolution d'un système d'équations linéaires. Les expressions de l'efficacité et du speed-up ne sont valides qu'en régime stationnaire. Notre analyse se portera sur l'effet des phases de transitions, sur l'efficacité et le speed-up, ainsi que leur importance dans le fonctionnement du modèle en fonction de la taille des vecteurs.

Dans ce qui suit nous commencerons par présenter le comportement des accès mémoires sous le modèle de traitement vectoriel désordonné, puis nous verrons les paramètres architecturaux influençant ce modèle qui sont de deux types : *les paramètres secondaires* qui ne sont pas pris en considération dans le modèle analytique, et *les paramètres primaires* (pris en compte par le modèle) qui permettent de valider les résultats théoriques.

5.2 Comportement du traitement vectoriel désordonné

Un système de files d'attentes présente deux modes de fonctionnement : le régime transitoire et le régime stationnaire (c'est-à-dire lorsque le système est en équilibre). L'étude du système se ramène aux équations qui décrivent son état d'équilibre parce que souvent la phase de transition est négligeable devant la phase stationnaire.

En accès vectoriel l'importance de l'une ou l'autre des deux phases dépend de la taille des vecteurs à traiter. Plus la taille des vecteurs est grande plus la phase stationnaire devient importante, la phase de transition peut alors être négligée. Cependant la phase de transition n'est, à première vue, pas négligeable dans les machines vectorielles registre-à-registre, car les registres vectoriels sont de taille limitée.

En traitement vectoriel désordonné, le comportement de la répartition des lignes et des bancs entre plusieurs flux de données est représenté par trois phases : phase de début, la phase stationnaire, et la phase de fin. Ces trois phases sont illustrées dans la figure (5.1).

1. **Phase de début (transitoire)**

   C'est la phase où les flux d'accès sont activés et les premiers conflits apparaissent. Durant cette phase les accès mémoires ne peuvent pas être optimisés. Les requêtes qui causent des conflits d'accès resteront dans le buffer en formant une file d'attente. La résolution des conflits consiste simplement à appliquer les systèmes de priorité sur les sections et sur les bancs mémoires. Les requêtes bloquées impliquent le blocage des ports respectifs, par conséquent l'efficacité et le speed-up sont faibles.

2. **Phase stationnaire**

   Cette phase est caractérisée par la résolution des conflits en proposant à chaque port une requête parmi celles contenues dans sa file de façon à ce que les autres ports ne soient pas bloqués. Le blocage d'une requête n'entraîne pas forcément le blocage du port correspondant. Le gain en performance est réalisé pendant cette phase. Elle se termine dès que l'un des flux d'accès est achevé, ainsi le débit effectif des processeurs diminue. Sa durée de vie dépend de la taille des vecteurs traités. Elle caractérise l'accès vectoriel désordonné.

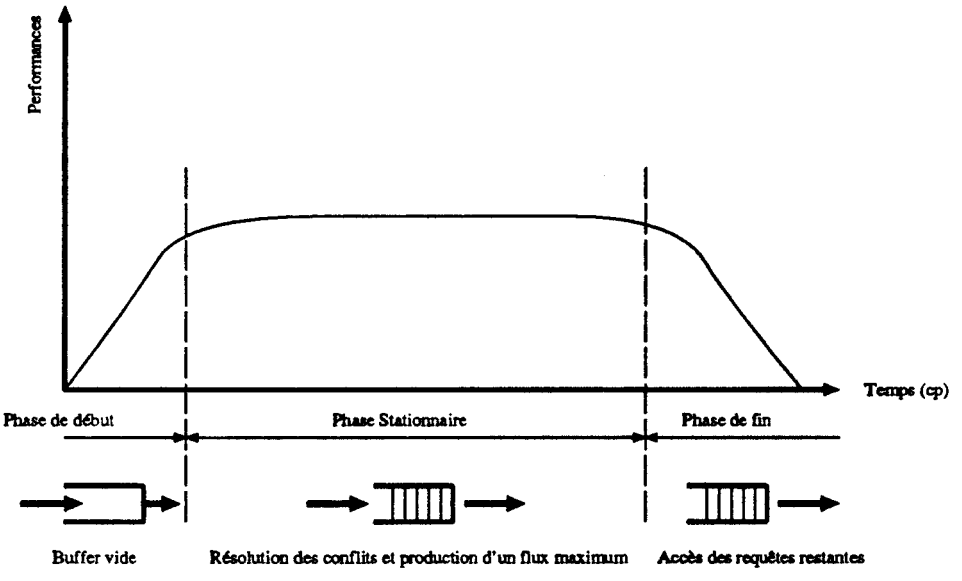


Figure 5.1 : Phases de fonctionnement en traitement vectoriel désordonné

3. **Phase de fin (transitoire)**

   Elle s'étale de la fin de la phase stationnaire jusqu'à la fin de tous les accès. Le nombre de flux actifs diminue de plus en plus, ce qui implique aussi la diminution du taux d'accès mémoires. Les performances se dégradent car le choix des requêtes sans conflit devient de moins en moins possible et le débit des processeurs est faible (un seul flux au lieu de trois). La durée de cette phase est en fonction de la dépendance des ports et du système de priorité appliqué sur les ports et les processeurs.

Naturellement les performances d'un système sont évaluées pour les trois phases. Cependant si la durée des deux phases de transition peut être négligées devant la phase stationnaire, l'estimation des performances quand le système est en équilibre est suffisante.

5.3 Paramètres architecturaux, influence sur le TVD

Nous distinguons deux types de simulations:

- les variations des paramètres primaires qui sont explicites dans le modèle analytique et ont permis sa validation (voir le chapitre 4),
- les variations des paramètres secondaires qui ne sont pas considérés dans le modèle analytique et permettent d'analyser le fonctionnement du modèle TVD et aussi de le comparer aux modèles classique et idéal.

D'après les résultats du chapitre 4, on peut affirmer que le modèle de traitement vectoriel désordonné est nettement meilleur que le modèle classique. Notre but ici est de donner l'ordre de grandeur ou de variation du gain (speed-up) du modèle TVD par rapport au modèle classique. Nous l'avons aussi comparé au modèle idéal afin de fixer l'échelle des paramètres architecturaux tout en gardant un gain et une efficacité plus proche de l'idéal. Ces résultats sont présentés dans [Dekeyser et al.92a], .

5.3.1 Description du simulateur

Le simulateur réalise deux types d'opérations: les opérations du types $C = A\ op\ B$ et les opérations d'accès mémoires sur des vecteurs indépendants (chargement, déchargement des vecteurs opérandes). Le premier type d'opérations permet de mettre en valeur le gain (réalisé lors des flux d'accès) et la conservation du gain (réalisé lors du chaînage des pipelines de calcul avec les opérations d'accès). Le deuxième type d'opérations permet de valider le modèle analytique qui considère que les ports sont indépendants.

Le simulateur implémente les différentes unités de la machine TVD et de la machine classique nécessaires à l'exécution des deux types d'opérations. L'exécution d'une instruction est simulée dans chaque unité. A chaque cycle toutes les unités sont mises à jour, car la machine fonctionne en mode synchrone. Il implémente aussi un certain nombre de techniques de sélection des requêtes d'accès. Le modèle simulé est montré dans le figure (5.2).

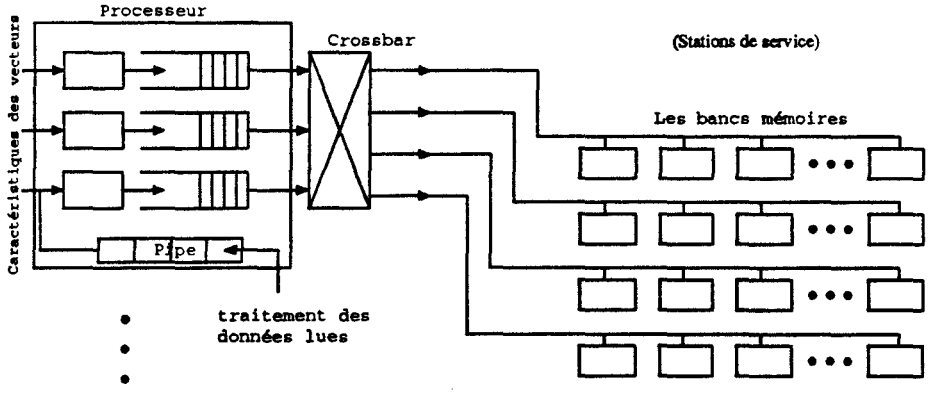


Figure 5.2: Représentation schématique du simulateur.

Le simulateur reçoit en entrée les caractéristiques des vecteurs (l'adresse de base, le type d'accès, le pas d'accès ou le vecteur d'index, et la taille du vecteur) et le type de l'expérience synthétisés par la variation des paramètres suivants:

- Le nombre de ports et/ou de processeurs,
- Le nombre de bancs mémoire,
- Le nombre de sections,
- La taille des vecteurs,
- Le temps de latence de la mémoire,
- Le temps de latence des pipelines,
- Le type des accès (pas réguliers ou dispersés),
- La technique de sélection.

Il délivre en sortie les performances simulées et théoriques de chaque modèle (l'efficacité et le speed-up), et quelques statistiques; comme le taux de bancs occupés, le nombre moyen de tentatives d'accès, le nombre moyen de chaque type de conflits, et la trace d'exécution pour un exemple donné. Notons que nous effectuons une simulation de 45000 opérations pour chaque valeur de variation d'un paramètre.

5.3.1.1 Méthode de validation des simulations

Les résultats de simultation ne sont pas valeurs exactes des paramètres ou de performances de la machine sous les mêmes conditions de simulation. La réponse à la question : "Etant donnée une configuration de la machine, quelle est la performance moyenne de la machine en mode vectoriel?" n'est pas un nombre unique, qui est la valeur moyenne $\overline{x}$ obtenue par simulation du modèle, mais par un intervalle de centre $\overline{x}$ et de demi-longueur Δx appelé *"intervalle de confiance"*. La moyenne $\overline{x}$ est une valeur statistique qui diffère aléatoirement de la valeur exacte e_{ex}. L'intervalle $\overline{x} \pm \Delta x$ contient donc e_{ex} avec une probabilité $p(\Delta x)$ à définir.

Pour une configuration donnée (paramètres fixés) le simulateur nous fournit une valeur moyenne $\overline{x}$ de la variable performance (efficacité ou speed-up). L'intervallle de confiance est donné par : $\overline{x} \pm a\sigma$. σ étant l'écart-type des simulations et a vérifie l'inégalité de Bienaymé-Tchebichef : $Pr[|\overline{x} - e_{ex}|] > a\sigma] < \frac{1}{a^2}$, car on n'a fait aucune hypothèse sur la loi de probabilité de $\overline{x}$. $\overline{x} \pm a\sigma$ est l'intervalle de confiance à $1 - \frac{1}{a^2}$.

L'écart type moyen de simulation de chacune des variations des paramètres est $\sigma = 1,3\%$. L'intervalle de confiance à 95% est $\overline{x} \pm 0,058$.

5.3.2 Les paramètres primaires

Les performances de la machine TVD dépendent des paramètres primaires qui sont : 1) le nombre de processeurs utilisés, 2) le nombre de bancs mémoires, 3) le temps de latence de la mémoire, 4) le nombre de sections dans le réseau d'interconnexions, et d'un paramètre de dépendance des ports qui est le temps de latence des pipelines.

Les trois premiers paramètres sont explicites dans le modèle analytique. Le quatrième paramètre intervient dans le calcul de la probabilité θ. Logiquement lorsqu'on augmente le nombre de sections, les conflits de réseau (sections) diminuent. Le dernier paramètre est secondaire, nous le traiterons plus loin.

Durant la variation d'un paramètre les autres restent fixes. Les valeurs par défaut de tous les paramètres sont :

- Le nombre de processeurs est $N = 2$, et le nombre de ports est constant $L = 3$.
- Le nombre de bancs est $M = 32$.
- Le nombre de sections est $S = 4$.
- Le temps de latence de la mémoire est $T = 4$.
- Le temps de latence des pipelines est $T_p = 4$.
- Les caractéristiques des vecteurs sont générées de manière aléatoire.
- La taille des vecteurs est VL = 64.
- La sélection des requêtes se fait par la technique de scrutation par zone.

5.3.2.1 Nombre de ports et de processeurs

Comme nous l'avons présenté la variation du nombre de ports et la variation du nombre de processeurs sont les mêmes en terme de débit pour la machine TVD. La variation du nombre de processeurs est une autre forme d'augmentation du nombre de ports. Simplement la variation du nombre de ports au sein d'un même processeur suppose que nous disposons d'un nombre suffisant de sections afin que tous les ports actifs ne soient pas bloqués par défaut de sections. L'augmentation du nombre de ports est influencée par les conflits de sections et la variation du nombre de processeurs est affectée par les conflits de simultanéité. Comme la durée de chacun des conflits est la même (un cycle), leurs effets sur les performances sont sensiblement les mêmes (voir la figure (5.3)).

Evidemment, en terme d'implémentation, il est beaucoup plus complexe de gérer un grand nombre de ports d'un même processeur, car en plus de la technique de sélection qui doit être simple et efficace, il faut les alimenter par des flux d'accès. La machine Hitachi S-810/20 possède 4 ports mémoires pouvant effectuer 4 accès en lecture ou 3 lectures et une écriture en concurrence, mais dans des cas réels, il est difficile de produire du code qui occupe tous les pipelines et les ports de cette machine [Lubeck et al.85a, Lubeck et al.85b].

En outre on a montré qu'à efficacité constante le nombre de ports doit être proportionnel au nombre de bancs [Tang et al.89]. Le réseau d'interconnexions n'est pas toujours un crossbar complet, on a montré que lorsque le nombre de ports est supérieur à 3, la chute de performances des machines Cray X-MP, le Y-MP et Ardent TITAN est supérieure à 30% [Calahan et al.88] [Oed et al.86] [Tang et al.89].

La figure (5.3) montre que le taux de bancs occupés est proportionnel au nombre de ports ou de processeurs. Il est évident que l'efficacité diminue lorsque le nombre de ports ou de processeurs augmente.

La chute de l'efficacité s'explique par deux phénomènes. D'un côté le débit de la mémoire est limité par le nombre de bancs et leur temps de latence (un débit de 8 données par cycle processeur), et d'un autre côté les conflits d'accès. Pour garder l'efficacité constante, il faut augmenter le nombre de processeurs et le nombre de bancs d'un même facteur k. Ceci est vérifié de manière théorique par nos modèles théoriques et aussi par de modèle de fonctionnement simulé. Les résultats de simulation sont montrés en figure (5.4).

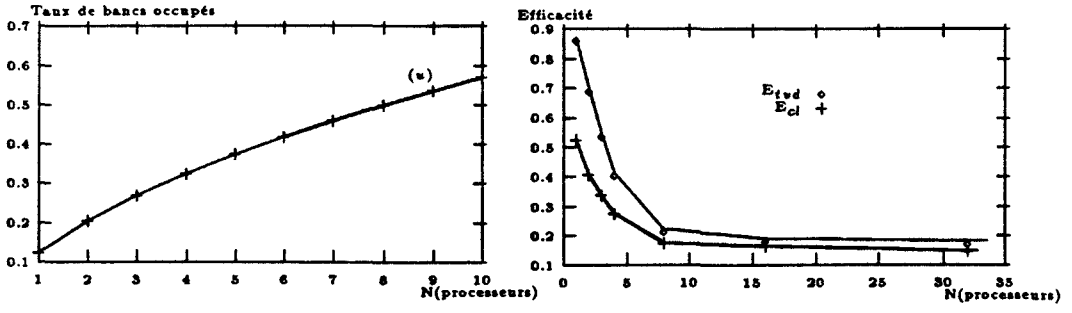


Figure 5.3 : Le tracé de gauche montre la variation de u en fonction du nombre de processeurs et le tracé de droite montre la variation de E_{tvd} en fonctions du nombre de processeurs en modèle simulé.

N (processeurs)	M (bancs)	facteur	E_{tvd}	E_{cl}
2	16	1	0,491767	0,303938
4	32	2	0,492055	0,304101
8	64	4	0,493716	0,303710
16	128	8	0,492373	0,304099
32	256	16	0,489718	0,307976

Figure 5.4 : Résultats de l'augmentation de N et M d'un même facteur k.

5.3.2.2 Nombre de bancs mémoires

La variation du nombre de bancs mémoires permet d'affirmer les résultats suivants :

- De très bonnes performances sont obtenues pour un nombre assez petit de bancs mémoires. L'efficacité est de 85% sur une mémoire de 64 bancs mémoires, et elle n'est que 50% pour le modèle classique (figure (5.5)). Notons que les requêtes sont générées par une fonction aléatoire du simulateur.
- L'efficacité du modèle classique commence à se stabiliser autour de 60%. Ce qui veut dire que les conflits de bancs occupés sont très pénalisants, car le blocage d'une requête en conflit de bancs occupé entraîne un retard sur toutes les autres requêtes qui suivent. Par conséquent l'augmentation du nombre de bancs mémoires n'est pas suffisant pour obtenir une efficacité qui avoisine 90%.

Lorsque le nombre de bancs est très réduit, la mémoire est vite saturée et les conflits de bancs occupés dominent le reste de tentatives d'accès (figure 5.5), d'où l'allure de la courbe.

L'augmentation du nombre de bancs est une solution pour diminuer les conflits de bancs occupés. Lors de la variation du nombre de bancs dans l'intervalle [4, 32] le speed-up croît, car le modèle classique n'a pas encore atteint son état d'équilibre. Le nombre de conflits de bancs occupés est encore important, et ils sont résolus dans le cas du TVD. A partir de 32 bancs le modèle classique rentre dans sa phase stationnaire. L'état stationnaire est atteint lorsque la bande passante effective de la mémoire devient suffisante pour satisfaire le débit des processeurs. Dans ce cas, la bande passante de la mémoire de 32 bancs est de 8 requêtes

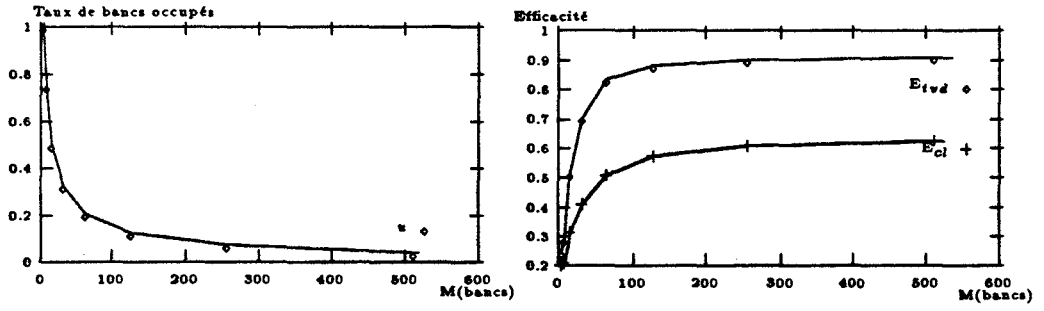


Figure 5.5: Le tracé de la figure gauche montre la variation de du taux u en fonction de M, et la figure de droite présente les deux efficacités E_{tvd} et E_{cl} en modèle simulé.

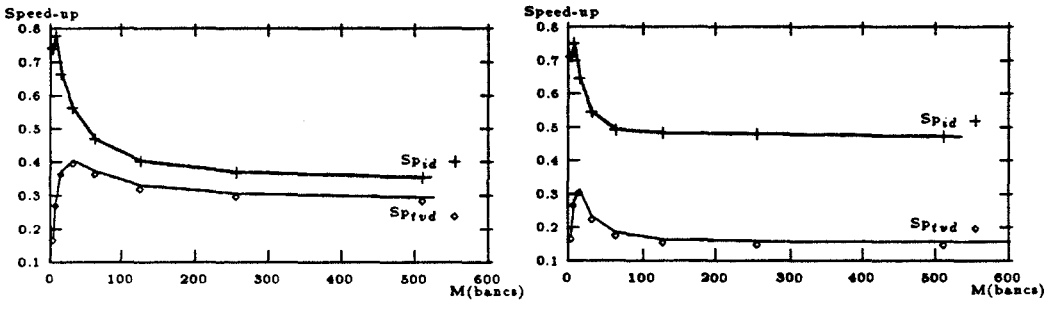


Figure 5.6: Le speed-up en fonction du nombre de bancs. La courbe de gauche est obtenue par simulation du modèle sans dépendance des ports. La courbe de droite est obtenue par simulation des opérations du type $C = A\ op\ B$; avec dépendance du port d'écriture de ceux des lectures.

par cycle processeur (le temps de latence est $T = 4$) et le débit effectif des processeurs est de 6 requêtes par cycle. Ce qui explique l'augmentation de son efficacité et la diminution du speed-up idéal et TVD (voir la figure (5.6)).

5.3.2.3 Le nombre de sections

Afin d'augmenter le champ de variation du nombre de sections, le nombre de bancs mémoires est fixé à 64. Les autres paramètres garderont leurs valeurs par défaut.

Les conflits de sections dépendent du nombre de sections N_s et du comportement des requêtes à accéder à la mémoire. Ils sont représentés, y compris les conflits de simultanéité, par la probabilité θ. Comme une requête mémoire ne peut pas causer un conflit de section et un conflit de simultanéité au même instant, les probabilités θ_{sec} et θ_{sim} d'avoir un conflit de section ou de simultanéité respectivement sont indépendantes. Le nombre de sections influe beaucoup sur les conflits de sections. Cependant il n'est pas explicite dans le modèle analytique, donc, il n'est pas contrôlable théoriquement.

La figure (5.7) montre que les conflits de sections ne sont pas très pénalisant lorsque le nombre

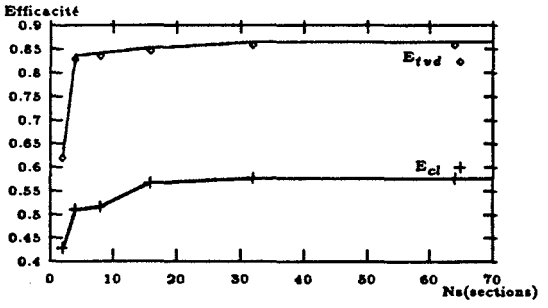


Figure 5.7: Tracé de l'efficacité en fonction du nombre de sections mémoires obtenu par simulation des deux modèles avec indépendance des ports.

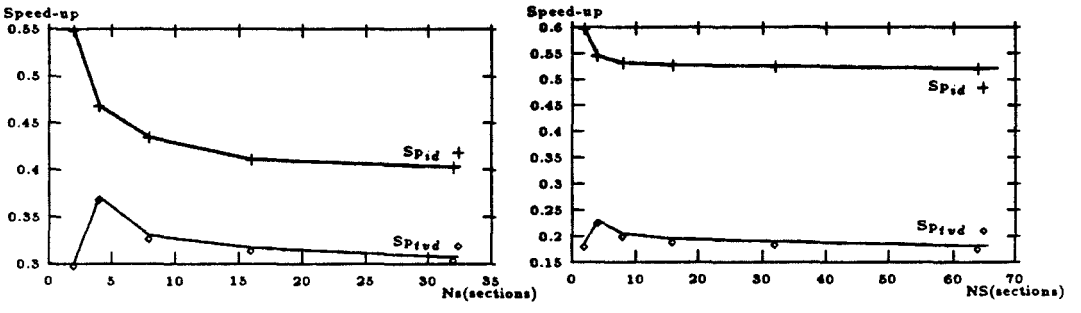


Figure 5.8 : Le speed-up en fonction du nombre de sections : les courbes de gauche sont obtenues sans dépendance des ports et les courbes de droite donnent le speed-up avec dépendance des ports.

de sections est "bien" choisi, c'est-à-dire il est supérieur au nombre de ports. En choisissant Ns= 2, les performances s'effondrent (63% pour le TVD et 43% pour le modèle classique au lieu de 86% et 58% respectivement), car le nombre de ports réellement en accès à chaque instant est 2 (il y a au minimum un conflit de section par cycle). L'efficacité est sensiblement la même dans l'intervalle [4, 64]. Ceci est aussi vérifié sur la fonction speed-up (figure 5.8).

En modèle classique les retards dus aux conflits de sections peuvent induire des conflits de bancs occupés. Ils sont appelés ''conflits liés, en anglais: (*linked conflicts*) [Cheung et al.84, Cheung et al.86]. Ils ont été modélisés par [Oed et al.85], [Oed et al.86]. Sur le Cray X-MP ces conflits sont périodiques lorsque le temps de latence de la mémoire est égal au nombre de sections. Deux solutions ont été proposées pour les éviter; l'augmentation du nombre de sections ou l'augmentation du temps de latence de la mémoire. En modèle de traitement vectoriel désordonné ceci ne risque pas de se produire par définition même du modèle.

5.3.2.4 Le temps de latence de la mémoire

La bande passante effective d'une mémoire de M bancs et de temps de latence T est $B_{ef} = \frac{M}{T}$, ainsi le nombre de requêtes que peut satisfaire à chaque cycle diminue avec l'augmentation

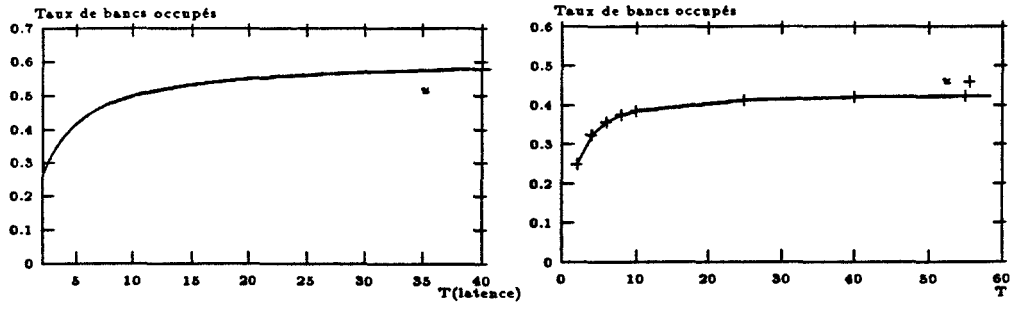


Figure 5.9 : Variation du temps de latence de la mémoire pour les deux modèles. Courbes de gauche : modèle classique. Courbes de droite : modèle TVD.

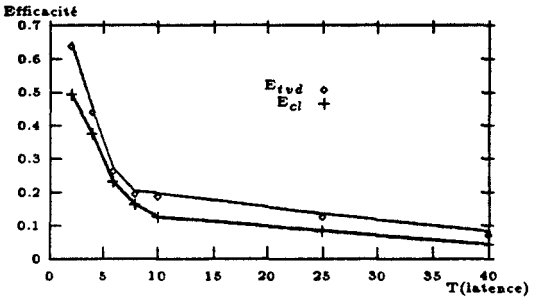


Figure 5.10 : Variation du temps de latence de la mémoire simulé

de T. En d'autre termes les conflits de bancs occupés augmentent avec T, d'où l'allure des courbes du taux de bancs occupés de la figure (5.9). A partir d'une certaine valeur de T la majorité des requêtes bloquées sont en conflits de bancs occupés, par conséquent les ports mémoires resteront bloqués jusqu'à la libération des bancs, d'où la dégradation de l'efficacité (figure (5.10)).

Le speed-up décroît lorsque le temps de latence T augmente. Le taux d'occupation de la mémoire est très élevé, les requêtes sont bloquées sur des bancs occupés qui ne sont résolus par aucun des deux modèles voir la figure (5.11). Ceci n'influe pas sur le modèle idéal car sa mémoire est considérée infinie. Pour conserver le gain il faut que la bande passante effective de la mémoire supporte le débit des processeurs. Ce qui veut dire que l'augmentation de T entraîne l'augmentation du nombre de bancs.

5.3.3 Paramètres secondaires

Les paramètres secondaires que nous avons considérés sont :

- Les types d'accès : nous informent sur le comportement de génération des requêtes et leur répartition sur les bancs mémoires.

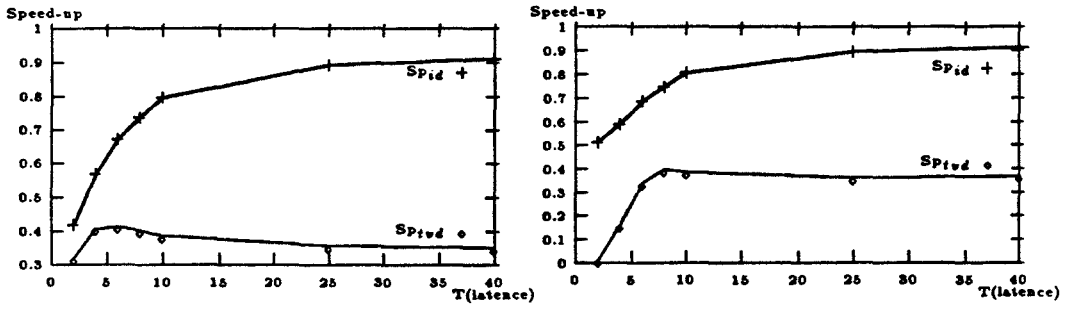


Figure 5.11 : Le speed-up en fonction du temps de latence T. Courbes de gauche : simulation sans dépendance. Courbes de droite : simulation avec dépendance.

- La taille des registres vectoriels qui influe sur les phases de transitions et la phase stationnaire.
- La dépendance des ports qui est introduite ici par le chaînage des flux de lecture avec le flux d'écriture.
- Les techniques de sélection des requêtes d'accès sans conflit.

5.3.3.1 Type des accès

En modèle analytique les arrivées (requêtes) sont générées de manière aléatoire afin d'assurer l'uniformité de distribution des accès sur les bancs mémoires. Cette hypothèse répond uniquement au problème d'accès dispersés. Dans les applications réelles les accès vectoriels sont souvent réguliers (par pas constants), ils représentent une partie importante de la totalité des accès vectoriels.

Accès réguliers

Le problème de conflits pour les accès réguliers est étudié par Oed & al. sur les machines Cray [Oed et al.85]. Ils ont dégagés des conditions nécessaires et suffisantes pour que les flux d'accès en parallèles puissent être réalisés sans conflits.

Les accès réguliers par pas impair, généralement, ne posent pas de problème de conflits, car les bancs sont uniformément et périodiquement sollicités durant tout le flux d'accès. En revanche les accès par pas pair causent beaucoup de conflits de bancs occupés. Seuls les bancs d'adresses divisibles ou multiples du pas sont référencés. Ce genre d'accès pénalise aussi le modèle TVD voir la table de la figure (5.12).

Les méthodes de répartition des données présentées dans le premier chapitre peuvent résoudre le problème de non uniformité des adresses des bancs sollicités lorsque le pas est pair, pour se ramener aux mêmes conditions que dans le cas des pas impairs.

Modèles	Accès contigus	Accès par pas >1	Pas pair	Pas impair	Accès dispersés
TVD	0,819403	0,520555	0,396089	0,807824	0,870521
Classique	0,806993	0,383842	0,336151	0,454175	0,525958

Figure 5.12: L'efficacité en fonction du type d'accès, modèle simulé.

Accès dispersés

En plus des accès par pas supérieur à 1, le modèle classique souffre aussi du problème de conflits lors de l'exécution des opérations gather/scatter. Les techniques de répartitions de données sont impuissantes et présentent même des chutes de performances considérables. Le modèle TVD semble une très bonne solution pour ce type d'accès. La table de la figure (5.12) montre une efficacité de 87%, proche du maximum.

5.3.3.2 La taille des registres vectoriels

La taille des registres vectoriels dans le modèle analytique est supposée suffisamment grande pour que le système rentre dans un état stationnaire. En effet c'est la taille des vecteurs qui décide de la durée de la phase stationnaire. La phase de start-up ne peut être négligée devant la phase stationnaire que dans la cas où VL est assez grand.

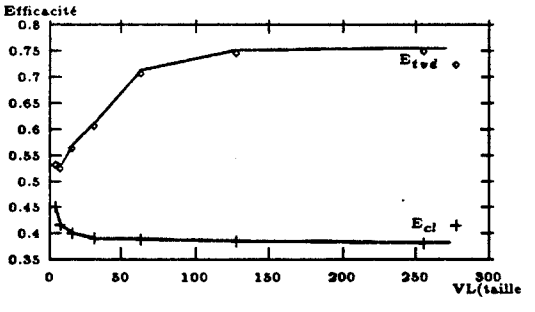


Figure 5.13: L'efficacité en fonction de la taille des vecteurs, modèle simulé.

Lorsque la taille des vecteurs est assez petite le système ne rentre pas en phase d'équilibre. Les buffers se vident plus tôt, sur le peu de requêtes qui restent le choix des requêtes d'accès sans conflits est difficile. Le système passe de la phase start-up à la phase de fin sans passer par la phase stationnaire. Le calcul d'adresses est souvent terminé dans la phase de transition. La figure (5.13) montre l'accroissement de l'efficacité pour atteindre l'état d'équilibre. A l'opposé l'efficacité du modèle classique diminue avec l'augmentation de VL . Ceci s'explique facilement par le fait que les effets des conflits s'accumulent en induisant un retard de plus en plus grand.

La figure (5.14) montre la variation du speed-up en fonction de la taille des registres vectoriels. On peut considérer que le système est en équilibre à partir d'une taille de VL = 32. Nous avons

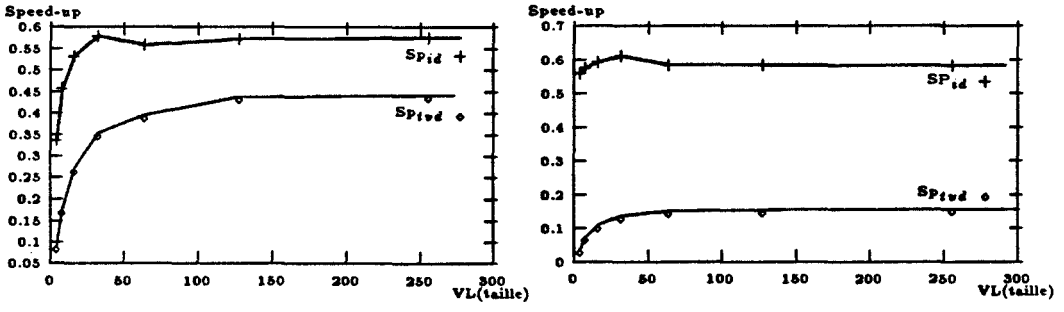


Figure 5.14 : Le speed-up en fonction de la taille des registres vectoriels. Les courbes de gauche : simulation sans dépendance. Courbes de droite : simulation avec dépendance du port d'écriture de ceux de lecture.

vu que pour le modèle classique tout conflit d'accès provoque un retard supplémentaire pour le flux d'accès. Pendant la phase de transition les conflits ne sont pas résolus de manière efficace dans le modèle TVD d'une part et d'autre part le retard, dû aux conflits pour le classique, n'est pas significatif, car le TVD en subit aussi.

Notons qu'en période stationnaire le speed-up est d'environ 50% sur des vecteurs de 128 éléments.

5.3.3.3 Dépendances des accès, le temps de latence des pipelines

Le temps de latence des pipelines de calcul T_p intervient lors de l'exécution du premier type d'opération. Le port d'écriture dépend des deux ports de lecture, car l'accès en écriture est lié à la disponibilité des composantes dans le registre vectoriel destination qui, elles aussi, dépendent des conflits d'accès mémoire lors des lectures. L'exécution d'une opération du type $C = A\ op\ B$, où A, B et C sont des vecteurs résidents en mémoire présente trois phases suivant le nombre de flux actifs en chaque phase. La première phase, les deux flux d'accès en lecture sont actifs. La deuxième phase les trois flux sont actifs et dans la dernière phase seul le flux d'écriture est actif (voir la figure (5.15)).

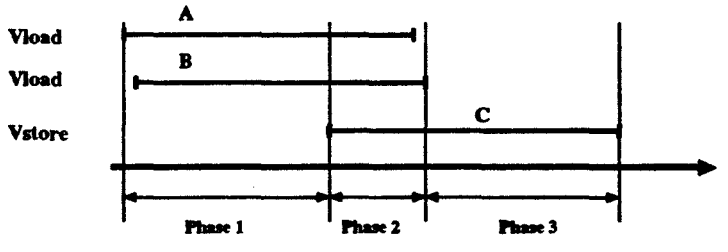


Figure 5.15 : Phases d'exécution des trois flux d'accès.

L'augmentation du temps de latence des pipelines permet d'augmenter le start-up du flux d'écriture (phase 1). Pour des vecteurs de petite taille la phase 2 est absente; c'est-à-dire les accès en lecture sont terminés avant que le flux d'accès en écriture ne soit déclenché. Le nombre de conflits est moindre car le débit des processeurs qui est caractérisé par la phase 2 est faible,

ce qui explique la borne supérieure de (62%) qui est atteinte pour un temps de latence du pipeline égal à 2. On peut remarquer que la courbe (figure (5.16)) de variation de l'efficacité en fonction de T_p est approximativement linéaire.

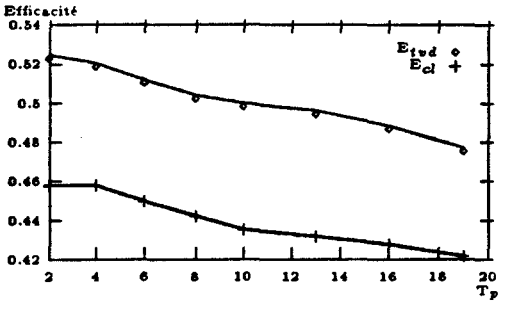


Figure 5.16: Variation du temps de latence des pipelines de calcul.

La dépendance des accès réduit les performances de la machine quelque soit le modèle utilisé, car la phase où les flux de lecture et d'écriture sont actifs simultanément est réduite à cause du start-up nécessaire à l'arrivée de la première requête au port d'écriture et la limitation de la taille des vecteurs.

5.3.3.4 Algorithmes de sélection

Nous avons simulés certaines techniques que nous avions décrit dans le chapitre 2. Ces techniques sont:

- **I– Priorité fixe**: Les conflits de sections et de simultanéité sont résolus en plaçant la priorité fixe entre les ports et les processeurs. Nous avons considéré que le port d'écriture est moins prioritaire que les deux autres. L'avantage est donné aux ports de lecture pour ne pas pénaliser les pipelines de calcul et le rangement du résultat. Certes elle est plus facile à implémenter, mais elle favorise le port et le processeur le plus prioritaire.
- **II – Priorité cyclique**: tous lés ports et les processeurs sont soumis à la priorité cyclique.
- **III – Priorité de scrutation**: seuls les processeurs sont gérés avec cette priorité. Les ports sont gérés avec la priorité cyclique.
- **IV – Priorité de scrutation par zone**: Cette technique est appliquée de la même manière que (**III**). Elle donne de meilleurs résultats que les autres (voir la table de la figure (5.17)). C'est l'algorithme que nous avons utilisé pour réaliser les précédentes simulations.

5.4 Conclusion

Nous avons montré que les résultats théoriques sont conformes aux résultats de simulations. Les performances théoriques de notre modèle de traitement vectoriel désordonné sont de bonnes approximations des performances réelles lorsque la taille des vecteurs est assez grande.

	I	II	III	IV
Efficacité TVD	0,476952	0,522271	0,535673	0,539823
Efficacité Classique	0,375882	0,384544	0,382461	0,385211
Speed-up TVD	0,1965221	0,203801	0,223990	0,225664
Speed-up idéal	0,605509	0,594806	0,596937	0,598001

Figure 5.17 : Le speed-up et l'efficacité pour chacune des techniques.

Le modèle TVD permet d'obtenir de très hautes performances pour un nombre de bancs assez réduit. Le débit effectif de l'ensemble des bancs doit satisfaire le débit des processeurs. Par exemple une machine à deux processeurs de 3 ports d'accès mémoires chacun nécessite une mémoire de 32 bancs de temps de latence $T = 4$ pour échanger les données avec une efficacité de 70%. Les deux débits, sont presque équivalents; débit des processeurs est de 6 requêtes par cycle et le débit effectif de la mémoire est 8 données par cycle processeur.

Nous avons montré aussi que le modèle TVD est très performant lorsque la taille des registres vectoriels est assez grande. L'augmentation de la taille des vecteurs pénalise sévèrement le modèle classique.

Le modèle TVD n'est pas sensible aux types des accès. Cependant les accès par pas puissance de 2 génèrent beaucoup de conflits de bancs occupés, car les accès sont répartis sur un très petit nombre de bancs.

La dépendance des ports d'écriture de ceux de lecture réduit l'efficacité et le speed-up d'un facteur k. Il est préférable de générer des flux d'accès indépendants, si c'est possible.

Comparé au modèle idéal, l'écart est constant en mode stationnaire, sur des valeurs des paramètres par défaut, dans chaque cas de figure, sauf pour la variation du temps de latence de la mémoire.

Conclusion

Les systèmes multiprocesseurs vectoriels à mémoires communes ont la caractéristique d'être simples et facilement programmables. Cependant leur problème majeur est la chute des performances due aux conflits d'accès à la mémoire. Jusqu'à présent il n'existe pas de méthodes générales qui éliminent ces conflits. Les interactions qui existent entre plusieurs programmes au niveau des accès à la mémoire dépendent du temps et de leur comportement vis à vis des accès mémoires. Ces deux facteurs ne sont pas contrôlables de manière formelle.

Nous avons proposé un modèle original pour le traitement vectoriel. Ce modèle permet d'améliorer les performances de ces machines tout en gardant une des caractéristiques importantes de celles-ci, à savoir le chaînage des unités fonctionnelles de chaque processeur.

Les caractéristiques fondamentales de la machine TVD sont la méthode d'accès désordonnés aux éléments d'un vecteur qui ne dépend aucunement des applications en cours d'exécution et le modèle d'exécution qui véhicule ce désordonnancement à travers toutes les unités fonctionnelles de calcul de chaque processeur.

Nous avons montré au cours de cette étude que les deux caractéristiques sont nécessaires pour obtenir de bonnes performances sur un calculateur multiprocesseur vectoriel. L'expérience montre que l'application de la méthode d'accès désordonnés n'est pas suffisante pour améliorer les performances. Elle se révéle défaillante, si elle nécessite l'absence de chaînage entre les unités fonctionnelles de calcul et les unités fonctionnelles d'entrées/sorties.

Nous avons associé à la méthode d'accès un modèle d'exécution qui véhicule l'effet des accès désordonnés à l'intérieur des unités vectorielles des processeurs. L'ensemble de ce fonctionnement est contrôlé par l'unité de sélection. Avec les progrès technologiques en terme de capacité d'intégration, cette unité peut être intégrée dans l'unité de contrôle et de commande du processeur.

Nous avons ensuite construit un modèle analytique pour évaluer le gain en performances de notre modèle par rapport au modèle classique. Ce modèle est construit sur des hypothèses simples. C'est un modèle approximatif, et il rejoint les résultats de simulation lorsque le système est en régime stationnaire. Nous avons montré que le modèle TVD est mieux adapté que le modèle classique quelque soit le type d'accès à effectuer. Sur les opérations gather/scatter, pour lesquelles toutes les méthodes de répartition de données ont échouées, le modèle TVD obtient des meilleures performances.

Le modèle de traitement désordonné n'est pas difficile à implémenter. Il induit quelques modifications sur les unités qui véhiculent le traitement vectoriel. Ces modifications peuvent varier d'une machine à l'autre suivant son organisation mémoire et l'architecture de son réseau d'interconnexions.

Nous avons montré (théoriquement et par des simulations) que le TVD affiche de très bonnes

performances lorsque les accès sont uniformément distribués sur la mémoire. L'association du modèle TVD avec une méthode de répartition uniforme des données sur l'ensemble des bancs mémoires ne peut qu'améliorer ses performances.

Appendix A

Prototype d'un processeur PVD

Nous avons implémenté les unités concernées par le traitement vectoriel désordonné pour étudier la faisabilité et la complexité de réalisation d'un processeur PVD. La saisie et la simulation se sont faites à l'aide des outils **Solo-1400**, un produit de chez **ES2**.

Ce prototype contient le minimum d'unités permettant de faire fonctionner le processeur en mode de traitement vectoriel désordonné.

Il est composé de deux pipelines de calcul, d'une unité de contrôle et de commande, d'une unité de sélection, d'une unité de registres vectoriels, des registres scalaires, d'une unité d'accès mémoires et de trois ports mémoires. La figure (A.1) montre les unités qui véhiculent les flux de données vectorielles et les différents chemins de données entre elles.

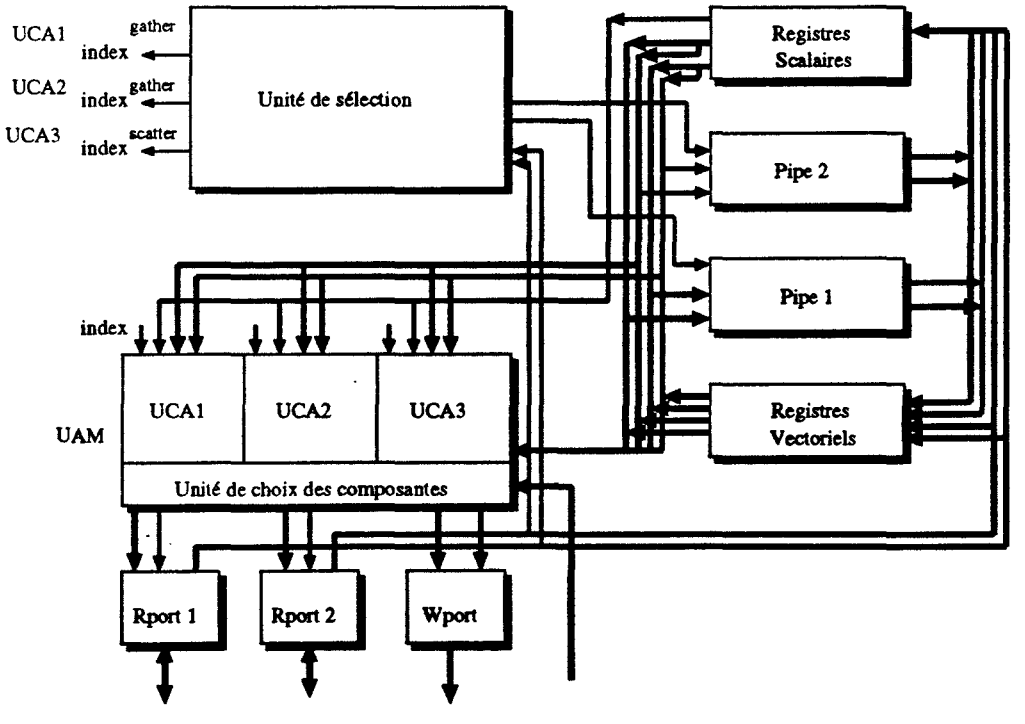


Figure A.1 : Le prototype d'un processeur vectoriel désordonné.

On peut déclencher deux opérations de lectures et une opération d'écriture qui peuvent être chaînées avec une des opérations que les deux pipelines peuvent exécuter. Sur ce prototype on ne peut réaliser pas des opérations gather/scatter simultanées. On traite un gather ou un

scatter à la fois. Ceci n'est pas difficile à réaliser; il suffit d'augmenter le nombre de connexions et les nombre d'unités dans le processeur. C'est un processeur 16 bits qui peut supporter une mémoire de 64 bancs de 2k octets.

A.1 L'unité d'accès mémoire

Cette unité est indépendante, elle s'occupe de tout ce qui est calcul d'adresses et sélection des composantes candidates pour des accès mémoires durant un cycle processeur donné. Cette unité est implémentée en Solo-1400 selon le schéma de la figure (A.2). Evidemment l'implémentation de cette unité par décomposition en plusieurs composants nous est imposé par le logiciel Solo-1400 qui rend le routage très difficile à effectuer et de part sa capacité d'intégration.

Figure A.2 : Schéma global d'implémentation de l'unité d'accès mémoire.

Le premier niveau de composants de la figure (??) intégre les trois unités de calcul d'adresses. Les deux UCALs sont intégrées dans un seul chip et l'UCAE est intégrée dans autre chip. Les trois circuits (2 UCALs et UCAE) peuvent être implémentées dans un seul chip. La figure (A.3) représentant le chip des deux UCALs montre bien qu'il reste encore de la place; le circuit n'est pas dense, cependant le chip global des trois unités ne peut pas supporter autant de signaux externes d'entrées/sorties.

Figure A.3 : Circuit des deux unités de calcul d'adresses en lecture : UCAL.

Appendix B

Le registre vectoriel

Contrairement aux machines vectoriels classiques, sur la machine TVD les lectures et les écritures dans un registre vectoriel sont aléatoires. Un registre vectoriel peut être vu comme une file de registres scalaires muni d'un décodeur d'adresse en lecture et d'un autre décodeur en écriture.

Il reçoit en entrée les signaux suivants :

- le bus d'adresses en lecture
- le bus d'adresse en écriture
- le bus de données en écriture
- un signal d'écriture
- un signal de contrôle de lecture
- un signal d'horloge.

En sortie il délivre la donnée lue (un bus de donnée de lecture).

La lecture et l'écriture simultanées dans un registre vectoriel est possible lorsque deux instructions vectorielles sont chaînées. Le problème de lecture et d'écriture dans un même emplacement ne se pose pas, car avant de lire une donnée il faut qu'elle soit disponible, donc déjà écrite.

Chaque registre vectoriel contient 64 éléments de 16 bits. Chaque élément d'un registre vectoriel (une composante) est formé de 16 cellules de mémorisation d'un bit. Cette dernière est réalisée par une porte `Nor`, une bascule `bdff` (D) et d'un inverseur pour amplifier la sortie de la bascule. Le circuit est montré dans la figure (B.1).

Chaque composante du registre peut être adressée directement en lecture ou en écriture par l'intermédiaire d'un décodeur (6×64).

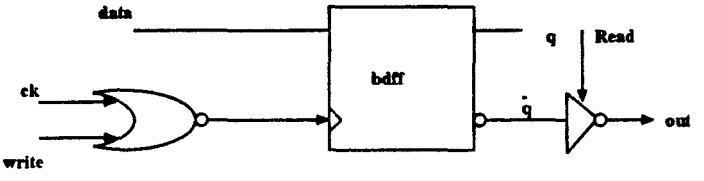
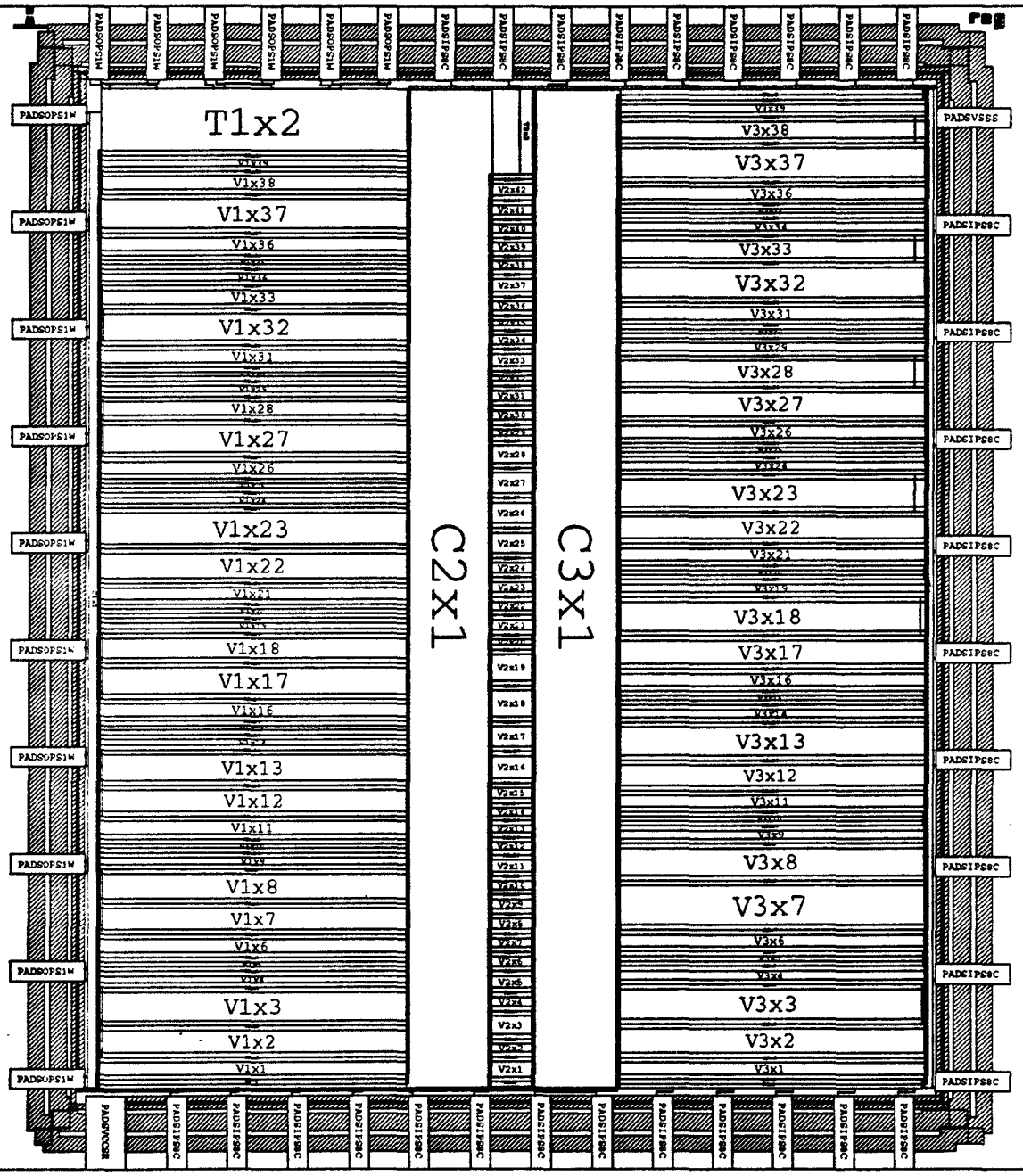


Figure B.1 : Cellule de base d'un élément d'un registre vectoriel.



LISATION" of file "reg.cif" plotted on Fri Feb 5 1993 at 01:33:33

Figure B.2 : Circuit imprimé d'un registre vectoriel.

Appendix C

l'unité de sélection

Cette unité doit assurer la sélection des composantes et le bon fonctionnement des unités fonctionnelles par une gestion efficace de leurs registres masques. Son circuit est montré en figure (C.1).

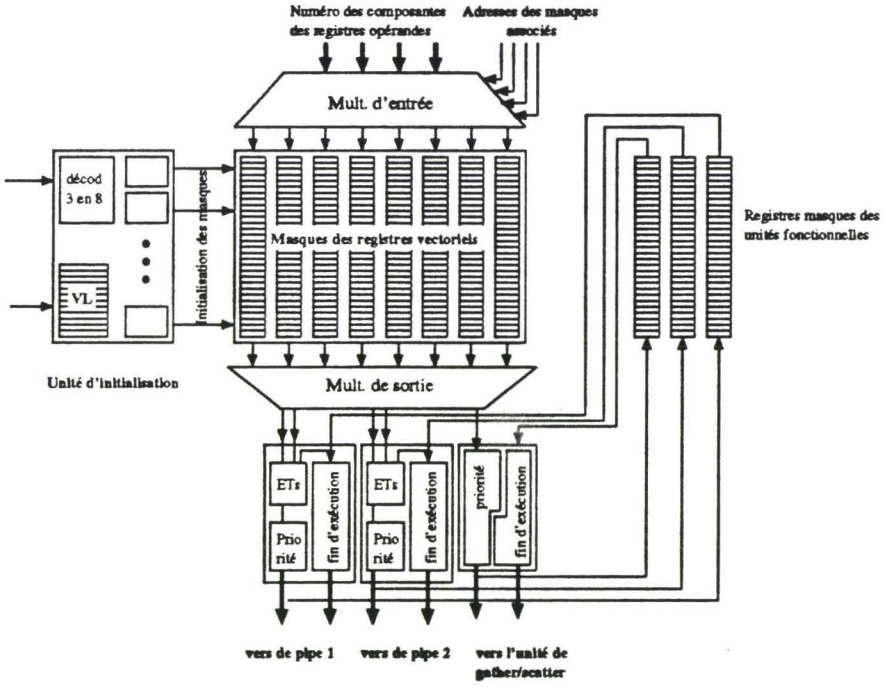


Figure C.1 : Circuit général de l'unité de sélection.

Cette unité est composée des composants suivants :

- Composant d'initialisation des registres masques. Chaque registre masque alloué comme opérande (associé à un registre vectoriel), ainsi que le registre masque de l'unité fonctionnelle sont initialisés en fonction du registre `VL`. Ce composant reçoit en entrée le numéro du registre et la taille du vecteur à traiter dans `VL` et délivre en sortie la valeur initiale du registre masque associé.

- Composant des masques des registres vectoriels. Les masques actifs sont mis à jour à chaque cycle, et ils sont aussi exploré pendant le même cycle pour contrôler la terminaison

de l'opération et pour déclencher le traitement sur un couple de composantes disponibles et conformes.

- Les composants de commandes des unités fonctionnelles contrôlent le déroulement des opérations (détection de la fin et déclenchement d'un nouveau calcul sur des composantes disponibles et conformes).
- L'ensemble des masques actifs sont identifiés par deux composants Multiplexeur et Démultiplexeur qui occupent une place importante dans le circuit.

Notons que la lecture et l'écriture pendant le même cycle dans un même registre masque. A un instant t, la lecture se fera sur la donnée de l'instant précédent $(t-1)$ du masque, donc il faut lire avant d'écrire la nouvelle valeur du masque. Ceci est rendu possible grâce au circuit élémentaire de la figure (C.2).

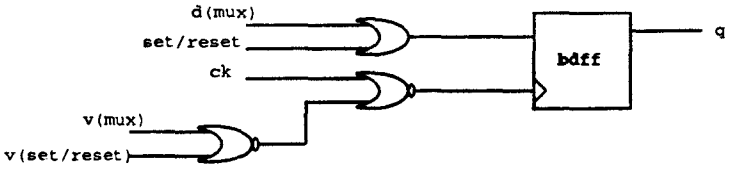


Figure C.2 : Cellule de base d'un registre masque.

L'utilisation des latch ou des bascules D avant ou après le multiplexeur (ou même après les registres masques) permet la synchronisation des lectures et des écritures. L'écriture dans la bascule D est retardée en mettant en place des buffers : avant que la nouvelle valeur arrive à l'entrée de la bascule D (bdff), la donnée précédente est déjà lue.

Table des figures

Bibliographie

[Abusufah et al.86] AbuSufah (W.) et Mahoney (A.D.). – Vector processing on the ALLIANT FX/8 Multiprocessor. *In : Proc. of the Int'l. Conference on Parallel Processing*, pp. 559–563.

[Allen80] Allen (A.O.). – Queueing Models of Computer Systems. *IEEE Computer*, vol. 13 (1), April 1980, pp. 13–24.

[August et al.89] August (M.C.), Brost (G.M.), Hsiung (C.C.) et Schiffleger (A.J.). – Cray X-MP : The Birth of a Supercomputer. *IEEE Computer*, vol. 22 (1), January 1989, pp. 45–52.

[Bailey87] Bailey (D.H.). – Vector Computer Memory Bank Contention. *IEEE Transactions on Computers*, vol. C-36 (3), March 1987, pp. 293–298.

[Barnes et al.68] Barnes (G.H.), Brown (R.M.), Kato (M.), Kuck (D.J.), Slotnick (D.L.) et Stockes (R.A.). – The ILLIAC IV Computer. *IEEE Transactions on Computers*, vol. C-17 (8), August 1968, pp. 746–770.

[Baskett et al.76] Baskett (F.) et Smith (A.J.). – Interference in Multiprocessor Computer Systems with Interleaved Memory. *Communication of the ACM*, vol. 19 (6), June 1976, pp. 327–334.

[Batcher77] Batcher (K.E.). – The Multidimisional Access Memory in STARAN. *IEEE Transactions on Computers*, vol. ** (2), February 1977, pp. 174–177.

[Bhandarkar75] Bhandarkar (D.P.). – Analysis of Memory Interference in Multiprocessors. *IEEE Transactions on Computers*, vol. C-24 (9), September 1975, pp. 897–908.

[Bird et al.91] Bird (P.L.) et Uhlig (R.A.). – Using Lookahead to Reduce Memory Bank Contention for Decoupled Operand References. *In : Proc. of Supercomputing*, pp. 187–196. – Albuquerque, New Mexico (18-22), November 1991.

[Briggs et al.77] Briggs (F.A.) et Davidson (E.S.). – Organization of Semiconductor Memories for Parallel-Pipelined Processors. *IEEE Transactions on Computers*, vol. C-26 (2), February 1977, pp. 162–169.

[Brooks et al.91] Brooks (J.), Grassel (C.) et Sandness (R.). – Modeling Perfomance of the Cray Y-MP C90 Architecture. *In : ********.

[Bucher et al.90] Bucher (I.Y.) et Calahan (D.A.). – Access Conflicts in Multiprocessor Memories Queueing Models and Simulation Studies. *In : Proc. of the Int'l. Conf. on Supercomputing*, pp. 428–438. – Amsterdam, The Netherlands, June 1990.

[Budnik et al.71] Budnik (P.) et Kuck (D.J.). – The Organization and Use Parallel Memories. *IEEE Transactions on Computers*, vol. C-20 (12), December 1971, pp. 1566–1569.

[Burnet et al.70] Burnet (G.) et Coffmann (JR.). – A Study of Interleaved Memory Systems. *In : Proc. AFIPS, Spring Joint Computer Conference,vol.36*, éd. par Press (AFIPS), pp. 467–474. – Montvale, N.J., 1970.

[Burnett et al.73] Burnett (G.J.) et Coffmann (JR.). – A Combinatorial Problem Related to Interleaved Memory Systems. *Journal of the ACM*, vol. 20 (1), January 1973, pp. 39–45.

[Burnett et al.75] Burnett (G.J.) et Coffmann (JR.). – Analysis of Interleaved Memory Systems Using Blockage Buffers. *Communication of the ACM*, vol. 19 (2), February 1975, pp. 91–95.

[Calahan et al.88] Calahan (D.A.) et Bailey (D.H.). – Measurement and Analysis of Memory Conflicts on Vetcor Multiprocessors. *In : Performance Evaluation of Supercomputers*, pp. 83–106.

[Chang et al.77] Chang (D.Y.), Kuck (D.J.) et Lawrie (D.H.). – On the Effective Bandwidth of Parallel Memories. *IEEE Transactions on Computers*, vol. C-24 (5), May 1977, pp. 480–489.

[Chen et al.89] Chen (C-L.) et Liao (C-K.). – Analysis of Vector Access Performance on Skewed Interleaved Memory. *In : The Annual International Symposium on Computer Architecture*, pp. 387–394. – (28(5)–1(6)), Jerusalem, Israel, May 1989.

[Cheung et al.84] Cheung (T.) et Smith (J.E.). – An Analysis of the Cray X-MP Memory System. *In : Proc. of the 2nd Int'l. Conference on Parallel Computing*, pp. 499–505. – Bellaire, MI. (21–24), August 1984.

[Cheung et al.86] Cheung (T.) et Smith (J.E.). – A Simulation Study of the Cray X-MP Memory System. *IEEE Transactions on Computers*, vol. C-35 (7), July 1986, pp. 613–622.

[Coffman jr et al.71] Coffman JR (E.G.), Burnett (G.J.) et Snowdon (R.A.). – On the Performance of Interleaved Memories with Multiple-Word Bandwidth. *IEEE Transactions on Computers*, vol. C-20 (12), December 1971, pp. 1570–1573.

[Coffman jr et al.78] Coffman, JR (E.G.) et Hofri (M.). – A Class of FIFO Queues Arising in Computer Systems. *SIAM Journal on Applied Mathematic*, vol. 26 (5), September-October 1978, pp. 864–880.

[Coffman jr68] Coffman JR. (E.G.). – A Simple Probability Model Yielding Performance Bounds for Modular Memory Systems. *IEEE Transactions on Computers*, no1, January 1968, pp. 87–89.

[Colbourn et al.92] Colbourn (C.J.) et Heinrich (K.). – Conflict-Free Access to Parallel Memories. *Journal of Parallel and Distributed Computing*, vol. 14, 1992, pp. 193–200.

[Creange et al.88] Creange (M.), Frailong (J.M.) et Plateau (B.). – Performance of Vector Computer with Memory Contention. *In : In Higth Performance Computer System*, éd. par Gelenbe (E.). pp. 195–208. – North-Holand.

[Das et al.85] Das (C.R.) et Bhuyan (L.N.). – Bandwidth Availability of Multiple-Bus Multiprocessors. *IEEE Transactions on Computers*, vol. C-35 (10), October 1985, pp. 919–926.

[dD91] de Dinnechin (B. D.). – An Ultra Fast Euclidean Division Algorithm for Prime Memory Systems. *In : Proc. of Supercomputing*, pp. 56–65. – Albuquerque, New Mexico (18-22), November 1991.

[Deitel84] Deitel (H.M.). – *An Introduction to Operating Systems (Revised First Edition)*. – (Massachusetts), Addison Wesley, 1984, 353–412p.

[Dekeyser et al.90] Dekeyser (J-L.), Kechadi (M.T.), Marquet (Ph.) et Preux (Ph.). – Un modèle d'exécution vectoriel pipeline désordonné. *Rapport Interne LIFL ERA-85*, October 1990.

[Dekeyser et al.91] Dekeyser (J-L.), Kechadi (M.T.), Marquet (Ph.) et Preux (Ph.). – Disordered Vector Pipelined Processor. *In : Proc. ISMM Workshop on Parallel Computing*, pp. 36–39. – Trani, Italie (10-13), September 1991.

[Dekeyser et al.92a] Dekeyser (J-L.), Kechadi (M.T.), Marquet (Ph.) et Preux (Ph.). – Performance Improvement for Vector Pipeline Multiprocessor Systems Using a Disordered Execution Model. *In : Int'l. Symp. on Computer Architecture (ISCA '92)*, pp. 433–433. – Queensland, Australia (19-21), May 1992.

[Dekeyser et al.92b] Dekeyser (J-L.), Kechadi (M.T.), Marquet (Ph.) et Preux (Ph.). – Résolution des Conflits d'Accès à la Mémoire par Désordonnancement. *In : RenPar4, 4es Rencontres du Parallélisme*, pp. 96–99. – Villeneuve d'ascq, France (18-20), March 1992.

[Dekeyser90] Dekeyser (J-L.). – *Algorithmes, Langages et Architectures Vectoriels — Le Projet WEST. Mémoire d'habilitation à diriger des recherches, Laboratoire d'Iformatique de Lille, Université de Lille 1.* – December 1990.

[Dongara87] Dongara (J.J.). – *Experimental Parallel Computing Architectures, Special Topics in Supercomputing.* – (Amsterdam), North-Holland, 1987.

[Eoyang et al.88] Eoyang (C.), Mendez (R.H.) et Lubeck (O.M.). – The Birth of the Second Generation : The Hitachi S-820/80. *In : Proc. of the Int'l. Conference on Supercomputing*, pp. 296–303.

[Eoyang et al.91] Eoyang (C.) et Mendez (R.). – NEC SX-3 Performance Study. *Vector Register*, vol. 4 (1), March 1991, pp. 3–7.

[Fatoohi89] Fatoohi (R.A.). – Vector Performance Analysis of three Supercomuters : Cray-2, Cray Y-MP, and ETA10-Q. *In : Proc. of Supercomputing'89*, pp. 779–788. – Reno, Nevada (13–17), November 1989.

[Flores64] Flores (I.). – Derivation of a Waiting-Time Factor for a Multiple-Bank Memory. *Journal of the ACM*, vol. 11 (7), July 1964, pp. 265–282.

[Flynn72] Flynn (M.J.). – Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, vol. C-21 (9), September 1972, pp. 948–960.

[Frailong et al.85] Frailong (J.M.), Jalby (W.) et Lenfant (J.). – XOR-Schemes : a Flexible Data Organization in Parallel Memories. *In : Proc. of the Int'l. Conference on Parallel Processing.*

[Frailong et al.87] Frailong (J.M.), Jalby (W.) et Lenfant (J.). – Diamond Schemes : an Organization of Parallel Memories for Efficient Array Processing. *In : in Organisation mimoire dans les supercalculateurs.*

[Gottlieb92] Gottlieb (A.). – Architectures for Parallel Supercomputing. *In : Proc. PACTA'92.* – Barcelona, Span.

[Hack86] Hack (J.J.). – Peak vs. Sustained Performance in Highly Concurrent Vector Machines. *IEEE Computer*, vol. 19 (9), September 1986, pp. 11–19.

[Harper et al.87] Harper (D.T.) et Jump (J.R.). – Vector Access Performance in Parallel memories Using a Skewed Storage Scheme. *IEEE Transactions on Computers*, vol. C-36 (12), December 1987, pp. 1440–1449.

[Harperiii et al.91] HarperIII (D.T.) et Lineberger (D.A.). – Conflict-Free Vector Access Using a Dynamic Storage Scheme. *IEEE Computer*, vol. 40 (3), March 1991, pp. 276–283.

[Hellerman66] Hellerman (H.). – On the Average Speed of a Multiple-Module Storage System. *IEEE Transactions on Electronic Computers*, vol. C-15 (1), August 1966, pp. 670–670.

[Hockney et al.88] Hockney (R.W.) et Jesshope (C.R.). – *Parallel Computers 2 : Architecture, Programming and Algorithms.* – (Bristol and Philadelphia), Adam Hilger, 1988.

[Hockney88] Hockney (R.W.). – Proble Related Performance for Supercomputers. *In : Perfromance Evaluation of Supercompuetrs.* pp. 215–235. – North-Holland.

[Hoogendoor77] Hoogendoor (C.H.). – A General Model for Memory Inteference in Multiprocessors. *IEEE Transactions on Computers*, vol. C-26 (10), October 1977, pp. 998–1005.

[Hwang et al.84] Hwang (Kai) et Briggs (Fayé A.). – *Computer Architecture and Parallel Processing.* – (New-York), McGraw-Hill, 1984.

[Hwang et al.89] Hwang (K.), Tseng (P.S.) et Kim (D.). – An Orthoganal Multiprocessor for Parallel Scientific Computations. *IEEE Computer*, vol. 38 (1), January 1989, pp. 47–61.

[Ibbett et al.89] Ibbett (R.N.) et Topham (N.P.). – *Architecture on High Performance Computers, Vol. 1.* – MacMillan Education LTD, 1989.

[inc88] inc. (Cray Recherche). – *Cray Y-MP Computer Systems Functional Description Manual, HR-4001*, January 1988.

[Intel91] Intel. – *Paragon^{TM} XP/S — Product Overview*, 1991.

[Jegou86] Jegou (Y.). – Le DSPA, Un Pipeline Synchronisé par les Données. *Centre de documentation INRIA (IRISA), Rapport de Recherche*, no574, October 1986.

[Kim et al.89] Kim (K.) et Kumar (V.K. Prasanna). – Perfect Latin Squares and Parallel Array Access. *In: The Annual International Symposium on Computer Architecture*, pp. 372–379. – (28(5)–1(6)), Jerusalem, Israel, May 1989.

[Knuth et al.75] Knuth (D.A.) et Rao (G.S.). – Activity in an Interleaved Memory. *IEEE Transactions on Computers*, vol. c-24 (9), September 1975, pp. 943–945.

[Knuth73] Knuth (D.E.). – *The Art of Computer Programming, Vol. 3.* – pp. 508–510, Addison-Wesley, 1973.

[Kuck et al.82] Kuck (D.J.) et Stockes (R.A.). – The Burroughs Scientific Processor(BSP). *IEEE Transactions on Computers*, vol. C-31 (5), May 1982, pp. 363–375.

[Lawrie et al.82] Lawrie (D.H.) et Vora (C.R.). – The Prime Memory System for Array Access. *IEEE Transactions on Computers*, vol. C-31 (5), May 1982, pp. 435–443.

[Lawrie75] Lawrie (D.H.). – Access and Alignement of Data in an Array processor. *IEEE Transactions on Computers*, vol. C-24 (12), December 1975, pp. 1145–1155.

[Lee88] Lee (De Lei). – Scrambled Storage for parallel memory Systems. *Computer Architecture News*, vol. 16 (2), May 1988, pp. 232–239.

[Leiserson et al.91] Leiserson (C.E.), Abuhamdeh (Z.S.), Douglas (D.C.), Feynman (C.R.), Ganmukhi (M. N.), Hill (J.V.), Hillis (W.D.), ans M.A.St. Pierre (B.C. Kuszmaul), Wells (D.S.), Wong (M.C.), Yang (S.W.) et Zak (R.). – The Network Architecture of the Connection Machine CM-5. *Thinking Machines Corporation – Cambridge, Massachusetts 02142*, Jul 1991, pp. 1–14.

[Litaize90] Litaize (D.). – Arcitectures Multiprocesseurs a Mémoire Commune. *In: Architectures Nouvelles de Machines, Deuzième Symposium*, pp. 1–71. – Toulouse, (12–14), September 1990.

[Lubeck et al.85a] Lubeck (O.), Moore (J.) et Mendez (R.). – A Benchmarck Comparison of three Supercomputers : Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. *IEEE Computer*, vol. 18 (12), December 1985, pp. 10–23.

[Lubeck et al.85b] Lubeck (O.), Moore (J.) et Mendez (R.). – A Benchmark Comparison of three Supercomputers : Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. *In : Proc. of the Int'l. Conference on Supercfomputing Systems*, pp. 320–327. – St. petersburg, Florida, USA. (15–20), December 1985.

[Marsan et al.82] Marsan (M.A.) et Gerla (M.). – Markov Models for Multiple Bus Multiprocessor Systems. *IEEE Transactions on Computers*, vol. C-31 (3), March 1982, pp. 239–248.

[Mudge et al.87] Mudge (T.N.), Hayes (J.P.) et Winsor (D.C.). – Multiple Bus Architecture. *IEEE Computer*, vol. 19 (6), June 1987, pp. 42–48.

[Oed et al.85] Oed (W.) et Lange (O.). – On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems. *IEEE Transactions on Computers*, vol. C-34 (10), October 1985, pp. 949–957.

[Oed et al.86] Oed (W.) et Lange (O.). – Modeling Measurement, and Simulation of Memory Interference in the Cray X-MP. *Parallel Computing*, vol. 3 (4), October 1986, pp. 343–358.

[Pack77] Pack (C.D.). – The Output of a D/M/1 Queue. *SIAM Journal on Applied Mathematic*, vol. 32 (3), May 1977, pp. 571–587.

[Pack78] Pack (C.D.). – The Output of Multiserver Queuing Systems. *Operations research*, vol. 26 (3), May-June 1978, pp. 492–509.

[Potier87] Potier (D.). – Analysis of Determinat Factors for the Performance of Vector Machines. *In : Supercomputing : State-of-the Art.*, éd. par North-Holland, pp. 221–236. – INRIA, 1987.

[Preux91] Preux (Ph.). – *Une Machine Virtuelle Vectorielle — Conséquences sur l'Architecture des Machines Vectorielles.* – PhD thesis, Laboratoire d'Iformatique de Lille, Université de Lille 1, January 1991.

[Raghavan et al.90] Raghavan (R.) et Hayes (J.). – On randomly Interleaved Memories. *In : Proc. of Supercomputing*, pp. 49–58. – New-York (12–16), USA., November 1990.

[Raghavendra et al.90] Raghavendra (C.S.) et Boppana (R.). – On Methods for Fast and Efficient Parallel Memory Access. *In : Proc. of the International Conference on Parallel Processing*, pp. I–76–I–83. – (13–17)., August 1990.

[Ramamoorthy et al.77] Ramamoorthy (C.V.) et Li (H.F.). – Pipeline Architecture. *Computing Surveys*, vol. 9 (1), March 1977, pp. 61–102.

[Rau et al.89a] Rau (B.R.), Yen (D.W.L.), Yen (Wei) et Towle (R.A.). – The CYDRA 5 Department Supercomputer (Design Philosophies, Decisions, and Trade-offs). *IEEE Computer*, vol. 22 (1), January 1989, pp. 12–35.

[Rau et al.89b] Rau (R.), Schansker (M.S.) et Yen (D.W.L.). – The Cydra 5 Stride-Intensive Memory System. *In : Proc. of the Int'l. Conf. on Parallel Processing, vol. I*, pp. 242–246.

[Rau79] Rau (B. Ramakrichna). – Interleaved Memory bandwidth in a Model of a Multiprocessor Computer System. *IEEE Transactions on Computers*, vol. C-28 (9), September 1979, pp. 678–681.

[Rau91] Rau (B.R.). – Pseudo-Randomly Interleaved Memory. *Proc. of the Int'l. Symposium on Computer Architecture in ACM CAN*, vol. 19 (3), 1991, pp. 74–83.

[Ravi72] Ravi (C.V.). – On the Bandwidth and Interference in Interleaved Memory Sysdtems. *IEEE Transactions on Computers*, vol. C-21 (8), August 1972, pp. 899–901.

[Russell78] Russell (R.M.). – The Cray-1 Computer System. *Communication of the ACM*, vol. 21 (1), January 1978, pp. 63–72.

[Sastry et al.75] Sastry (K.V.) et Kain (R.Y.). – Of the Performance of Certain Multiprocessor Computer Organizations. *IEEE Transactions on Computers*, vol. C-24 (11), November 1975, pp. 1066–1074.

[Sauer et al.80] Sauer (C.H.) et Chandy (K.M.). – Approximate Solution of Queueing Models. *IEEE Computer*, vol. 13 (4), April 1980, pp. 25–32.

[Schonauer87] Schonauer (W.). – *Scientific Computing on Vector Computers - Special Topics in Supercomputing 2.* – (Amsterdam), North-Holland, 1987.

[Sethi et al.79] Sethi (A.S.) et N.Deo. – Interference in Multiprocessor Sustems with Localized Memory Access Probabilities. *IEEE Transactions on Computers*, vol. C-28 (2), February 1979, pp. 157–163.

[Shapiro78] Shapiro (H.D.). – Theoretical Limitations on the Effecient Use of Parallel Memories. *IEEE Transactions on Computers*, vol. C-27 (5), May 1978, pp. 421–429.

[Simmons et al.91a] Simmons (M.L.) et Wasserman (H.J.). – C-90 Performance Study. *Vector Register*, vol. 4 (3), November 1991, pp. 3–7.

[Simmons et al.91b] Simmons (M.L.), Wasserman (H.J.), Lubeck (O.M.), Eoyang (C.), Harada (H.), Ishiguro (M.) et Mendez (R.). – A Performance Comparison of Three Supercomputers : Fujitsu VP2600, NEC SX-3, and Cray Y-MP. *In : Proc. of Supercomputing*, pp. 150–157. – Albuquerque, New Mexico (18-22), November 1991.

[Skinner et al.69] Skinner (C.H.) et Asher (J.R.). – Effects of Storage Contention on System performance. *IBM Journal Res. Develop.*, 1969.

[Smilauer85] Smilauer (B.). – General Model for Memory Interference in Multiprocessors and Mean Value Analysis. *IEEE Transactions on Computers*, vol. C-34 (6), June 1985, pp. 506–522.

[Spragins80] Spragins (J.). – Analytical Queueing Models. *IEEE Computer*, vol. 13 (4), April 1980, pp. 9–11.

[Tang et al.89] Tang (P.) et Mendez (R.H.). – Memory Conflict and Machine Performance. *In : Proc. Supercomputing*, pp. 826–831. – Reno, Nevada, 1989.

[Wasserman et al.88] Wasserman (H.J.), Simmons (M.L.) et Lubeck (O.M.). – The Performance of Minisupercomputers : Alliant FX/8, Convex C-1, and SCS-40. *Parallel Computing*, vol. 8 (1-3), October 1988, pp. 185–293.

[Weiss89] Weiss (S.). – An Aperiodic Storage Scheme to Reduce Memory Conflicts in Vector Processors. *In : The Annual International Symposium on Computer Architecture*, pp. 380–387. – (28(5)–1(6)), Jerusalem, Israel, May 1989.

[Weiss91] Weiss (M.). – Strip-Mining on SIMD Architectures. *In : Proc. of Int'l. Conference on supercomputing.* pp. 234–243. – Cologne, Germany.

[Wijshoff et al.85] Wijshoff (H.A.G.) et Leeuwen (J.V.). – The Structure of Periodic Storage Schemes for Parallel memories. *IEEE Transactions on Computers*, vol. C-34 (6), June 1985, pp. 501–505.

[Wijshoff et al.87] Wijshoff (H.A.G.) et Leeuwen (J. Van). – On Linear Skewing Schemes and d-Ordered Vectors. *IEEE Transactions on Computers*, vol. C-36 (2), February 1987, pp. 233–239.

[Yen et al.82] Yen (D.W.L.), Patel (J.H.) et davidson (E.S.). – Memory Interference in Synchronous Multiprocessor Systems. *IEEE Transactions on Computers*, vol. C-31 (12), April 1982, pp. 1116–1121.