Conception et analyse d'algorithmes numériques parallèles

Denis Delesalle

THÈSE

présentée par

DENIS DELESALLE

pour obtenir le grade de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrété ministériel du 30 Mars 1992)

(Spécialité : Mathématiques Appliquées)

Conception et analyse d'algorithmes numériques parallèles : Réalisations sur la Connection Machine

Date de soutenance : 12 Février 1993

Composition du jury :	Président :	P. WITOMSKI
	Rapporteurs :	W. JALBY
		J. ROMAN
	Examinateurs :	Y. ROBERT
		D. TRYSTRAM

Thèse préparée au sein du
LABORATOIRE DE MODÉLISATION ET DE CALCUL

A Isa sans qui tout cela n'aurait peut-être pas existé,

et à toute ma famille.

Remerciements

Je tiens à exprimer mes remerciements aux membres du jury :

A Patrick Witomsky pour l'honneur qu'il me fait de bien vouloir présider ce jury.
A Jean Roman et William Jalby pour leurs conseils et leurs remarques qui m'ont permis d'améliorer la qualité de ce manuscrit.
A Yves Robert qui me fait l'honneur de faire partie de mon jury, lui qui a été une des premières personnes à me faire connaître le parallélisme et dont certains cours hantent encore les pages de cette thèse.
Et enfin Denis Trystram, plus un ami qu'un « chef », sans qui toutes les pages qui suivent n'auraient jamais vu le jour.

A Isabelle, ma fiancée, membre de l'équipe de Calcul Formel, qui a du en supporter beaucoup pour que ce document arrive à ce qu'il est maintenant. Qu'on se le dise dans les deux équipes, ils sont plutôt sympas les gens d'en face !!
A ma maman, car je vous assure que relire 160 pages d'une thèse alors qu'elle hait l'informatique ...
A Didier, pour les heures passées à la Kfet à discuter autour d'un café et d'une cigarette en ce qui le concerne. Que ces jumelles lui apportent tout le bonheur possible, ainsi qu'à sa femme Elisabeth.
Aux deux co-occupants de mon bureau, Jean-Yves et Laurent. J'ai usé le premier qui a préféré s'exiler à Paris. Le deuxième est encore là mais il est arrivé plus tard. Qu'ils soient ici remerciés pour leur patience, mais surtout qu'ils préparent leurs plans car le juge arbitre va bientôt être libre.
A Philippe et Pascal, les vieux de la vieille dans l'équipe des Mickey. Qu'ils restent ce qu'ils sont, leur bonne humeur fait vivre ce laboratoire.
A Christophe, un des nouveaux dans l'équipe pour avoir supporté deux présoutenances sans broncher. A Gilles, le seul varois comme moi, dans la grisaille Grenobloise. Et à tous les autres comme Hervé, Fred, Jean-Marc, tous les Laurent, qu'ils ne se sentent pas exclus si leur nom n'apparaît pas. Un laboratoire est un ensemble où j'ai passé plus de trois ans de ma vie et je ne peux pas citer tout le monde.
Un merci spécial à Angèle, qui gère tous nos petits soucis administratifs avec la plus grande gentillesse et toujours un grand sourire.

Je finirais par une pensée pour Nathalie, en lui souhaitant que tous ces soucis ne soient que passagers. Qu'elle trouve ici exprimée toute notre amitié à Isabelle et à moi.

Chapitre I

Introduction

1. Le parallélisme en général.

L'accroissement de la taille des problèmes traités (volume de données et complexité) conduit à des exigences au niveau de la rapidité de calcul de plus en plus importantes. L'emploi du parallélisme semble une réponse possible tant du point de vue des performances que de celui du coût financier. En effet, il fournit un moyen simple pour accélérer les calculs tout en conservant un rapport constant entre la vitesse de calcul et le coût. L'idée du calcul parallèle est simple : différentes parties indépendantes d'un algorithme peuvent être résolues simultanément si l'on dispose de composants de traitement distincts. Dans la pratique, on se heurte à des problèmes dont principalement les échanges d'informations entre les unités de traitement.
Un important effort de modélisation a été fait dans le domaine des algorithmes parallèles. Des concepts d'optimalité et d'accélération par rapport au séquentiel ont été développés. Le modèle le plus répandu est le modèle PRAM[1] [69]. Le but de cette étude est de trouver le meilleur temps d'exécution d'un algorithme sur un nombre polynomial de processeurs (classe *NC*). Dans la réalité, les machines ne disposent que d'un nombre limité de processeurs. On doit donc modifier en conséquence l'algorithme pour qu'il s'adapte au mieux aux différentes architectures.

Selon Edelman [47], il y a dizaine d'années encore peu de gens s'intéressait au parallélisme sur des machines de grande taille, et certains experts en algèbre linéaire pensaient même que les machines parallèles n'étaient et ne seraient pas

[1]Parallel Random Acces Memory

utiles pour résoudre les très grands systèmes denses comme par exemple la décomposition LU d'un système de taille supérieure à 10000 dont le nombre les éléments nuls est négligeable . Les performances obtenues sur certaines machines actuelles avec jeux d'essais de LINPACK [44], ainsi que les records actuels mettent en doute leurs opinions. Le plus gros problème actuellement traité porte sur la résolution LU d'un système non-structuré de taille n=75264 sur un iPSC/860 à 128 processeurs double précision 64-bits en 64 heures. La course au record continue toujours.... Toujours plus vite, mais ne demande-t-on pas tout trop vite ?

2. Les différentes approches.

Selon Edelman [47], il existe trois approches différentes pour résoudre les problèmes de grandes tailles. Cette démarche n'est pas propre au parallélisme, on la retrouve aussi bien en séquentiel que dans d'autres domaines différents de l'informatique.

La course à la puissance.

La *première*, sûrement la plus connue, consiste à obtenir les meilleurs performances possibles sur une machine bien précise. Plus communément, on devrait dire: Combien de "machaflops" peut-on atteindre sur cette machine pour un problème ciblé ? Cette recherche de puissance est un des maîtres mots des scientifiques, surtout pour ceux qui emploient le parallélisme comme outil d'accélération. Cette démarche est un challenge que beaucoup de spécialistes d'autres domaines tels que la mécanique, l'astrophysique, etc.. essayent de relever. Une connaissance profonde de la machine et de ses avantages assure de bons résultats, mais en contrepartie la méthode finale n'est véritablement valable que sur cette machine.

Les boites noires.

Une solution est alors de se tourner vers une *deuxième approche* et d'utiliser une bibliothèque telle que LAPACK [44, 30] développée à Oak Ridge et l'université du Tenessee. Là aussi les performances sont importantes, mais l'idée directrice est de fournir à l'utilisateur une boite noire qui lui permettra de réaliser n'importe quelle procédure d'algèbre linéaire. Pour illustrer la philosophie de LAPACK, on peut citer la comparaison entre une voiture de tourisme de monsieur tout-le-monde et une Ferrari pour la première approche. D'un coté, on peut tout faire mais à vitesse modérée même les chemins de terre, de l'autre on va très vite seulement dans des conditions idéales, mais alors quand on se trouve hors du circuit toutes les performances chutent rapidement... Ecrit en Fortran 77, LAPACK avec le

nouveau Fortran HPF (Hight Performance Fortran) ou Fortan 90 laisse présager de meilleures performances.

La structures des données.

La *dernière approche* plus technique, trouve son public auprès des mathématiciens appliqués. Elle se base sur la constatation suivante : chaque grand système a sa propre structure. Ainsi, il semble intéressant d'effectuer une étude spécifique à chaque type de problème afin de tenir compte des propriétés de la matrice au lieu de la considérer comme un banal tableau de nombres. Cette approche est déjà employée dans le cas des matrices bande-diagonale [97], Tœplitz, Vandermonde, ou plus simplement orthogonale ou symétrique. On se retrouve alors confronté au même type de problème que lors de la première approche, rien ne nous garantit que la solution pour un problème donné, sera la même pour le suivant. Pour continuer l'analogie du paragraphe précédent, on peut comparer cette dernière méthodologie à un garage de préparateurs automobiles qui va modifier la voiture de monsieur tout-le-monde afin qu'elle s'adapte au mieux de ces performances dans le milieu où elle va devoir évoluer.

3. Quel type de parallélisme et de machine utilisés ?

Le parallélisme est une notion que l'on retrouve depuis longtemps en informatique, pourtant les première machines ne sont disponibles que depuis 20 ans. Et ça ne fait qu'une dizaine d'années que de nombreux constructeurs inondent le marché en proposant différents types de machines parallèles comme l'Alliant Campus, le Cray MPP, la Concerto, le nCube, la Paragon, la CM2, la Maspar, etc.... Cette liste ne cite que certaines machines existantes. Une présentation plus exhaustive se trouve dans [2]. L'objectif technique avoué par les constructeurs américains fortement soutenus par leur gouvernement, est d'atteindre l'objectif 3T : le Tera[2] Flops de puissance de résolution, le Tera Octet de mémoire et le Tera Octet/Seconde pour le transfert de données entre processeurs. Les trois projets principaux sont celui d'Intel avec le projet Touchstone, celui de Thinking Machines Corporation (T.M.C.) avec la Connection Machine 5, et celui de Cray avec le MPP.
Le parallélisme peut prendre beaucoup de formes différentes. La plus simple est sans aucun doute de travailler de manière synchrone en effectuant la même instruction sur des données multiples. On parle de parallélisme de type S.I.M.D. (de l'anglais Single Instruction Multiple Data) par opposition au mode M.I.M.D. (Multiple Instruction Multiple Data). C'est le cas de la Connection Machine.

[2]Un Tera est égal à mille milliard soit 10^{12}.

Conçue par T.M.C., elle est une des premières machines massivement parallèles, et surtout la précurseur de la nouvelle CM5. De nombreux travaux ont été effectués sur machine dans des domaines aussi différents tels que la physique [105], l'algorithmique numérique [86, 13, 65], la biologie [68, 74], les automates cellulaires [12, 104], la dynamique moléculaire [54], ou encore plus connus comme pour les problèmes de vision et de visualisation [108, 99].

Ce type de parallélisme semble avec son grain très fin, être intéressant pour l'étude des problèmes d'algèbre linéaire. Nous nous proposons ici de tester les possibilités mais surtout de mettre en évidence les limites du mode S.I.M.D. en utilisant toutes les approches décrites ci-dessus dans le cadre de la programmation de ce type de problème. Le choix de la Connection Machine est quant à lui plus pratique que théorique. Ce travail a commencé avec l'installation de la première machine installée en Europe, plus précisément sur le site de l'E.T.C.A. à Paris. Et il y a 4ans, elle était la seule machine S.I.M.D. présentée comme généraliste, c'est-à-dire pouvant résoudre tout type de problème, contrairement à d'autres spécialisées dans un domaine comme par exemple le traitement d'images avec de DAPP.

4. Plan.

Le chapitre II est un parfait exemple de la *première approche* qui consiste cherche à atteindre les performances maximale. En effet, nous montrons les deux étapes importantes que sont le placement des données et l'optimisation des communications, pour obtenir des résultats optimaux pour des applications. Nous développons les résultats sur l'exemple du produit matrice-vecteur itéré, brique de base de beaucoup d'algorithmes d'algèbre linéaire. Nous présentons de plus un nouvel algorithme de construction d'arbre équilibré, qui permet d'écrire des procédures de communication optimales. Le chapitre III est entièrement consacré à l'expérimentation pratique sur la Connection Machine, des résultats théoriques développés dans le chapitre précédent. Les performances estimées à partir des travaux effectués sur une CM2 avec 8K-processeurs sont de l'ordre du Gigaflops (milliard d'opérations flottantes par seconde). Ce niveau de performances démontre que l'on pourra, dans un avenir proche, résoudre des applications numériques en vraie grandeur, comme la résolution numérique du problème posé par la météorologie. Ces résultats sont obtenus grâce à la création de procédures de communication basées sur un nouvel algorithme de construction d'arbres équilibrés, et d'un placement de données judicieux.

Néanmoins tout n'est pas toujours aussi facile même quand on connaît parfaitement l'architecture de la machine. Dans le chapitre IV par l'étude de deux algorithmes de résolution triangulaire, nous faisons ressortir les inconvénients et

les contraintes du mode synchrone lors de la programmation des méthodes directes.

Dans l'ensemble des chapitres et plus particulièrement au chapitre V, nous nous sommes intéressés à la *deuxième approche*. En effet, l'étude algorithmique de chaque méthode a consisté à décomposer l'algorithme en une suite d'instructions appartenant aux différents niveaux BLAS [40] de la bibliothèque LINPACK. Cette décomposition en instructions de faible volume (si possible $\mathcal{O}(n)$, $\mathcal{O}(n^2)$) permet une exploitation efficace du mode synchrone et de la faible granularité de la Connection Machine.
Dans le chapitre V, nous montrons que certaines méthodes d'algèbre linéaire sont efficaces en séquentiel mais deviennent parfois trop coûteuses lorsqu'on les transfère sur des machines parallèles. Alors il vaut mieux pour certaines d'entre elles, les remplacer par des méthodes ayant une convergence théorique équivalente voire plus lente, mais qui s'avéreront être plus rapides en parallèle.
La parallélisation du Gradient Conjugué met en évidence le coût prohibitif des opérations de réduction pour le calcul des paramètres optimaux à chaque itération. Après l'analyse de la méthode dans laquelle on remplace les paramètres du Gradient Conjugué par des constantes (méthode de Richardson du second ordre), nous comparons les deux méthodes par des expérimentations séquentielles puis parallèle sur la CM2.
Au cours de ce même chapitre, nous étudions un cas particulier de matrice structurée avec la matrice issue du problème de discrétisation du Laplacien. Nous montrons ici clairement le gain important apporté par la troisième approche: gain en mémoire mais surtout en performances.

Le chapitre VI est entièrement consacré au problème de communication générale avec la parallélisation de l'algorithme de Burg (utilisé pour résoudre le problème de la prédiction linéaire d'un signal). L'étude et l'implémentation parallèle de cet algorithme ont été réalisées sur de nombreuses machines existantes qu'elles soient massivement parallèles comme la Maspar ou le NASA/Goodyear MPP, ou M.I.M.D. comme l'iPSC/2, le Cray X-MP/48 ou encore le nCube. Leurs conclusions montrent une certaine supériorité du calcul massivement parallèle. Mais, en général les analyses ne portent que sur le calcul des coefficients de réflexion. Nous examinons plus particulièrement les calculs des coefficients autorégressifs qui font appel à des permutations. Ces permutations consistent en un mouvement structuré de données ayant pour conséquence un nombre plus ou moins important de communications. Or, les communications sont un des problèmes cruciaux en raison d'une perte de temps relativement importante. En conclusion de cette étude, nous proposons un algorithme qui permet d'obtenir à la fois les coefficients de réflexion et ceux d'autorégression sans coût supplémentaire.

L'annexe est entièrement consacrée à la présentation de la Connection Machine:

architecture, communication et langage. Elle est rédigée à partir de [35], document distribué à l'initiative du C.N.R.S. et qui a connu une large diffusion. Des ajouts ont été apportés en particulier sur le langage CMIS utilisé pour obtenir les performances maximales, et sur un outil d'optimisation des communications : le Communication Compiler.

Chapitre II

Comment programmer efficacement

Application au produit Matrice-Vecteur itéré

L'écriture de programmes efficaces sur les machines parallèles à mémoire distribuée est liée au problème l'optimisation des communications entre processeurs. Etant donné un algorithme, il va en effet falloir organiser la rencontre des données. Ceci consiste en un placement des données duquel on déduit les schémas de communication à utiliser. Pour implanter un algorithme sur n'importe quelle machine, il est donc nécessaire de répondre aux deux questions suivantes : « Comment placer les données pour avoir le moins de communications possible ? », « Comment réaliser au mieux un schéma de communication particulier ? ».

Nous allons dans ce chapitre, essayer de répondre aux deux questions précédentes dans le cas de l'implantation du produit matrice-vecteur itéré. Ce produit matrice-vecteur est utilisé dans des applications telles que les réseaux de neurones [53] ou la résolution de grands systèmes linéaires (méthodes itératives) [55, 62]. Dans une première partie, nous étudions les placements usuels que sont les stockages *ligne* et *colonne*, et proposons une version intermédiaire qui permet de minimiser le nombre d'étapes de communication. Puis dans une deuxième partie, nous démontrons des résultats d'optimalité concernant deux schémas fondamentaux de communication (l'échange total et l'échange total personnalisé avec accumulation) ; schémas nécessaires lors de la réalisation du produit matrice-vecteur. Le modèle de machine parallèle considéré est une machine multiprocesseur S.I.M.D. à mémoire distribuée dont le réseau d'interconnexion est un hypercube. La méthodologie employé tout au long de ce chapitre est générale, et permet une extension

des résultats sur les grilles.

1. Le placement des données

Dans cette première partie, nous utiliserons les procédures de communication en les descrivant simplement. Des définition plus précises se trouvent dans la deuxième partie (Cf 2) qui leur est entièrement consacrée.

1.1. Présentation du problème

Soit A une matrice $n \times n$ et x un vecteur de IR^n. On se propose de calculer, sur une machine multiprocesseurs à mémoire distribuée, le produit de A par x de manière à pouvoir itérer ce produit, c'est-à-dire calculer le produit de A par Ax et ainsi de suite. Ce choix du produit itéré impose une contrainte majeure: le résultat du produit doit être stocké de la même façon que le vecteur initial.

On utilise la notation usuelle, a_{ij} l'élément de la matrice A en ligne i et colonne j. En séquentiel, on peut écrire l'algorithme sous la forme suivante:

Algorithme II.1 : Produit Matrice Vecteur Séquentiel

```
Pour tout i
    y_i ← 0
    Pour tout j
        yi ← y_i + a_ij × x_j

Pour tout i
    x_i ← y_i
```

Sur une machine multiprocesseurs à mémoire distribuée massivement parallèle, la valeur de chaque élément de x (et de A) n'est connue que par un seul processeur (on s'interdit les duplications de données). Les processeurs vont donc devoir échanger des informations.
On note $p(i)$ le processeur responsable de la i-ème composante des vecteurs de I R^n et $P(i,j)$ le processeur sur lequel est alloué l'élément a_{ij}. Comme nous voulons enchaîner le produit matrice-vecteur, le processeur $p(i)$ doit récupérer à la fin d'un calcul de produit, la i-ème composante du résultat.

Nous allons présenter les différentes méthodes usuelles parallèles, c'est-à-dire les

deux placements naturels de A : le placement en lignes et en colonnes (voir figure II.1), correspondant aux versions DOT et AXPY des procédures fondamentales d'algèbre linéaire (BLAS) [9] [48].

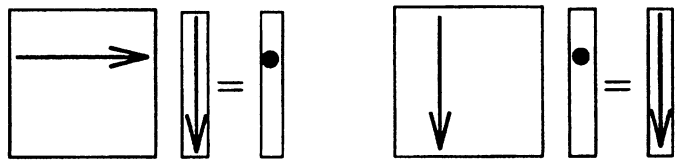


Figure II.1. Comparaison des deux méthodes usuelles.

1.2. Version par lignes

Si A est stockée en lignes, c'est-à-dire si $\forall\, j\ P(i,j) = p(i)$, alors toute une ligne appartient à un seul processeur. Il est donc nécessaire de faire parvenir sur ce processeur toutes les composantes du vecteur pour pouvoir effectuer sur place le produit scalaire. Comme tous les processeurs contiennent au moins une ligne, ils doivent tous recevoir l'ensemble du vecteur. Il n'y a donc qu'une seul étape de communication à effectuer. Elle consiste en un échange total, c'est-à-dire que chaque processeur doit envoyer un même message à tous les autres.
On représente schématiquement le principe par la figure II.2. Le processeur $p(i)$ qui contient x_i doit recevoir tous les x_j $(j \neq i)$ pour pouvoir effectuer de façon locale le calcul $\sum_{i=1}^{n} a_{ij}x_j$. En conséquence le processeur $p(j)$ doit envoyer x_j (i.e. un seul message identique à tous les autres), alors que $p(i)$ reçoit $n-1$ messages. Le calcul, la multiplication suivie de la somme, se fait alors sur place et est symbolisé sur la figure par $\times$.

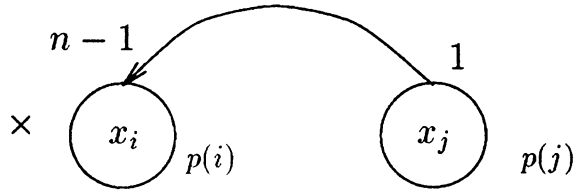


Figure II.2. Produit matrice-vecteur en ligne.

L'algorithme pour l'ensemble des processeurs s'écrit :

Algorithme II.2 : Version Ligne

Echange total de x.
Produit scalaire de la ligne par le vecteur x.

1.3. Version par colonnes

Si A est stockée en colonnes, c'est-à-dire si $\forall i\ P(i,j) = p(j)$, alors on effectue tout d'abord sur chaque processeur, la multiplication de la colonne par la coordonnée x_j. Et tous les $a_{ij} \times x_j$ sont connus. Il faut alors les envoyer vers tous les autres processeurs de façon personnalisée: $a_{ij} \times x_j$ doit parvenir au processeur $p(i)$ qui contient x_i, et $a_{kj} \times x_j$ au processeur $p(k)$ qui contient x_k. Pour qu'à la fin le processeur $p(i)$ connaisse le nouvel $x_i = \sum_{j=1}^{n} a_{ij}x_j$, les différents produits s'accumulent de proche en proche. Il n'y a qu'une étape de communication. Elle consiste en un échange total personnalisé avec accumulation, c'est-à-dire que chaque processeur envoie un message personnalisé à chacun des autres processeurs, tous les messages destinés à un même processeur étant combinables (ici par addition).
Si l'on utilise la même représentation que précédemment,on observe que:

- la multiplication symbolisée par $\times$ est tout d'abord effectuée sur le processeur $P(j)$
- Tous les processeurs ont besoin d'un $a_{ij}x_j$ différent, et le processeur $p(j)$ va devoir envoyer $n-1$ messages différents. Par exemple, le message $a_{ij}x_j$ doit parvenir au processeur $p(i)$.
- Le processeur $p(i)$ recevra un seul message contenant $\sum_{k=1}^{n} a_{ik}x_k - a_{ii}x_i$. L'accumulation se fait au fur et à mesure.

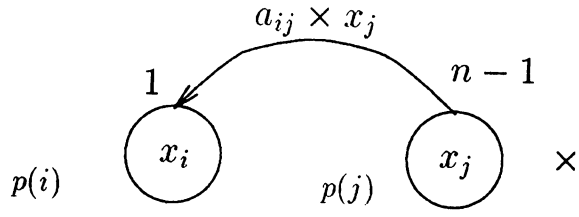


Figure II.3. produit matrice-vecteur en colonne.

L'algorithme pour l'ensemble des processeurs s'écrit:

Algorithme II.3 : Version Colonne

Multiplication de la colonne par la composante connue de x.
Echange total personnalisé avec accumulation du vecteur précédemment obtenu.

Propriété 1 : Si on compare les deux méthodes, on observe les caractéristiques suivantes :

- Elles ont le même volume de calcul.
- Et surtout le même nombre d'étapes de communication comme nous le verrons dans la section suivante.

1.4. Version intermédiaire (où l'on déduit un meilleur placement)

Les deux différents schémas de communication utilisés par les deux méthodes peuvent être optimaux, mais l'implémentation du calcul matrice-vecteur qui les utilisent ne l'est pas forcément ! En effet, si la dimension de l'hypercube augmente, alors le nombre d'étapes de communication augmente aussi. Aussi dès que cette dimension est supérieure à 4, le nombre d'étapes nécessaire pour effectuer l'un ou l'autre des deux schémas, devient supérieur au diamètre de l'hypercube. Ce dernier représente le nombre minimum de communication nécessaire pour effectuer le produit matrice-vecteur.
Le problème que nous nous posons maintenant est le suivant :
Existe-t-il un autre placement de données qui permettent de minimiser le volume global de communications, c'est-à-dire avoir un nombre d'étapes de communication qui soit le plus proche possible du diamètre.

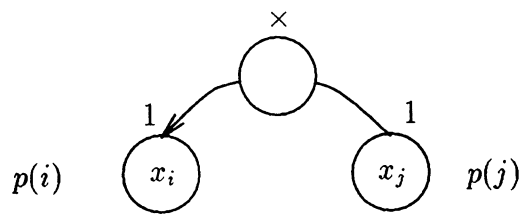


Figure II.4. Existe-t-il un meilleur placement ?

Une autre formulation du problème revient à se poser la question : Existe-t-il un processeur qui serve de station intermédiaire, tel que dans une première phase on effectue un échange total, puis un calcul local pour finir par un échange total

personnalisé avec accumulation.
En faisant référence à la figure II.4, on voit bien l'intérêt de ce processeur qui permet de diminuer le volume global de communication. On a un message au départ et un seul à l'arrivée. Sur le processeur intermédiaire, un petit produit matrice-vecteur est effectué. Cette figure montre ce qui se passe dans le cas d'un processeur par rapport à un autre. Si on regarde ce qui se passe pour un processeur par rapport à l'ensemble des autres, la diffusion va se faire dans toutes les directions, et x_j va être diffusé vers un groupe de processeurs que nous appellerons « voisinage ». Ce voisinage sera le même pour tous les processeurs à une translation près.

Le calcul du produit matrice-vecteur pour l'ensemble des processeurs s'écrit sous la forme :

Algorithme II.4 : Version Intermédiaire

Diffusion partielle et parallèle de x sur un voisinage V.
Produit matrice-vecteur du bloc connu avec la partie de x que l'on vient de recevoir.
Accumulation partielle et parallèle du vecteur obtenu ci-dessus sur un voisinage W.

Cherchons maintenant une condition nécessaire et suffisante pour que cet algorithme effectue un produit matrice-vecteur.

Soient deux processeurs p et q respectivement responsables des i-ème et j-ème coordonnées du vecteur x. Il est alors nécessaire et suffisant qu'il existe un et un seul processeur appartenant simultanément à $p + W$ et $q + V$ et contenant l'élément a_{ij} de la matrice A.

Pour que la condition précédente soit vérifiée, il suffit de définir une structure croisée à partir des deux voisinages. La solution est de faire le produit cartésien.

Définition 1 : On appelle *produit cartésien* (ou somme cartésienne) de deux graphes V et W noté $\otimes$, le graphe dont les sommets sont tous des couples (u, v) où u est un sommet de V et v un sommet de W et les arêtes sont définies par : deux sommets (u, v) et (x, y) sont reliés par une arête si et seulement si $u = x$ et $[v, y]$ est une arête du graphe W, ou $v = y$ et $[u, x]$ est une arête de V [8].

Dans la pratique, on est confronté au problème inverse : la topologie finale est connue. Comme la condition est nécessaire et suffisante, il suffit d'inverser et de découper la topologie en un produit cartésien de deux sous-topologies plus simples.

Dans le cas où la topologie utilisée est un hypercube noté H_n, une solution évidente est de le décomposer en un produit de deux sous-hypercubes [2]. De plus pour que cette décomposition minimise le coût en nombre d'étapes de communication, chaque sous hypercube doit être de dimension minimum.
Donc $\forall n$, la meilleur décomposition est la suivante:

$$H_n = H_{\lceil \frac{n}{2} \rceil} \otimes H_{\lfloor \frac{n}{2} \rfloor}.$$

Théorème 1 : Cette décomposition est optimale en nombre d'étapes de communication, pour tous les hypercubes de dimension inférieure ou égale à 8.

La démonstration est évidente à partir de la propriété suivante:

Propriété 2 : Si $H = G \otimes G'$ alors Diamètre(H) est égal à Diamètre(G) + Diamètre(G').

Or, sur tous les hypercubes de dimension inférieure ou égale à 4, les deux schémas de communication s'effectuent en un nombre d'étapes égales au diamètre. Par conséquent, pour toutes les décompositions jusqu'à la dimension 8, le nombre de communication pour réaliser le produit matrice-vecteur est aussi égale au diamètre, ce qui est le mieux que l'on puisse obtenir.

Remarque 1 : Pour les hypercubes de dimension supérieure à 8, la décomposition en un produit de deux sous graphes ne permet plus d'atteindre le diamètre mais le nombre global d'étapes de communication est tout de même un inférieur aux méthodes lignes et colonnes.

La même démarche peut être appliquée sur d'autres topologies. Par exemple, un tore de taille $n \times m$ peut se décomposer en un produit cartésien de deux anneaux de taille n et m. Dans ce cas encore, la méthode est optimale. Une présentation est faite dans [53]. Ce travail a été effectué par H. Frydlender appliqué à la programmation de l'algorithme de rétropropagation du gradient, sur le MegaNode (machine M.I.M.D. à 128 transputers configurée en tore). Dans ce cas, on atteint toujours le diamètre.

1.5. Influence sur le placement des données

On en déduit une nouvelle répartition des données que nous allons décrire dans cette section.
Posons les hypothèse suivantes:

- l'échange total se fait le long du premier voisinage défini par le produit cartésien
- l'échange total personnalisé avec accumulation se fait le long du deuxième voisinage.
- on a P processeurs

La représentation du produit cartésien des deux voisinages consiste en une grille de taille *la taille du premier voisinage* $\times$ *la taille du second voisinage*. Sur la figure II.5, les hypothèses sont équivalentes à faire l'échange total sur les colonnes et l'échange total personnalisé avec accumulation le long des lignes.

Soit x_i un élément du vecteur x, nous allons étudié les influences de son placement sur un processeur $p(i)$ sur celui des éléments a_{kl} de la matrice.

- x_i doit rencontrer tous les éléments de la colonne i de la matrice pour permettre le calcul de $a_{ki} \times x_i$. Il faut donc faire une diffusion de x_i. En raison de notre hypothèse initiale sur le sens de l'échange total, les éléments a_{ki} doivent se trouver sur la même colonne de processeurs que $p(i)$.
- De même, lorsque l'on calcule la somme $\sum_{l=1}^{n} a_{il} \times x_l$, il faut faire une réduction le long de la ligne i, c'est-à-dire effectuer l'échange total personnalisé avec accumulation. En raison de nos hypothèses, les éléments a_{il}, doivent appartenir á la même ligne de processeur que $p(i)$.

la figure II.5 schématise les différentes influences sur le placement des données.

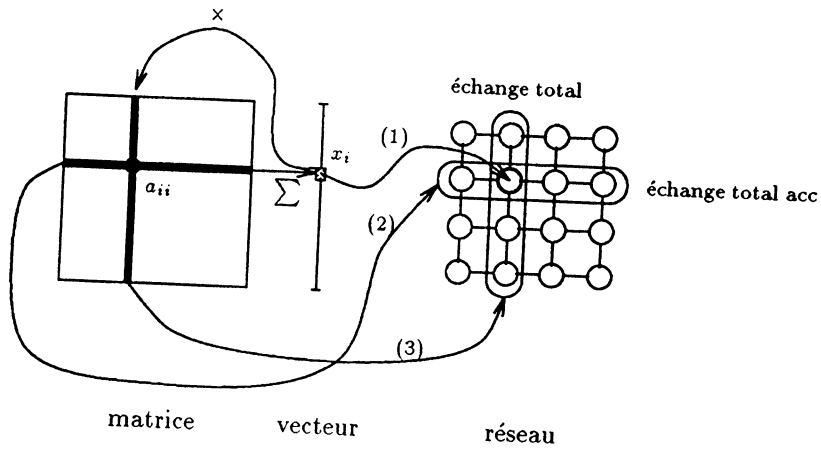


Figure II.5. (1) Placement de x_i sur le processeur $p(i)$. (2) a_{ki} placé sur la même colonne de processeurs que $p(i)$. (3) a_{il} placé sur la même ligne de processeurs que $p(i)$.

Le choix du placement exact de l'élément dans la colonne (respectivement la ligne) de processeurs de l'élément a_{ki} (resp. a_{il}) est fonction du placement de x_k (resp. x_l).

Il est évident que pour être optimal, le stockage doit être équilibré ; les x_i doivent donc être répartis de façon équitable.

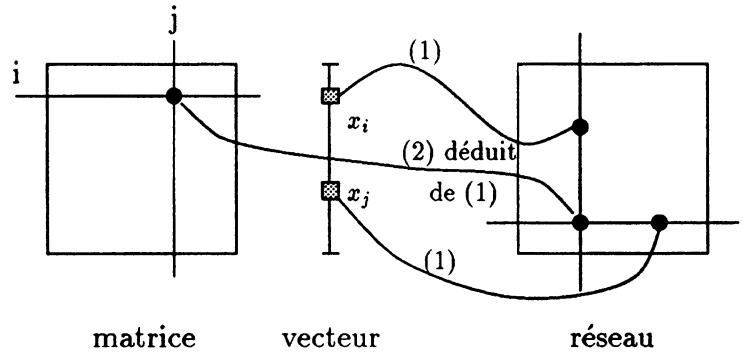


Figure II.6. Placement d'un élément a_{ij} de la matrice en fonction de celui du vecteur.

Chaque processeurs a $\frac{n}{P}$ éléments du vecteur (on suppose par la suite que ce nombre est un entier). Montrons qu'alors chaque processeur a $\frac{n^2}{P}$ éléments de la matrice.

1/ Pour chaque processeur de la même colonne
Pour chaque processeur de la même ligne
on a $(\frac{n}{P})^2$ éléments de la matrice.

2/ Soit l le nombre de lignes de processeurs, et c le nombre de colonnes, alors on a $l \times c = P$ couples (l_i, c_j) de processeurs différents possibles.

De 1/ et 2/ on en déduit que le nombre d'éléments de la matrice sur un processeur représenté par un couple(l_i, c_j) est de $(\frac{n}{P})^2$ divisé par $l \times c$ soit $\frac{n^2}{P}$.

Au niveau du stockage des données a_{ij} de la matrice, on s'aperçoit qu'on se ramène à un stockage de type par bloc, mais avec des blocs non groupés en un seul endroit.

2. Optimisation des schémas de communication

Après avoir décrit, un placement de données qui permet de diminuer le nombre d'étapes de communication, il convient de regarder de plus près la mise en œuvre des deux schémas de communication nécessaires à la réalisation du produit matrice-vecteur: l'échange total et l'échange total personnalisé avec accumulation.

2.1. Définitions des schémas de communication.

De nombreux travaux ont été réalisés pour déterminer des chemins optimaux sur différentes architectures pour effectuer plusieurs types de communication comme la diffusion ou l'échange total [66, 87, 102].
Les schémas fondamentaux que nous nous proposons d'étudier maintenant sont la diffusion et l'accumulation de messages.

Définition 2 : Dans le cas d'une diffusion (au sens large du terme), les messages sont initialement connus par un seul processeur. Puis, ils sont dupliqués de proche en proche, et à la fin tous les processeurs concernés connaissent les messages qu'ils ont à connaître.

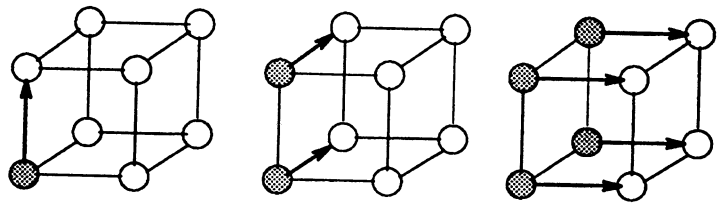


Figure II.7. Exemple de diffusion de la valeur du processeur gris à tous les autres sur un 3-cube. Première étape diffusion sur la première dimension, deuxième étape diffusion sur la deuxième dimension, le processeur ayant reçu le message à l'étape précédente diffuse également. Troisième étape, on fait comme l'étape précédente sur la troisième dimension.

Comme cas particulier de la diffusion nous étudierons plus particulièrement l'échange total [51, 66, 94, 102].

Définition 3 : Dans le cas de l'échange total, tous les processeurs ont initialement un message qu'ils destinent à tous les autres processeurs et doivent finalement avoir reçu un message de tous les autres processeurs soit $P - 1$ messages sur un réseau de P processeurs.

Il est possible de limiter la diffusion à un ensemble strict des processeurs; nous parlerons alors d'échange partiel.

Définition 4 : Lors d'une accumulation, problème inverse de la diffusion, des messages se trouvant initialement sur des processeurs différents se regroupent de processeur en processeur vers une destination commune. A chaque regroupement, ils sont combinés soit par addition soit par toute autre opération binaire, commutative et associative.

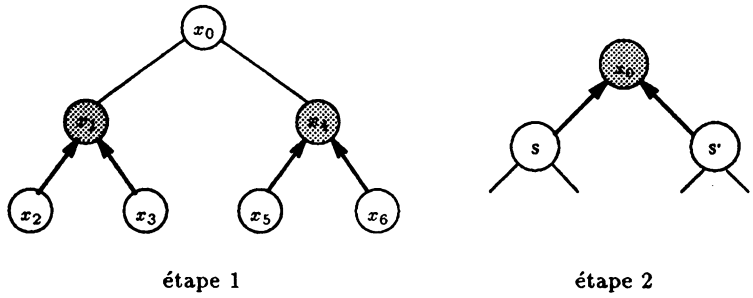


Figure II.8. Exemple d'accumulation par addition sur un arbre binaire à 4 feuilles. A la première étape, chaque feuille envoie sa valeur à son père qui alors peut effectuer sur place l'addition de deux fils avec lui même ($s = x_1 + x_2 + x_3$ et $s' = x_4 + x_5 + x_6$). A l'étape suivante, on fait de même avec les résultats pour monter d'un niveau dans l'arbre.

Comme cas particulier d'accumulation nous étudierons plus spécifiquement l'échange total personnalisé avec accumulation.

Définition 5 : Dans ce cas d'un échange total personnalisé avec accumulation, tous les processeurs ont initialement un message différent à faire parvenir aux autres. A la fin, ils reçoivent la combinaison de tous les messages qui leur étaient destinés soit $P - 1$ messages à combiner pour chaque processeur d'un réseau de P processeurs.

De nombreux résultats concernant de tels schémas de communication ont été récemment développés pour les réseaux de processeurs en mode M.I.M.D. Nous pouvons citer des algorithmes optimaux pour des topologies régulières : échange total sur les grilles [87], diffusion et échange total [66], échange total personnalisé [102], échange total personnalisé sans accumulation [46] sur les hypercubes, diffusion personnalisée sur l'anneau [52].

2.2. Le modèle de machine.

Avant de renter plus avant dans la description des deux schémas, définissons le modèle d'architecture de référence qui va être utilisé maintenant.

Posons les hypothèses suivantes :

- Le réseaux de communication est un hypercube de dimension d ($2^d = P$) disposant chacun d'une mémoire locale et ayant un fonctionnement S.I.M.D. (fonctionnement synchrone).

- Les communications sont parallèles c'est-à-dire plusieurs messages peuvent être émis ou reçu par un même processeur selon les différentes directions.
- Les communications sont bidirectionelles c'est-à-dire qu'un processeur reçoit et émet un message en même temps.
- Les nœuds sont des unités de calculs flottants .

Le schéma élémentaire de communication est l'« échange », sur des flottants. Dans chaque processeur et à chaque direction i est associé un registre, que nous noterons lien(i). A chaque lien de communication entre deux processeurs voisins est associé un couple de registres et, lors d'un échange, tous les liens échangent le contenu des deux registres qui sont à leurs extrémités, comme le montre la figure suivante sur un seul lien :

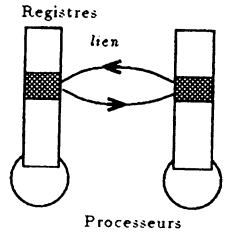


Figure II.9. Echange entre deux processeurs.

2.3. Les communications sur un réseau de processeurs.

De manière générale, lors d'une communication, il est nécessaire de connaître l'origine et la destination des messages. Toute information, pour être conservée, doit être liée au message lorsque celui-ci passe par des processeurs intermédiaires. Lors d'une communication de type "échange total" seule la connaissance de l'origine est nécessaire puisque chaque message est destiné à tous les processeurs. De même, dans le cas d'un "échange total personnalisé avec accumulation" seule la connaissance de la destination est nécessaire puisque les messages pour un même processeur sont combinables.

Sur un réseau quelconque, cette information, d'origine et de destination, peut être résumée en la donnée d'un chemin restant à suivre (pour la destination) et d'un chemin déjà parcouru (pour l'origine). Ces deux informations sont mises à jour au fur et à mesure du déplacement du message de processeur en processeur. Cela revient à travailler avec des adresses relatives.

Le réseau d'interconnexion des processeurs étant un réseau quelconque de degré d, on note $\{1, \cdots, d\}$ l'ensemble des directions.

Définition 6 : Un chemin (au sens de la théorie des graphes) est entièrement défini par la donnée d'un processeur p, l'origine du chemin, et d'une succession $d_1 d_2 \cdots d_k$ de directions à suivre.

Remarque 2 : Le processeur terminal d'un tel chemin ne dépend pas de l'ordre des d_i mais seulement de la parité du nombre d'occurrences de chaque direction dans la succession $d_1 d_2 \cdots d_k$.

En effet, sur un hypercube partant d'un processeur p on atteint aussi bien le même processeur en franchissant la direction i puis la direction j qu'en franchissant d'abord la direction j puis la direction i. De plus, partant d'un processeur p on revient à p si l'on suit deux fois de suite une même direction.

Remarque 3 : Les même remarques et définitions peuvent être faites pour d'autres réseaux comme le tore avec 4 directions, mais c'est plus compliqué à mettre en œuvre à cause des directions opposées deux à deux.

Définition 7 : La notion de chemin ne nous servant qu'à désigner un processeur à partir d'un autre, on appellera maintenant abusivement chemin, noté par l'initial c, tout sous-ensemble de $\{1, \cdots, d\}$.

En pratique, un chemin sera codé par un nombre binaire de d bits, chaque bit correspondant à une direction.

Ces définitions et remarques nous conduisent à énoncer la propriété suivante :

Propriété 3 : Partant d'un processeur p, le processeur atteint en suivant le chemin c_1 puis le chemin c_2 est le même que celui atteint en suivant le chemin $(c_1 \Delta c_2)$. (où Δ désigne la différence symétrique de deux ensembles).

En pratique, on ne calculera que des combinaisons de chemins de la forme $c\Delta\{i\}$; ce qui revient à inverser le i-ème bit de la représentation binaire de c. On passe alors à un nœud voisin par définition de l'hypercube.

2.4. L'échange.

L'opération d'échange permet d'envoyer simultanément un message sur chaque direction. Un processeur peut donc envoyer puis recevoir en un seul échange autant de messages qu'il y a de directions. Le problème est de connaître l'origine des messages reçus par un processeur. Examinons de plus près ce qui se passe au cours d'un échange.

Soit p et q deux processeurs voisins sur l'hypercube et reliés par leur i-ème lien. Supposons que p envoie, sur son i-ème lien, un message ayant déjà parcouru le chemin c_1 et q envoie, au même moment, lui aussi sur son i-ème lien, un message ayant parcouru le chemin c_2. Alors, d'après la propriété 3, à la fin de l'échange, le processeur p a reçu, sur son i-ème lien, un message ayant parcouru le chemin $c_2\Delta\{i\}$ et le processeur q a reçu, sur son i-ème lien, un message ayant parcouru le chemin $c_1\Delta\{i\}$. En conclusion, à la fin d'un échange, nous connaissons exactement l'origine des messages reçus sur les liens.

Notons que le même résultat reste valable si les deux processeurs choisissent les messages émis en fonction du chemin qui leur reste à parcourir et non plus en fonction du chemin déjà parcouru.

Pour nos algorithmes de communication, les hypothèses de notre modèle S.I.M.D. nous permettent de nous placer dans un cas plus simple où tous les processeurs effectuent le même chose. Ce choix impose alors au niveau des messages émis suivant une direction la condition suivante: tous les processeurs émettent un message de même origine relative.
Plus précisément :
soient i une direction de l'hypercube, c un chemin, et *message(c)* le message reçu par tous les processeurs ayant parcouru le chemin c pour leur parvenir. Si tous les processeurs envoient ce message sur la i-ème direction au cours d'un cube-swap, alors tous les processeurs reçoivent à la fin, un message qui a suivi le chemin $c\Delta\{i\}$.

Remarque 4 : nous avons le même résultat si les processeurs choisissent les messages émis en fonction du chemin qui leur reste à parcourir et non plus en fonction du chemin déjà parcouru.

Ecrivons maintenant une procédure de base que nous appellerons *diffusion élémentaire* dont le rôle est de faire connaître un nouveau message à chaque processeur.

Si nous voulons que chaque processeur reçoive sur son i-ème lien un message ayant suivi le chemin c_i pour lui parvenir alors la suite d'instructions à exécuter par chacun est :

lien(i) $\leftarrow$ message($c_i\Delta\{i\}$)
cube-swap
message(c_i) $\leftarrow$ lien(i)

Remarque 5 : Cette diffusion élémentaire a pour résultat celui désiré si et seulement si tous les processeurs ont reçu au préalable le message *message*($c_i\Delta\{i\}$), ayant parcouru le chemin $c_i\Delta\{i\}$ pour leur parvenir.

Remarque 6 : Si l'on se place au niveau des messages, la diffusion élémentaire se traduit par le fait que tous les messages ayant déjà parcouru le chemin $c_i\Delta\{i\}$ se déplacent suivant la direction i.

Propriété 4 : Si, après une suite de diffusions élémentaires, un processeur a reçu un message ayant parcouru un chemin c, alors tous les processeurs ont reçu un message ayant parcouru le même chemin c.

Cette propriété découle de deux faits :

1) Initialement tous les processeurs ne connaissent qu'un seul message (le leur)
2) Tous les messages suivent des chemins parallèles et « identiques » à une translation près.

Elle nous permet, de plus, de suivre l'évolution de l'algorithme sur un seul processeur.

Dans le cas de l'accumulation, on retrouve les même principes. La seule différence est que c'est la destination des messages et non plus leur origine qui est contrôlée. Aussi on désigne ici par *message(c)* tout message devant suivre le chemin c pour arriver à destination.

Si l'on veut que chaque processeur envoie sur la i-ème direction un message ayant encore à parcourir un chemin c_i alors la suite d'instructions à exécuter par chaque processeur est :

$\mathsf{lien}(i) \leftarrow \mathsf{message}(c_i)$
$\mathsf{message}(c_i) \leftarrow 0$
cube-swap
$\mathsf{message}(\ c_i\Delta\{i\}\ ) \leftarrow \mathsf{message}(\ c_i\Delta\{i\}\ ) + \mathsf{lien}(i)$

Nous appellerons *accumulation élémentaire* cette suite d'instructions.

Remarque 7 : L'instruction de remise à zéro est nécessaire a priori car un autre message ayant encore à suivre le chemin c_i pourrait parvenir ultérieurement.

Propriété 5 : Si, après une suite d'accumulations élémentaires, un processeur n'a plus de message devant parcourir le chemin c , alors plus aucun processeur n'a de message ayant à parcourir le chemin c.

Ce sont des successions de telles instructions élémentaires que nous allons utiliser pour nos procédures de communication. La seule différence est que maintenant nous travaillerons sur toutes les directions simultanément.

2.5. Description et optimalité des schémas de communication

Nous allons décrire maintenant un algorithme constructif optimal pour l'échange total sur un hypercube S.I.M.D. (donc a fortiori utilisable en M.I.M.D.) plus simple à mettre en œuvre que celui de [66]. Ce principe s'étend de plus au deuxième schéma de type échange total personnalisé avec accumulation et également aux échanges partiels optimaux.

2.5.1. L'échange total

L'algorithme d'échange total consiste en une succession de diffusions élémentaires qui peuvent utiliser l'ensemble des directions à chaque étape. Grâce à la propriété 4, c'est-à-dire du fait que notre graphe soit à sommets transitifs [8], il nous suffit de choisir un seul processeur, et de suivre son évolution pour connaître celle de l'ensemble de l'algorithme.
Notons c_i^t le chemin qu'a parcouru le message reçu par ce processeur sur son i-ème lien au cours de la t-ième diffusion élémentaire.
L'algorithme de l'échange total est alors de la forme suivante :

Algorithme II.5 : Echange total

Pour t variant de 1 jusqu'à t_{max}

  Pour toute direction i
    lien(i) $\leftarrow$ message($c_i^t \Delta \{i\}$)

  cube-swap
  Pour toute direction i
    message(c_i^t) $\leftarrow$ lien(i)

Examinons les contraintes que doivent respecter les paramètres c_i^t pour que cet algorithme corresponde effectivement à un échange total :

i) La première à respecter est une contrainte de précédence comme celle décrite dans la remarque 5. Cette contrainte traduit qu'un processeur ne peut émettre qu'un message qu'il a déjà reçu.
Plus formellement :

$$\forall\, t \text{ et } i \begin{cases} c_i^t = i \quad \text{ou} \\ \exists\, \tau < t \text{ et } j \text{ tels que } c_i^t \Delta \{i\} = c_j^\tau. \end{cases}$$

ii) La deuxième est une contrainte de terminaison : le processeur témoin doit avoir reçu tous les messages. Autrement dit :

$$\forall c,\ \exists t \text{ et } i \text{ tels que } c = c_i^t.$$

Les deux premières contraintes suffisent pour obtenir un échange total. Mais il est possible d'ajouter troisième contrainte pour garantir l'optimalité de cet échange total.

iii) La contrainte d'optimalité : à chaque diffusion élémentaire le processeur témoin doit recevoir un maximum de nouveaux messages.

Détaillons un échange total sur un 3-cube. Sur la figure II.10, les différents pas de l'algorithme sont visualisés par rapport à la réception par un processeur donné (le coin inférieur gauche).

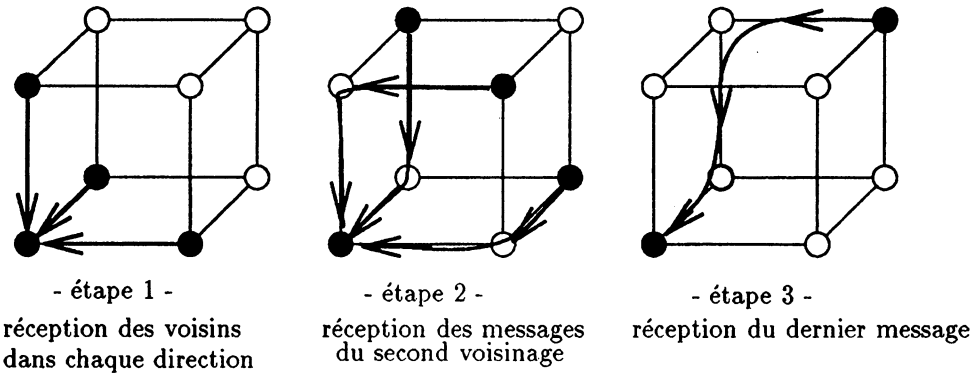


Figure II.10. Echange total sur un 3−cube

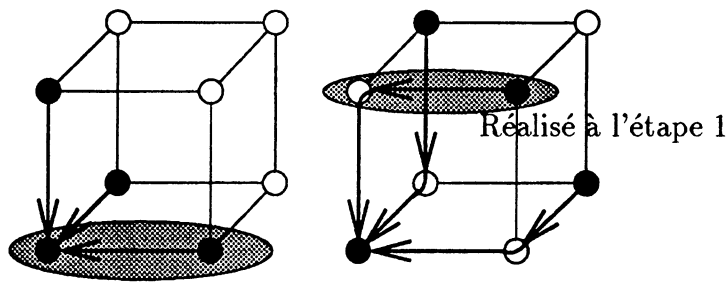
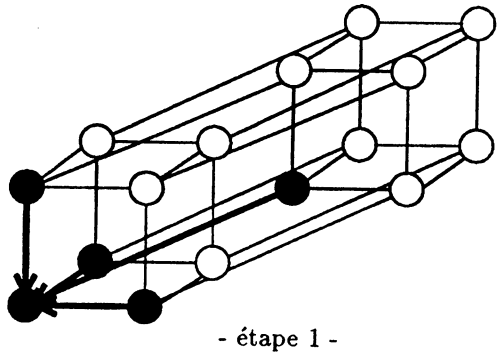


Figure II.11. détail de deux cube-swap successifs (vue du processeur 0).

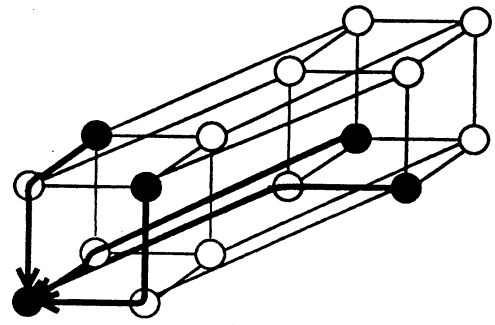
Remarque 8 : Le fait que le nombre d'étapes soit égal à la dimension du cube est un cas particulier pour le 3-cube.

Sur cet exemple du 3-cube, à chaque étape, on remarque que les messages émis ont effectivement été reçus antérieurement (comme le montre la figure II.11 l'étape 2 est valide parce que tous les messages émis ont été reçus à l'étape 1).

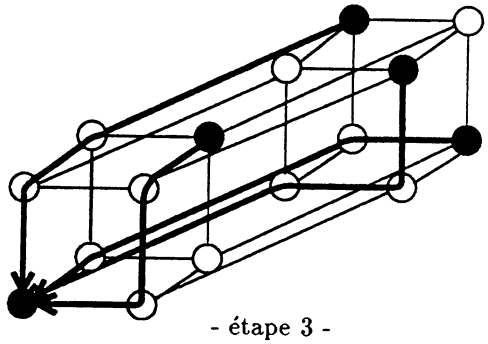
Nous donnons un autre exemple sur le 4-cube pour avoir un exemple plus général.



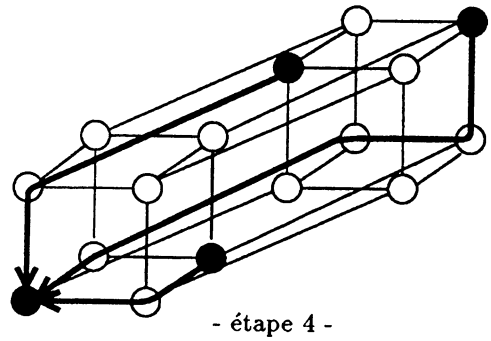
- étape 1 -
réception des messages
du premier voisinage



- étape 2 -
réception des messages
du deuxième voisinage



- étape 3 -
réception des messages
du troisième voisinage



- étape 4 -
réception des messages
des nœuds cycliques

Figure II.12

2.5.2. Construction des C_i^t.

Il existe une méthode simple pour construire les c_i^t, basée sur la contrainte de précédence. En effet, il faut avoir reçu un message pour pouvoir le renvoyer. Donc pour ne pas avoir de problème, il semble judicieux de recevoir tous les messages appartenant à tous les processeurs à distance q du processeur témoin, avant de s'intéresser à ceux à distance $q+1$. Cette construction permet de ramener à la fin du rapatriement des messages à distance q, l'ensemble des messages à distance

$q+1$ dans les processeurs voisins (i.e. à distance 1).

L'algorithme de construction s'écrit :

Algorithme II.6 : Construction des c_i^t

Pour $i = 1$ jusqu'au *Diamètre* faire
    Recevoir tous les messages à distance i

Cette version de l'algorithme est suffisante pour permettre d'effectuer un échange total. Mais l'optimalité définie en iii) n'est pas toujours vérifiée, car le nombre de messages reçus par un processeur n'est pas maximum. Pour que le « débit » soit maximum à chaque étape, il faut que :

$$\forall k, |Q_k| \text{ soit divisible par le degré}$$

où $|Q_k|$ désigne le cardinal de l'ensemble des messages à distance k

Cette condition n'est vérifiée seulement pour un hypercube dont la dimension d est nombre premier. Pour toutes les autres dimensions, il existe au moins un niveau où le cardinal n'est pas divisible par le degré, c'est-à-dire $\frac{C_p^d}{d}$ n'est pas un entier.

Remarque 9 : Ce même type de construction niveau par niveau est aussi valable sur les grilles toriques carrées ($n \times n$), car le cardinal de chaque niveau est divisible par 4. En effet par définition, $|Q_i| = 4 + |Q_{i-1}|$ et comme $|Q_1| = 4$, le cardinal est toujours divisible par le degré (4).

Il faut donc trouver une méthode plus générale qui marche pour toutes les dimensions de l'hypercube. Nous nous sommes aperçus que pour les cas où la dimension est un nombre premier, la construction des c_i^t est équivalente à la génération des arbres équilibrés de Ho et Johnsson [66]. Nous avons étudié de plus prés leur génération dans le cas général, et avons observés deux *défauts* pour le cas particulier des c_i^t :

- les arbres obtenus à partir de leur algorithme ont une profondeur, c'est-à-dire un nombre d'étapes, supérieure à celui désirée $\frac{2^d-1}{d}$. En fait, leurs arbres ne sont pas vraiment équilibrés, mais seulement "à peu près équilibrés",
- les contraintes nécessaires pour obtenir un arbre équilibré sont plus fortes que celles pour obtenir les c_i^t. Pour cela nous avons supprimé cette contrainte de paternité, trop restrictive, pour la remplacer par la contrainte de précédence i), plus faible.

Nous allons voir maintenant comment, à partir des définitions de Ho et Johnsson, obtenir les c_i^t de façon à effectuer un échange total optimal en nombre de communication pour toutes les dimensions d'un hypercube. Il est nécessaire avant tout de rappeler certaines définitions utiles à la présentation de l'algorithme.

Définition 8 : A tout nœud d'un hypercube, on peut associer un représentation binaire construite à partir de la définition récursive d'un hypercube. Elle s'appelle le *code de Gray* [95].

Définition 9 : On appelle *arbre équilibré* un arbre construit à partir de n'importe quel processeur dont les d premières branches sont de même taille, à un élément près.

Pour plus de simplicité, nous étudierons par la suite uniquement le cas où la racine est le processeur 0. En effet, quelle que soit la racine choisie, il suffit de faire un XOR (« OU exclusif » logique) sur les représentations binaires associées aux sommets avec la racine pour connaître les liaisons entre processeurs et ainsi construire un nouvel arbre équilibré.

Définition 10 : On appelle *Période* d'un nœud le nombre minimum de décalages circulaires par la droite nécessaires pour obtenir une nouvelle fois la même représentation binaire.

Définition 11 : On appelle *Nœud Cyclique*, un nœud dont la période est strictement inférieure au nombre de bits utilisés pour son codage binaire (i.e. d).

Définition 12 : On appelle *Base* d'un nœud le plus petit nombre de décalages circulaires par la droite nécessaires pour obtenir la représentation binaire donnant le plus petit entier possible, C'est-à-dire la représentation ayant le plus grand nombre de zéro possible à gauche.

Par exemple, le nœud 11010 appartient à la base 3 car 01011 est la représentation la plus petite possible obtenue après 3 décalages circulaires vers le droite.

Remarque 10 : Tous les éléments d'une base ont tous le même bit à 1, c'est celui du numéro de la base. On appellera ce bit le bit caractéristique de la base.

Ho et Johnsson proposent une règle qui définit un lien de parenté interne aux bases. Cette règle est la suivante :

Règle 1 : Si un nœud n'a qu'un seul 1 dans sa représentation alors son père est la racine, sinon il suffit de complémenter le premier bit à 1 se trouvant à droite du bit caractéristique de la base pour obtenir sa représentation binaire.

Avec ces définitions, et cette règle, la construction des arbres est alors simple. Il suffit :

- dans une première étape, de regrouper tous les nœuds de l'hypercube par base. Grâce à ces bases, on peut créer d branches distinctes.
- Puis dans une deuxième de créer les liaisons de parenté entre les nœuds internes à la base..

La caractéristique première de cette règle est de conserver la définition de base. Une fois, les nœuds regroupés entre eux dans une base, leur père est obligatoirement dans la même base. Une démonstration se trouve dans [66]

Finalement, l'algorithme de construction s'écrit :

Algorithme II.7 : Arbre de Ho et Johnsson

Pour l'ensemble des nœuds **Faire**
Chercher la base du nœud
Placer le nœud dans la branche associée à la base.
Créer le lien avec son père.

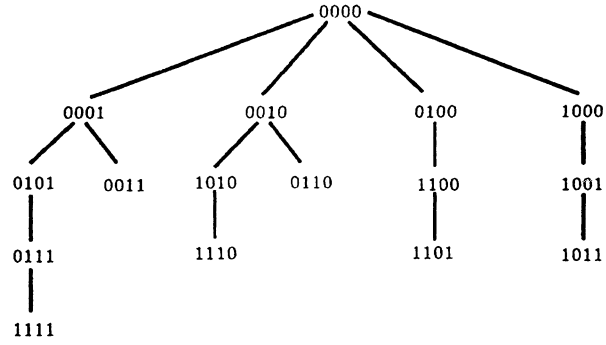


Figure II.13. Exemple d'arbre de Ho et Johnsson non équilibré.

Le problème majeur de cet algorithme de construction provient de la définition de base telle qu'elle est décrite. Elle ne permet pas d'équilibrer les branches car les nœuds cycliques peuvent appartenir à plusieurs bases. Alors le choix arbitraire de Ho et Johnsson d'imposer le nœud dans la base la plus petite, déséquilibre l'arbre. Alors on obtient des arbres avec plus de nœuds sur les premières branches (cf figure II.13). Ainsi cet algorithme n'est optimal que pour les hypercubes dont la taille est un nombre premier car dans ce cas il n'existe pas de nœud cyclique.

Pour trouver une solution à ce problème, nous avons créé un nouvel algorithme. Celui-ci s'appuie fortement sur le schéma précédent. La première phase reste identique, en se limitant aux nœuds non-cycliques. Dans une deuxième phase, on va placer les nœuds cycliques à la fin en remplissant les liens au maximum pour vérifier la contrainte d'optimalité. L'algorithme s'écrit :

Algorithme II.8 : Version améliorée

Pour l'ensemble des nœuds *non cycliques* **Faire**
Chercher la base du nœud
Placer le nœud dans la branche associée à la base.
Créer le lien avec son père.

Pour l'ensemble des nœuds *cycliques* **Faire**
Placer les nœuds un à un successivement dans toutes les branches.

Vérifions que cet deuxième phase permet de garantir que l'on suit bien les deux autres contraintes : celle de terminaison et celle de précédence. Pour la première, il n'y a pas de problème, par construction l'ensemble de messages est traité. Pour la deuxième, il faut s'assurer qu'il existe un message c_j^τ pour tous les nœuds cycliques.

Théorème 2 : Si on complémente n'importe quel bit d'un nœud cyclique p, alors $\forall i$, $p\Delta\{i\}$ est un nœud non cyclique.

M:ontrons qu'il est impossible d'obtenir un nœud cyclique à partir d'un autre nœud cyclique.
Pour cela utilisons la figure II.14. En complémentant un bit d'un nœud cyclique, on annule la répétition du cycle, et il faut donc en trouver une autre. Or il est impossible d'en créer une autre. En effet, si cela était possible cette nouvelle répétition serait obligatoirement de taille supérieure car sur la partie non modifiée le plus petit cycle est donné par l'ancienne répétition. Et comme elle est supérieure, elle serait sûrement basée sur l'ancienne qui n'existe plus dans la partie où on a modifié le bit. En conclusion il est impossible d'obtenir un nœud cyclique en modifiant un seul bit d'un nœud cyclique. □

Nous venons de démontrer que cette méthode permet de construire la table des c_i^t pour effectuer un échange total. Montrons sur l'exemple du 4-cube comment obtenir et utiliser utiliser cette table.

Dans la pratique, il suffit de générer en premier tous les nombres de base 0 non cycliques dans un ordre croissant pour avoir la propriété i). Ensuite il faut les

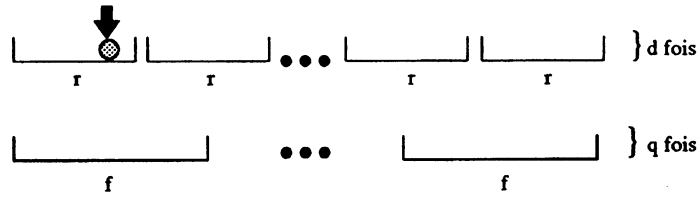


Figure II.14. Modification sur un nœud cyclique par le complément d'un bit.

placer dans la première colonne du tableau (cf tableau II.1). Les autres colonnes s'obtiennent alors facilement par un simple décalage de la représentation binaire du nœud d'un bit vers la gauche pour avoir les éléments de la base 1, de deux pour la base 2,... et ainsi de suite pour obtenir le tableau complet des nœuds non-cycliques.

Il reste donc à la fin de cette étape à arranger tous les nombres restants: nombres cycliques. (voir tableau II.2). Pour cela, il reste seulement à les prendre un à un et de les placer au fur et à mesure en ligne à la suite du tableau II.1.

Il existe deux versions différentes pour l'ordre dans lequel on il est possible de choisir les nœuds cycliques:

- Soit en prenant les nœuds dans l'ordre croissant.
- Soit en plaçant en premier les nœuds cycliques à deux bases, puis à trois, et ainsi de suite jusqu'à $\frac{p}{2}$.

Sur les petites tailles, ces deux méthodes donnent les mêmes tables. Détaillons les deux étapes à suivre dans le cadre de la première méthode sur un 4-cube à l'aide de tableaux et de figures. On obtient à la fin de la première étape le tableau suivant:

Base0	Base1	Base2	Base3
0001	0010	0100	1000
0011	0110	1100	1001
0111	1110	1101	1011

Tableau II.1. Placement des nombres non-cycliques.

Les nombres de base 0 non cycliques sont 0001, 0011, 0111. Et à la fin de l'algorithme, la table est la suivante:

Remarque 11 : Il n'existe pas qu'un seul tableau de c_i^t, l'algorithme ci-dessus n'est qu'une manière de l'obtenir de façon systématique.

lien0	lien1	lien2	lien3
0001	0010	0100	1000
0011	0110	1100	1001
0111	1110	1101	1011
0101	1010	1111	

Tableau II.2. Une des solutions finales possible.

Comment lire cette table pour connaître le chemin suivi par le message. Il suffit de complémenter le bit à 1 qui correspond à la colonne où se trouve le message. Par exemple, 1010 se trouve dans la deuxième colonne, on complémente le deuxième bit et on obtient 1000. On a donc reçu le message 1010 sur le lien 2 à partir du message 1000. Pour connaître le chemin complet on recommence avec le message 1000 qui a été reçu sur le lien 4 à partir de l'origine. En conclusion le message 1010 a traversé successivement le lien 4 puis le lien 2 pour arriver à destination.

2.5.3. Amélioration des arbres de Ho et Johnsson

Ayant trouvé une solution pour les c_i^t, nous sommes intéressés de plus près au problème du non-équilibrage des arbres de Ho et Johnsson. En effet, les arbres équilibrés sont comme nous venons de le voir utile pour la réalisation d'un échange total mais servent aussi dans d'autres schémas de communication comme la diffusion. Nous nous sommes alors posés la question suivante: Existe-t-il une version qui permettent de définir un arbre équilibré, ou plutôt existe-t-il une répartition des nœuds cycliques qui équilibre la table? Nous avons seulement réussi à répondre à cette question de façon expérimentale avec l'algorithme suivant:

Algorithme II.9 : Nouvel arbre amélioré

Pour l'ensemble des nœuds *non cycliques* **Faire**
Chercher la base du nœud
Placer le nœud dans la branche associée à la base.
Créer le lien avec son père.

Pour l'ensemble des nœuds *cycliques* **Faire**
Chercher les bases possibles.
Equilibrer les branches au fur et à mesure.
Créer le lien avec son père.

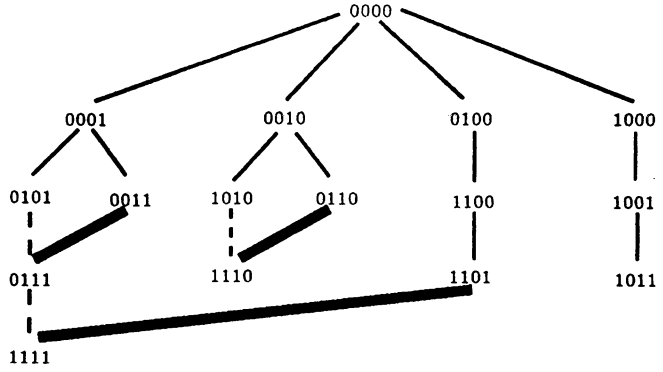


Figure II.15. Arbres équilibrés obtenus avec notre algorithme

Ces deux versions en fonction de l'ordre du choix des nœuds, ont été implantées sur Sparc Station. Elles donnent des résultats optimaux jusqu'à la taille 22, taille limite de stockage possible sur notre station de travail [36]. Une preuve de l'existence d'une solution optimale pour la répartition des nœuds cycliques n'a pu être développée en utilisant le formalisme de [66] .

2.5.4. Les arbres de Bertsekas et Tsitsiklis

Au même moment, Bertsekas et Tsitsiklis ont développé une nouvelle méthode de construction qui ne repose plus sur les définitions précédentes [9]. La construction de ces arbres se fait sur une nouvelle règle, basée sur le placement des sommets par nombre croissant de 1 dans la représentation binaire des nœuds, c'est-à-dire niveau par niveau.
Avant de la présenter, il est nécessaire de rappeler quelques définitions et d'expliciter clairement leur nouvelle règle.

Définition 13 : On appelle N_k la classe contenant tous les nœuds ayant k bits à 1. On remarque que chaque classe comprend C_k^n éléments (où C_k^n représente le coefficient binomial). En particulier, N_0 et N_d contiennent un seul élément, le nœud $(0\cdots00)$ et $(1\cdots11)$ respectivement.

Définition 14 : On appelle R_{ki} un sous ensemble de N_k dont tous les éléments sont équivalents à des rotations près.
R_{k1} est choisi de tel sorte qu'il contienne le nœud composé de k bits consécutifs à 1 le plus à droite possible. Par exemple, 00011 est un élément de R_{21}.

Par exemple, 100110 et 011010 appartiennent au même sous ensemble R_{3i} car le deuxième est obtenu après deux décalages du premier vers la gauche.

Remarque 12 : Le nombre maximum d'éléments dans un sous ensemble R_{ki} est de d.

Définition 15 : On appelle n_k le nombre maximum de sous ensembles possibles pour la classe N_k.

A partir des différentes définitions ci-dessus, on crée la suite :

$$N_0\ N_1\ \cdots\ N_{d-1}\ N_d$$

Grâce à la définition 14, on obtient :

$$(00\cdots00)\,R_{11}\,R_{21}\,\cdots\,R_{2n_2}\,\cdots\,R_{k1}\,\cdots\,R_{kn_k}\,\cdots\,R_{(d-1)1}\,(11\cdots11) \qquad (1)$$

Cette liste permet de placer dans l'ordre croissant de 1 dans la représentation binaire l'ensemble des nœuds. On définit alors la fonction *place* qui, à chaque élément de la liste, associe sa position dans celle-ci.

$$\begin{aligned} place &: \mathrm{IR} \longrightarrow \mathrm{IR} \\ & t \longrightarrow place(t) \end{aligned}$$

Soit la relation

$$m(t) = 1 + [(place(t) - 1) mod\, d] \qquad (2)$$

On obtient alors la suite :

$$1, 2, \cdots, n, 1, 2, \cdots, n, 1, \cdots \qquad (3)$$

En superposant les suites 1 où l'on a supprimé la racine $(00\cdots00)$ et 3, on crée facilement une "structure" par niveau où le premier niveau est composé des d premiers éléments, le deuxième des d suivants, et ainsi de suite. Le problème est maintenant de savoir comment ordonner les nœuds à l'intérieur même des sous ensembles R_{ki}, ainsi que comment liés les niveaux entre eux pour obtenir un arbre équilibré avec les bons liens de parenté.
C'est pourquoi, Bertsekas et Tsitsiklis ont défini une nouvelle règle qui spécifie cet ordre et ces liens.

Règle 2 : Le premier élément de R_{ki} doit avoir le bit en position m(t) à 1. les suivants sont obtenus par simple décalage d'un bit vers la gauche.

En imposant le bit en position m(t)−1 soit à 0 pour les sous ensembles R_{k1}, ils définissent alors comme pour les arbres la méthode précédente, d groupes distincts de nœuds ayant le même bit à 1.
Comme lien de parenté entre les nœuds, ils utilisent le lien défini par le bit imposé à 1 en position $m(t)$ pour communiquer. Pour trouver le père d'un élément, il suffit de complémenter ce bit.
Par exemple, dans un 5-cube le nœud 01101 appartient à la classe R_{32} où il occupe la troisième position. Son père est donc le nœud 01001. Il a suffit de complémenter le troisième bit en partant de la droite.

Mais ces communications définissent-elles un arbre équilibré ?

Algorithme II.10 : Arbres de Bertsekas et Tsitsiklis

 # Création de la liste #
Pour $k = 1 \cdots p$
    Répartir les nœuds dans les classes R_{ki} **selon les règles précédentes.**

Pour $etage = 1 \cdots \left\lceil \frac{2^p-1}{p} \right\rceil$
    Pour $k = 1 \cdots p$
        Créer le lien de parenté avec un étage précédent.

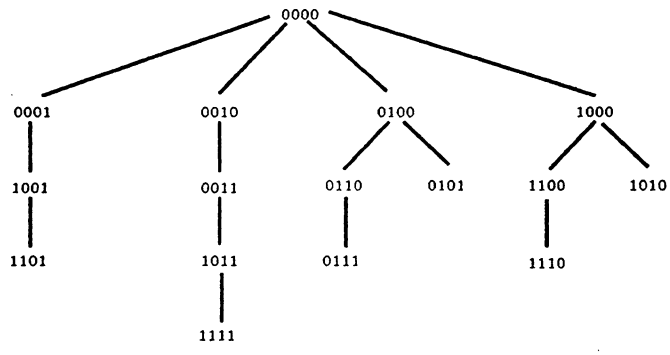


Figure II.16. Arbres équilibrés de Bersekas et Tsitsiklis

Le point le plus important sur la méthode de Bertsekas et Tsitsiklis est qu'ils imposent un bit à 1 en position précise. C'est cette propriété qui permet de trouver un père à chacun des nœuds. Ils prouvent ainsi facilement que ce père se trouve au moins un niveau au dessus dans l'arbre [9] et qu'ils construisent donc bien un arbre équilibré.

L'existence d'un père dans un niveau supérieur tient à deux arguments :

- La propriété du bit à 1,
- et l'ordre de classement des nœud qui place tous les nœuds ayant $k-1$ 1 avant de placer ceux qui en ont k.

De plus, le fait d'imposer la place pour chaque élément de R_{k1} permet "d'équilibrer" l'arbre. Ce sont souvent les éléments de cette classe qui sont les pères des

éléments des classes supérieures $R_{k+1,j}$, surtout ceux de $R_{k+1,1}$.

En conclusion, l'algorithme de Bersekas et Tsitsiklis est plus général que le notre, mais la construction des arbres est un peu plus complexe. En revanche pour la création d'un échange total, il est évident que notre méthode [36] est plus facile à mettre en œuvre.

2.6. L'échange total personnalisé avec accumulation.

L'algorithme d'échange total personnalisé avec accumulation est similaire à celui de l'échange total: les diffusions élémentaires sont remplacées par des accumulations élémentaires (voir 2.4 et les messages parcourent les mêmes chemins mais en sens contraire.

Notons c_i^t le chemin que doit parcourir le message envoyé par un processeur sur son i-ème lien au cours de la t-ième accumulation élémentaire. L'algorithme de l'échange total personnalisé avec accumulation prend alors la forme suivante:

Algorithme II.11 : échange total personnalisé avec accumulation

Pour toute direction i
    lien(i) $\leftarrow$ message(c_i^t)
    message(c_i^t) $\leftarrow$ 0

cube-swap
Pour toute direction i
    message($c_i^t \Delta \{i\}$) $\leftarrow$ message($c_i^t \Delta \{i\}$) + lien(i)

Les paramètres c_i^t sont les mêmes que ceux de l'algorithme de l'échange total mais pris dans l'ordre inverse chronologique (les messages reçus en dernier lors de l'échange total sont les premiers a être émis lors de l'échange total personnalisé avec accumulation.).
Avec un tel choix des c_i^t, l'instruction de mise à zéro n'est plus nécessaire. En effet, la contrainte i) des c_i^t de l'échange total prise en remontant dans le temps garantit qu'un processeur ne recevra plus de message pour un autre processeur à partir du moment où il lui a déjà envoyé un message.

2.7. L'échange partiel

Définition 16 : Un voisinage d'un processeur p est un ensemble connexe de processeurs contenant p.

Informellement, lors d'un échange partiel sur un sous-ensemble de processeurs (que nous appellerons abusivement voisinage), chaque processeur a initialement un message qu'il destine à tous ceux du voisinage et doit à la fin avoir reçu un message de chacun d'entre eux (les voisinages de deux processeurs quelconques étant identiques "à une translation près"). Si ce voisinage est égal à l'ensemble de tous les processeurs on retrouve alors un échange total.
Pour pouvoir avoir une définition plus formelle, précisons d'abord la notion d'identité "à une translation près".

Un voisinage relatif est un ensemble de chemins V tel que pour tout processeur p l'ensemble des processeurs atteints depuis p en suivant les chemins de V soit un voisinage de p.
On note $p + V$ cet ensemble. Si q appartient à $p + V$ alors p appartient à $q + V$.

Etant donné un voisinage relatif V, lors d'un échange partiel sur V, chaque processeur a initialement un message qu'il destine à tous les processeurs de son voisinage $p + V$ et finalement doit avoir reçu un message de chaque processeur de ce même voisinage.

Pour écrire un algorithme d'échange partiel sur un voisinage V, il suffit de décrire ce voisinage par un ensemble de chemins et d'ordonner ces chemins en respectant la contrainte de précédence i) comme nous l'avons vu lors de l'étude de l'échange total (Cf 2.5.1).

3. Conclusion

Nous avons présenté dans ce chapitre une analyse théorique des procédures de communications optimales pour les schémas de communication régulier que sont l'échange total et l'échange total personnalisé avec accumulation. Le même type de démarche est valable sur un réseau en grille torique (produit cartésien de deux anneaux) [53]. Il est alors possible de programmer de façon optimale en nombre d'étape de communication, le produit matrice-vecteur avec les même procédures de communications et le même placement des données. De manière plus générale, il serait intéressant d'étudier les différents placements en fonction de la topologie du réseau d'interconnexion : De bruijn, Cube-connected-cycle, etc..

Nous avons proposé de plus un nouvel algorithme pour la création d'arbres équilibrés. L'implantation de cette version donne l'optimalité expérimentalement jusqu'à la dimension 22 d'un hypercube. Vu les dimensions employées dans les différentes architectures parallèles, cette solution est actuellement la plus simple à utiliser. Une preuve de cet algorithme n'a pour l'instant pas été trouvée.

Chapitre III

Expérimentation du produit Matrice-Vecteur sur la CM2

Ce chapitre est consacré à l'expérimentation sur la Connection Machine 2 du produit matrice-vecteur itéré. Nous détaillons les résultats en utilisant différents niveaux de programmation. Mais pour utiliser le modèle employé lors de la description de la méthode, modèle qui peut surprendre les utilisateurs de la Connection Machine qui ont la vision classique des processeurs 1-bit, il faut oublier les processeurs 1-bits et seulement retenir la vision par rapport aux processeurs flottants [46, 13]. On peut alors seulement à condition de descendre au niveau CMIS [13, 35, 46] programmer la méthode décrite dans le chapitre précédent.

1. Implémentation sur la Connection Machine

1.1. Implantation en langage de haut-niveau.

Présentons tout d'abord une implantation en langage de haut-niveau ***lisp**. Le placement est à la charge du programmeur, et suit la règle d'or « une donnée par processeur ». On raisonne donc ici sur une grille de processeurs (correspondant à la structure logique de la matrice) dont les quelques opérations parallèles disponibles utilisent la topologie en hypercube, transparente à l'utilisateur [35].

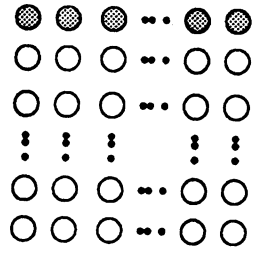


Figure III.17. Placement initial : l'élément a_{ij} est dans le processeur $P(i,j)$ et le vecteur $x(i)$ dans $P(1,i)$ (en gris).

On rappelle que l'algorithme à implanter est celui du produit matrice-vecteur itératif, et que donc à la fin de chaque itération, on doit être en mesure de recommencer (i.e. être dans le même état qu'au début, en gris sur la figure III.17).

Une itération s'écrit sous la forme :

Algorithme III.1 : Une itération du produit matrice vecteur.

Duplication du vecteur x selon les colonnes.
Calcul sur chaque processeur $P(i,j)$ du produit $a_{ij} \times x(j)$.
Scan avec addition pour effectuer le produit scalaire sur les lignes.
Transposition du résultat qui se trouve dans la première colonne sur la première ligne.

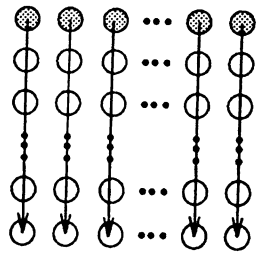


Figure III.18. Duplication du vecteur x dans tous les processeurs (suivant les colonnes).

Les trois figures III.18, III.19 et III.20 qui décrivent les trois étapes de l'algorithme, montrent clairement le nombre important de communications à effectuer. La plus coûteuse d'entre elles est le problème de transposition du résultat pour pouvoir itérer le produit. Elle fait appel au routeur général, appel le plus cher

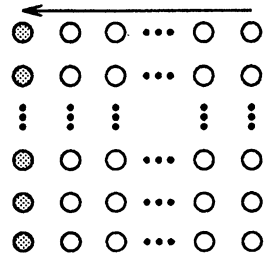


Figure III.19. Réduction (produit-scalaire) selon les lignes. (résultat en gris)

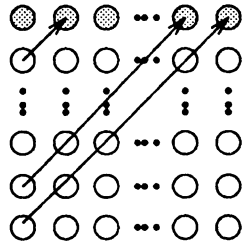


Figure III.20. Transposition du vecteur résultat.

des trois.
L'utilisation du Communication Compiler, vu le caractère répétitif de la transposition, est une solution envisageable pour améliorer les performances.

Dans le tableau ci-dessous, nous présentons les temps obtenus pour réaliser 100 produits matrice-vecteur sur une Connection Machine avec 8k-processeurs, avec et sans la transposition du vecteur résultat.

Taille de la matrice	Temps avec transposition	Temps sans transposition
128 × 128	0.389	0.16
256 × 256	2.07	0.29
512 × 512	8.019	0.73
1024 × 1024	32.09	2.16

Tableau III.3. Temps en seconde des programmes *lisp pour 100 produits enchaînés.

1.2. Le niveau CMIS

La seule façon de programmer le produit matrice-vecteur itéré tels qu'il présenté dans le chapitre II sur la Connection Machine 2 est de descendre au niveau le plus bas : le niveau CMIS. CMIS (CM Instruction Set) est l'assembleur de la Connection Machine (pour plus de détails voir Annexe A). Un point intéressant de CMIS est d'avoir directement accès aux liens de l'hypercube par des échanges (cube-swap) de mots de 32 bits entre les modules. Les registres de chaque module servent de buffer pour les messages à envoyer. Le principe de communication correspond au modèle présenté en 2.2. Il est ainsi possible de programmer efficacement les deux routines de communications.

1.3. Implantation des routines de communication

L'implantation des routines a été réalisée à partir des tables des c_i^t obtenues avec notre algorithme. Leur programmation a été faite en les décomposant en tops de communications. C'est-à-dire qu'on a programmé chaque top de communication en prenant une ligne de la table. L'inconvénient de cette méthode apparaît lorsque la dimension de l'hypercube augmente. Par exemple pour un hypercube de dimension 10 demande une décomposition en 103 étapes. Une programmation automatique de la table serait idéale à partir de la dimension 9.

Comparons tout d'abord les deux procédures de communications nécessaires : l'échange total et l'échange total personnalisé avec accumulation.

Taille de l'hypercube	Echange total	Echange total Acc
4	955	1710
5	1790	3345
6	2971	6169
7	5255	11728
8	9060	22082

Tableau III.4. Temps des procédures de communication en Ticks.

Remarque 13 : Les temps sont donnés en Ticks, mesure de temps propre à la Connection Machine. La relation entre ticks et tops horloge est la suivante : le nombre total de ticks représente un quart du nombre total de cycles internes utilisés pour effectuer la procédure. La transformation en temps dépend alors de la fréquence de l'horloge de la machine. T.M.C annonce une fréquence de 8Mhz pour ses derniers modèles, un tick équivaut donc à 0.5μ secondes.

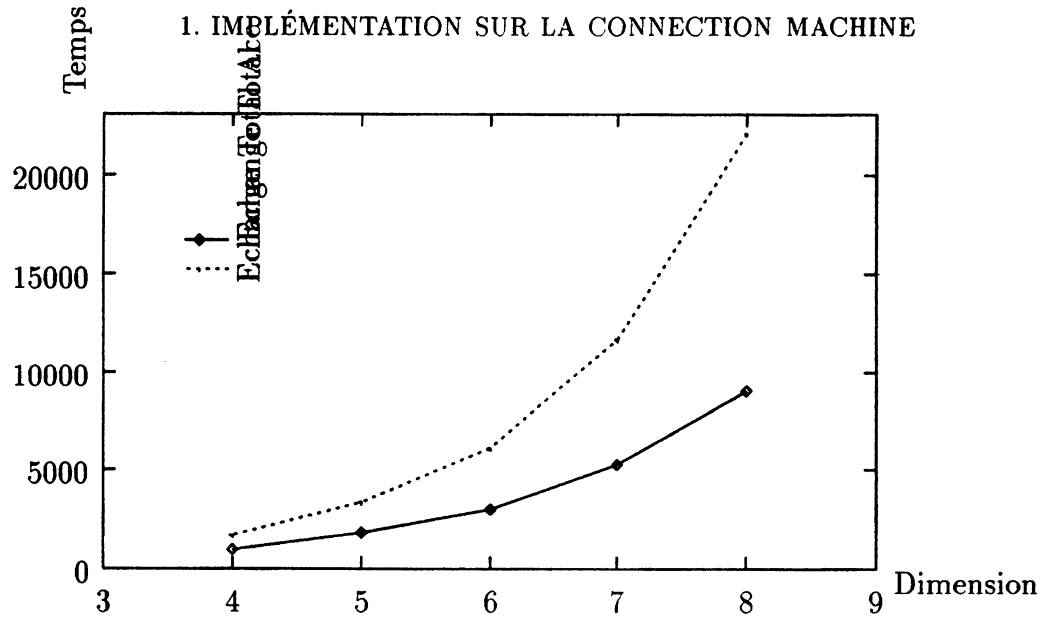


Figure III.21. différence entre les deux types de communication

Le rapport entre eux est d'environ deux, et augmente avec la dimension de l'hypercube. L'influence des calculs dans l'échange total personnalisé avec accumulation est donc assez important.

1.4. Le produit matrice-vecteur

L'utilisation du placement de données décrit dans le chapitre précédent (section 1) et des procédures de communications décrites dans la section 2 conduit à une solution optimale sur une Connection Machine 8k-processeurs. Dans ce cas, la dimension de l'hypercube formé par les processeurs flottants, est un hypercube de dimension 8 peut se décomposer en le produit cartésien de deux hypercubes de dimension 4 :

$$H_8 = H_4 \otimes H_4.$$

Chaque étape de communication de l'algorithme ci-dessus est alors réalisée en 4 tops, donc au total 8 tops, ce qui permet d'atteindre la borne optimale du diamètre.

Le schéma est exactement celui présenté en II.1.4. Les deux sous graphes choisis sont deux hypercubes de dimension 4 (H_4), qui permettent une programmation optimale en nombre de tops de communication sur la Connection Machine à 8k-processeurs. Les procédures de communication utilisées ont celles décrites ci-

dessus, le produit matrice-vecteur sur chaque nœud (processeur flottant) est celui décrit dans le chapitre sur CMIS.

Les résultats obtenus sont :

Taille du bloc	Echange Total	Echange Total Acc	Calculs
16	955	1710	310
32	1236	2665	1620
64	2460	5325	7150
128	5000	10650	28820
256	9860	21300	116500

Tableau III.5. Répartition du temps global entre les trois routines en Ticks

La complexité du produit matrice-vecteur est de $2n^2 - n$ opérations arithmétiques. On obtient alors les performances suivantes :

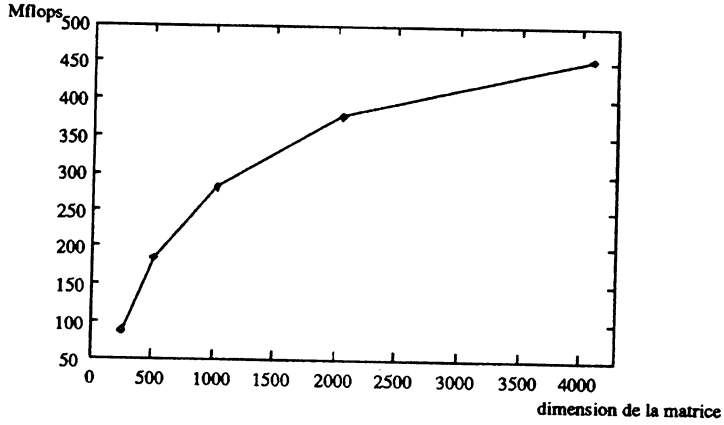


Figure III.22. Performances sur une Connection Machine avec 8K-processeurs

On s'aperçoit que le gain en performances diminue avec la dimension de la matrice. Ceci s'explique très bien dans le tableau III.5 avec la part de plus en plus importante du calcul par rapport aux deux étapes de communication. On va donc tendre rapidement vers une performance crête due à la puissance de calcul.

A partir des résultats obtenus sur la 8K-processeurs, il est facile de les étendre aux Connection Machine de taille supérieure en mesurant les différents temps pour effectuer les deux schémas de communication sur des hypercubes de dimension supérieure. En effet, le temps de calcul sur un processeur flottant reste le même

quelque soit la taille de la machine, et donc une estimation des performances est faite par la formule suivante :

$$\text{Perf} = \text{Tps de l'ET} + \text{Tps de l'ET-Acc} + \text{Tps Calcul}$$

où Tps de l'ET et Tps de l'ET-Acc dépendent de la dimension des deux sous graphes ainsi que du nombre de coordonnées du vecteur sur un processeur, et Tps Calcul uniquement du nombre de coordonnées du vecteur.

Par exemple, pour une Connection Machine à 32K-processeurs, en utilisant la décomposition d'un hypercube de dimension 10 en un produit cartésien de deux hypercubes de dimension 5. ($H_5 \otimes H_5$), une estimation de performance donne une puissance de l'ordre de 1.6 Gigaflops (i.e. 1.6 Milliards d'opérations à la seconde).

Remarque 14 : Le nombre de tops de communication nécessaire pour effectuer l'ensemble des communications est de 14, alors que le plus court chemin est de 10. Peut-être existe-t-il un meilleur choix de voisinage pour diminuer leur nombre. Nous n'avons trouvé que des solutions équivalentes avec des voisinages différents.

2. Conclusion

Dans le cas de la Connection Machine, il est nécessaire d'insister sur le fait que ce n'est pas l'optimalité des procédures de communication qui entraîne l'optimalité du produit matrice-vecteur mais surtout la décomposition en produit cartésien d'un hypercube de taille 8. Pour les tailles supérieures, rien ne nous permet d'affirmer qu'on ne peut jamais atteindre le plus court chemin avec un autre découpage.

Une comparaison avec la bibliothèque CMSSL[1] du constructeur T.M.C.[2] a été réalisée avec l'aide monsieur Roch Bourbonnais l'ingénieur de T.M.C. sur le site de l'E.T.C.A. Les performances obtenues sont d'environ un rapport deux fois plus lentes que celles obtenues avec la bibliothèque CMSSL. Mais ce rapport reste honnête lorsque l'on sait que les procédures de CMMSL sont écrites dans un langage proche de CMIS mais qui ont été développées en étroite collaboration avec le concepteur matériel.

[1] Connection Machine Scientific Software Library
[2] Thinking Machines Corporation

Chapitre IV

Tout n'est pas toujours aussi facile.

Nous avons vu dans le chapitre précédent qu'il est possible de programmer efficacement en mode S.I.M.D., et d'implanter les algorithmes avec un investissement minimum sur la Connection Machine. Nous allons maintenant étudier certains algorithmes dont les performances chutent rapidement sur des machines massivement parallèles. On s'intéresse plus particulièrement au cas de la résolution triangulaire, en étudiant sa complexité théorique sur ce type de machine. Nous proposons une autre méthode basée sur les réseaux systoliques, dont la comparaison avec la méthode classique apporte des conclusions contradictoires.

1. Résolution de systèmes triangulaires.

Les méthodes de résolution de systèmes linéaires triangulaires (supérieurs et inférieurs) sont souvent utilisées comme dernière étape pour résoudre $ax = b$. Par exemple, la factorisation de Cholesky ou Gauss [55, 85], ou encore des systèmes issus du préconditionnement du Gradient conjugué [93] demandent pour obtenir le vecteur x une résolution triangulaire.

La parallélisation de ces méthodes est un sujet souvent traité. Ces travaux sont presque tous exclusivement effectués en mode M.I.M.D.[79, 61, 91, 85, 32]. Heath et Romine [61] proposent trois types d'algorithmes différents pour la résolution triangulaire : premièrement, les algorithmes de type « fan-in » et « fan-out » (c'est-à-dire distribution et regroupement), deuxièmement ceux de type « front de propagation », et troisièmement les algorithmes cycliques. Leurs implantations ont été réalisées sur un hypercube à mémoire distribuée. Dans [78], Li et Coleman présentent un nouvel algorithme, meilleur que celui de [61] pour le cas d'une

distribution par colonne toujours sur un machine multiprocesseur configurée en hypercube. Le point le plus important de leur étude porte sur les avantages de leur méthode lorsqu'il se trouve dans le contexte plus général de la résolution du problème $Ax = b$ par la décomposition LU. Ils montrent de plus clairement l'important trafic de messages qu'engendre la résolution triangulaire; de l'ordre de n messages pour n^2 opérations flottantes. Leur méthode, entraîne dès que n est suffisamment grand, un recouvrement calcul-communications. Des expérimentations ont été effectuées sur iPSC 16 processeurs.

Natarajan et Pattnaik [85] reprennent ce même algorithme pour l'adapter à un modèle plus simple pour les communications, et surtout ils interdisent le recouvrement calcul-communications (ce modèle permet de décrire un réseau de stations utilisant Express[1]). Puis ils décrivent un nouvel algorithme pipeliné sur un anneau virtuel où ils remplacent les communications de type échange total par des communications entre voisins. Une étude pratique, faite en utilisant les valeurs simulant au mieux les machines actuelles, montre une supériorité de ce deuxième algorithme, excepté dans le cas où le rapport communication sur calcul diminue. Les expérimentations effectuées à la fois sur un réseau en tore de 256 Transputers et sur un réseau en token ring de stations de travail IBM RS6000 confirme le supériorité de l'algorithme pipeliné. Une dernière version est proposée dans [32]. Une comparaison avec les principales méthodes présentées précédemment, est développée par les auteurs. Ils mettent en évidence le caractère d'adapabilité de leur méthode. Des expérimentations sur une *Meiko* permettent d'espérer des efficacités supérieures à 80%.

Toutes ces méthodes utilisent en général le recouvrement des communications par le calcul, mais surtout une granularité à gros grain qui permettent d'atteindre en mode M.I.M.D des efficaité correcte. Comme de plus, la complexité théorique de la résolution triangulaire pour un modèle PRAM sur un réseau comprenant n^3 processeurs est en $\log_2^2(n)$, le mode S.I.M.D. ne semble pas très approprié à ce type de résolution.
Néanmoins, ce problème de résolution triangulaire est une brique de base de l'algèbre linéaire, et sa programmation en mode massivement parallèle est intéressante à étudier dans le cadre d'application à des problèmes réels. Aussi nous allons détailler son implantation sur la Connection Machine.
Dorénavant, nous nous intéressons uniquement au cas de la résolution d'un système triangulaire supérieur pour des raisons de symétrie évidentes. Les mêmes algorithmes s'appliquent avec peu de changement pour la résolution d'un système triangulaire inférieur.

Soit A une matrice $n \times n$ triangulaire supérieure et b un vecteur de IR^n, on cherche la solution x du système $Ax = b$. L'algorithme séquentiel s'écrit sous la

[1]marketed by Parasoft Corporation, Pasadena

forme suivante :

Algorithme IV.1 : Résolution triangulaire supérieure.

Pour i=n jusqu'à 1
  $x_i = b_i$
  Pour j=n jusqu'à i+1
    $x_i = x_i - A_{ij}\, x_j$

  $x_i = \frac{x_i}{A_{ii}}$

1.1. Première implantation : la méthode usuelle.

Supposons être dans un cadre plus général que la résolution triangulaire en elle-même. Soit une application réelle où la résolution triangulaire n'est qu'une étape parmi d'autres. Le placement des données suit alors la règle d'or du parallélisme massif : Une donnée par processeur (ou processeur virtuel).
Nous supposons de plus avoir configuré la Connection Machine en grille à deux dimensions pour garder une certaine cohérence avec le produit matrice-vecteur en ***lisp** (voir 1). La première version de cet algorithme donne :

Algorithme IV.2 : Première version.

Pour i=n jusqu'à 1
  DOT sur la ligne i pour avoir $Somme = \sum_{j=i+1}^{n} A_{ij} x_j$
  Envoi de Somme au processeur contenant x_i
  Calcul de $x_i = b_i - Somme$
  Envoi sur l'ensemble de la colonne i de x_i

Elle suit à la lettre la méthode séquentielle, mais comporte deux inconvénients :

- Pour être efficace, le vecteur b doit être stocké sur la diagonale, ce qui n'est pas habituel sur la Connection Machine.
- le « Dot » s'effectue sur les éléments de la ligne d'indice supérieur à i. Il faut donc effectuer une communication de plus pour envoyer le résultat sur le processeur devant calculer x_i.

Il est facile de trouver une solution aux deux problèmes. Pour le premier, il suffit d'inclure dès le début la valeur de b_i dans la somme. Pour cela, il faut décomposer le Dot, en divisant les différents calculs, afin de les répartir sur l'ensemble des itérations, c'est-à-dire d'effectuer les calculs le long de la colonne et non plus de la ligne.
La solution du deuxième problème est une conséquence de la résolution du premier. En décomposant le Dot, la communication supplémentaire fait partie du schéma global d'une itération.

En effet à chaque itération, le schéma devient :

1) Calcul de x_i.
2) Propager le long de la colonne i la valeur de x_i qui vient juste d'être calculée.
3) Mise à jour de colonne i pour que tous les processeurs connaissent $\sum_{j=i}^{n} A_{ij}x_j$.
4) Transfert de cette somme vers la colonne $i-1$ de gauche pour pouvoir itérer le principe, et calculer x_{i-1}.

La figure IV.23 décrit les quatre étapes d'une itération.

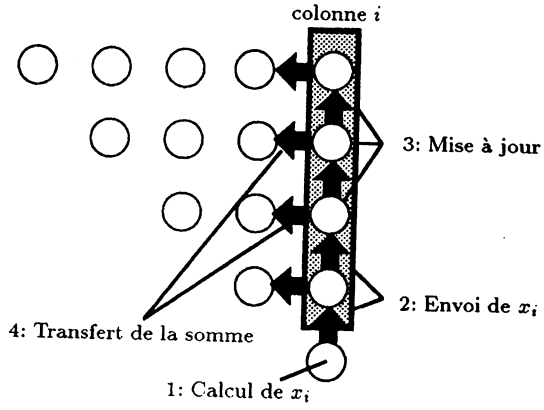


Figure IV.23. Description d'une itération de la première version.

En itérant n fois ce processus, on obtient le vecteur solution $x = (x_1, x_2, \cdots, x_n)$.

L'implantation des deux phases de communication sur la Connection Machine s'obtient de manière différente. La propagation se fait par un *spread-with-copy* utilisant les liens de l'hypercube (cf 2.4.1), qui s'exécute en $\lceil \log_2(i) \rceil$ étapes de communication. Le transfert quant à lui, utilise les communications entre voisins

Algorithme IV.3 : Première version améliorée

$\forall i$ placer b_i dans x_i sur la dernière colonne
Pour i=n jusqu'à 1 faire
  $x_i = \frac{x_i}{A_{ii}}$
  Propager x_i le long de la colonne i
  Mise à jour de x_j pour $j < i$ tels que $x_j = b_j - \sum_{k=i}^{n} A_{jk}x_k$
  Envoi de x_i à son voisin de gauche dans la colonne $i-1$

(*News*, cf 2.3), profitant ainsi de la configuration en grille.

Evaluons le coût théorique de cet algorithme sur la Connection Machine. Pour cela, posons préalablement certaines hypothèses :

- la topologie est une grille $n \times n$.
- L'unité de mesure adoptée correspond sur la Connection Machine au temps de la réalisation d'une instruction élémentaire (i.e. une opération arithmétique).
- le coût d'une communication de type *News* est de l'ordre de 1.
- Le coût d'une communication de type *spread* est en $log_2(n)$ où n désigne le nombre de processeurs concernés par la communication. Cette hypothèse se base sur le fait que l'instruction utilise les liens de l'hypercube lors de son exécution.
- Le VP-Ratio est de 1, c'est-à-dire qu'un processeur physique n'a qu'un élément de la matrice.

La partie la plus coûteuse de l'algorithme est l'envoi du dernier x_i calculé le long de la colonne i. On peut la réaliser soit par des *news*, le coût est alors de i, soit par un *spread* qui est lui de l'ordre de $\lceil \log_2(i) \rceil$. On a bien évidemment choisi la deuxième solution. Les trois autres instructions ayant une complexité de l'ordre de 1, la complexité finale de cet algorithme est de $O(n \log_2(n))$.
Plus précisément, lors de chaque itération, on a trois opérations de coût 1 plus une communication de coût logarithmique, soit un coût final en $3n+n\times(\log_2(n)-1) = n \times (\log_2(n)+2)$.

L'ensemble des exécutions a été effectué en ***lisp**. Le tableau IV.6 présente les résultats obtenus.

Taille du système	Temps d'exécution (s)
8	0,049
16	0,105
32	0,218
64	0.442
128	0.889
256	5.661
512	42.376
1024	324.78

Tableau IV.6. temps d'exécution obtenus sur une Connection Machine à 8K processeurs

Remarque 15 : Bien que tous les processeurs ne soient pas utilisés, dans le cadre d'une étude avec J.Y. Blanc [10] nous avons pris le temps d'exécution sur des petits systèmes linéaires afin de comparer différentes machines ainsi que les deux modes de programmation S.I.M.D. et M.I.M.D..

Remarque 16 : Les expérimentations s'arrêtent pour des matrices de dimension 1024×1024 pour deux raisons :

- la première est due au fait que la résolution triangulaire fait partie d'un ensemble de tâches qui ont pour but la résolution du problème plus général : $Ax = b$. Les autres tâches demandent elles aussi un place mémoire qui diminue la taille maximum du problème possible.

Il faut noter que cette raison n'est pas suffisante car selon les cas, on pourra résoudre des problèmes plus grands. Cette limitation provient surtout de la deuxième raison :

- Et c'est une conséquence de la remarque précédente. En effet, dans le but de comparer certaines méthodes d'algèbre linéaire sur différentes machines parallèles , nous nous sommes limités volontairement aux tailles pouvant être atteintes sur toutes les machines.

Lors de la résolution triangulaire, le nombre d'opérations arithmétiques est de n^2 . On obtient donc les performances (voir figure IV.24).

On s'aperçoit très rapidement que comme nous l'avons dit précédemment, ce type d'algorithme n'est absolument pas adapté au mode massivement parallèle synchrone. La performance crête atteinte n'est que de l'ordre de 0.02 Mégaflops, alors que la puissance crête de la machine est de l'ordre du Gigaflop.
Les mauvaises performances initiales sont une conséquence de la remarque précédente. Le nombre de processeurs utilisés est trop faible pour deux raisons évidentes.

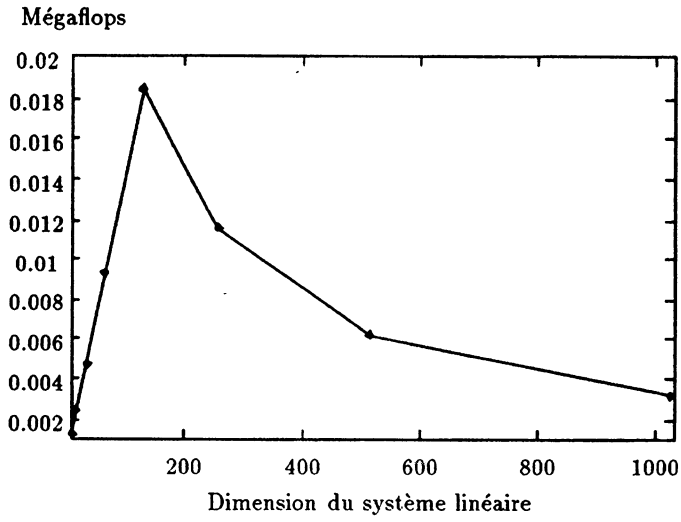


Figure IV.24. Résolution d'un système triangulaire (première version).

- D'une part à cause de la topologie et du placement utilisés, puisque tous les processeurs constituant la moitié inférieure de la matrice sont inactifs.
- D'autre part, une étape de l'algorithme ne fait travailler qu'une colonne à la fois. L'efficacité est encore diminuée. Et même, au pire instant c'est-à-dire lors du calcul du nouveau x_i, on ne fait travailler qu'un seul processeur.

On observe une croissance linéaire des performances jusqu'à un certain seuil. Ce seuil est le moment où le VP-Ratio devient supérieur à 1. Il semble donc normal d'avoir cette linéarité car on augmente la taille du problème par quatre alors que la charge d'un processeur reste identique.
Au delà de cette limite les performances chutent rapidement en raison du VP-ratio de plus en plus important au niveau de chaque processeur physique. Chaque processeur va devoir effectuer plusieurs fois la même opération sur des données différentes.

1.2. Deuxième implantation : la méthode systolique.

L'analyse de la méthode précédente nous montre que l'instruction la plus coûteuse est la propagation le long de la colonne. Notre but est donc de diminuer son coût global afin d'obtenir un coût d'ordre linéaire, à savoir de supprimer le facteur $\log_2(n)$. L'idée de base est de décomposer la communication de type *spread-with-copy* comme lors de la première amélioration.

La solution est une méthode qui s'inspire des algorithmes systoliques de résolution triangulaire décrit dans [89]. Le principe est de décomposer le *spread* en une suite de communications avec les voisins (*News*) de coût théorique 1 et de les répartir dans les étapes suivantes. On va ainsi créer un front qui va avancer dans la grille comme dans un réseau systolique où chaque processeur est une unité de traitement connaissant un nombre minimum d'instructions. Ces instructions sont :

- Envoi au processeur Nord.
- Envoi au processeur Ouest.
- La division pour les processeurs de la diagonale principale
  Le AXPY pour les autres.

La topologie utilisée est comme précédemment une grille de dimension n où l'élément A_{ij} de la matrice est stocké sur le processeur p_{ij}. On travaille donc toujours sur un demi effectif de processeurs physiques.
Détaillons la première itération de l'algorithme sur ce réseau : on va expliquer comment calculer x_n et pouvoir enchaîner afin d'obtenir x_{n-1} lors de la deuxième itération.
La première chose à faire est, avant de commencer la première itération, d'inclure le vecteur b au début de la méthode. Pour cela on le stocke sur la dernière colonne dans la variable *inter*. Au fur et à mesure que cette variable progresse sur la ligne, elle va y accumuler les produits $A_{ij}x_j$. Elle contiendra, donc lorsqu'elle arrivera sur les processeurs possédant les éléments diagonaux, la somme $b_i - \sum_{j=i+1}^{n} A_{ij}x_j$ où i représente l'indice de l'élément diagonal.

Au début de la première itération, le calcul de x_n est immédiat, une seule division suffit pour l'obtenir. A l'étape 1, après avoir effectuer le calcul, on va envoyer la valeur de x_n du processeur de coordonnée (n, n) sur la grille au processeur Nord de coordonnée $(n-1, n)$. Celui-ci va alors pouvoir lors de l'étape 2, mettre à jour la variable *inter*$= b_{n-1} - A_{n-1\,n}x_n$ puis la transmettre au processeur Ouest de coordonnée $(n-1, n-1)$.

Il est alors possible dès la fin de cette étape de calculer le nouvel x_{n-1}, mais impossible d'enchaîner les étapes car, si on propage x_{n-1} au processeur Nord, il n'a pas encore reçu la variable *inter*. Il faut donc pour continuer d'itérer le processus, avoir fait parvenir auparavant la variable *inter* contenant $b_{n-2} - A_{n-2\,n}x_n$ au processeur de coordonnée $(n-2, n-1)$.

Il faut ajouter deux étapes supplémentaires. Dans la première, on va communiquer la valeur de x_n stockée dans processeur $(n-1, n)$ au processeur $(n-2, n)$, puis effectuer la mise à jour de *inter*. Pour terminer, il suffit lors dans une dernière étape de communiquer la variable au processeur Ouest de coordonnée $(n-2, n-1)$.

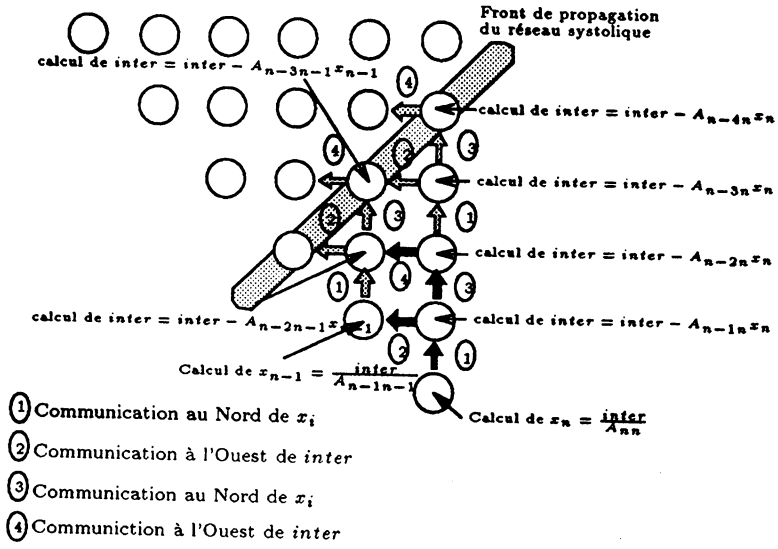


Figure IV.25. Version systolique, description des deux premières itérations.

On peut alors itérer le processus en calculant x_{n-1} puis en l'envoyant au Nord et ainsi connaître les quatre étapes un nouvel x_i. La figure IV.25 schématise les deux premières itérations en décomposant les différentes étapes de communication.

L'algorithme général s'écrit sous la forme:

Algorithme IV.4 : Version systolique

$\forall i$ placer b_i dans $inter$ sur la dernière colonne
Pour i=n jusqu'à 1 faire
  $x_i = \frac{inter}{A_{ii}}$
  Pour tous les processeurs appartenant au front de propagation
    Envoyer x_j à son voisin Nord

  Pour tous les processeurs ayant reçu x_j
    %décalage du front d'un processeur vers le nord
    Mise à jour de $inter$
    Envoyer $inter$ à son voisin Ouest
    Envoyer x_j à son voisin Nord

  Pour tous les processeurs ayant reçu x_j
    %nouveau décalage du front d'un processeur vers le nord
    Mise à jour de $inter$
    Envoyer $inter$ à son voisin Ouest

Evaluons maintenant le coût de cette nouvelle version de la résolution triangulaire: chaque itération est constituée de quatre étapes qui chacune, exceptée la dernière, se compose d'un calcul et d'une communication avec un voisin. D'après nos hypothèses, le coût d'une étape est donc de 2 sauf pour la dernière qui est de 1 car elle ne réalise qu'une communication. Donc le coût d'une itération est de 7, soit un complexité en $O(n)$ pour la résolution en elle-même.
Le coût précis de la résolution triangulaire est en $7n - 2$ car lors de la dernière itération il n'est pas nécessaire d'effectuer les étapes 3 et 4.

Les temps d'exécution obtenus sont donnés dans le tableau IV.7, suivi par la figure IV.26 qui présente les performances sur une Connection Machine 8k processeurs.

Taille du système	Temps d'exécution (s)
8	0,128
16	0,275
32	0,560
64	1.156
128	2,263
256	14,77
512	110,146
1024	1485,561

Tableau IV.7. Temps d'exécution obtenus sur une Connection Machine 8k processeurs.

On observe comme pour la version précédente, les phénomènes suivants: une première croissance linéaire suivi d'une chute dramatique dès que le VP-Ratio devient supérieur à 1. Les raisons sont identiques. Il devient donc plus intéressant de comparer les deux méthodes.

1.3. Comparaison des deux méthodes

L'analyse de la complexité semble donner une préférence à la version systolique qui, avec son coût linéaire, est bien inférieur au $nlog_2(n)$ de l'autre.
Il faut quand même se rappeler que cette supériorité n'est vraie qu'à partir d'un certain rang. En effet, pour que la version systolique soit meilleure il faut que la taille du système à résoudre vérifie la relation suivante:

$$7n > (\log_2(n) + 2) \times n \quad soit \quad \log_2(n) > 5$$

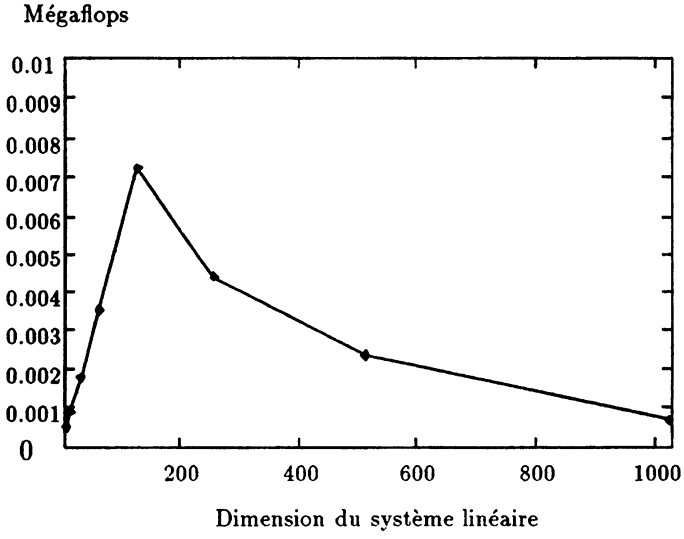


Figure IV.26. Résolution d'un système triangulaire (version systolique)

Donc à partir d'un système de taille 32, la dernière version devrait être théoriquement meilleure que la version améliorée. Or dans la pratique les deux courbes ne se rejoignent jamais. La figure IV.27 nous montre même un rapport constant pour les petites tailles de l'ordre de 2.6 qui va en augmentant pour les tailles très grandes.

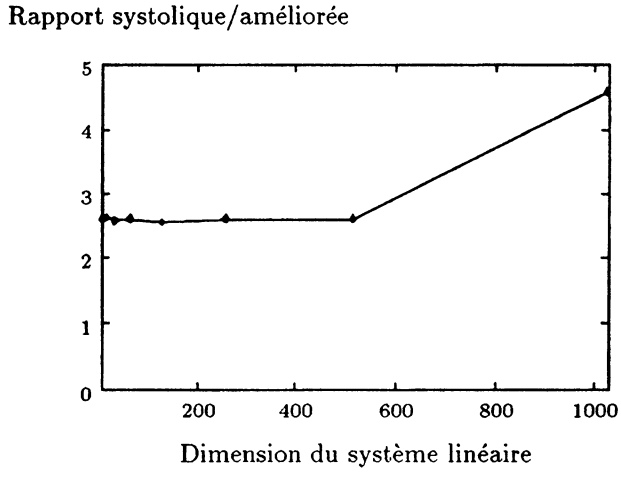


Figure IV.27. Rapport entre la version systolique et la version améliorée.

Pour les systèmes de grandes tailles, cela provient de l'effet du VP-Ratio qui diminue l'importance du coût logarithmique. En effet dans la pratique, dès lors

qu'on a dépassé le seuil d'un VP-Ratio égal à 1, le coût d'un *spread* est logarithmique sur les processeurs physiques alors qu'il est linéaire entre les processeurs virtuels. Le coût devient alors plus fort sur la version systolique de l'ordre de $7n$ par rapport à la version améliorée qui va tendre asymptotiquement vers $4n$.
Autre remarque, le nombre de calculs effectués en même temps est plus important sur la première version que sur celle du réseau systolique. On travaille sur une colonne entière sur la première alors qu'on utilise un faible voisinage pour l'autre. En conséquence, le rapport entre les deux méthodes va s'accentuer avec la taille du problème à résoudre.

Pour les tailles 32 et 64, cela s'explique par les tests nécessaires pour définir les processeurs actifs. Lors de la première version, on a besoin d'un seul test pour rendre actifs seulement les processeurs de la colonne considérée, et d'un deuxième pour le calcul de x_i. Dans la version systolique, la définition de l'ensemble des processeurs actifs est plus compliquée, et demande des tests supplémentaires.

2. Conclusion

De manière générale, la résolution triangulaire a des performances catastrophiques par rapport à celles obtenues dans le chapitre précédent. Il convient donc de faire attention au choix de la méthode lorsqu'on désire résoudre un système. Dans [35], nous avons montré que la méthode de Gauss donne des performances satisfaisantes, mais en enchaînant avec la résolution triangulaire pour obtenir la solution, les performances globales vont chuter dramatiquement.
Donc aux vues des résultats de ces deux derniers chapitres, une conclusion s'impose : les méthodes itératives à base de produits matrice vecteur semblent beaucoup mieux adaptées que les méthodes dites directes à ce type de machine, et même au mode S.I.M.D..

De plus, suite à cette étude, le choix d'une méthode à implanter sur la Connection Machine ne semble pas toujours évident à faire. La complexité n'est pas toujours le meilleur critère. Il vaut mieux souvent choisir en fonction de l'architecture et des structures imposées par les constructeurs de la machine. Cela revient à dire que régulièrement il est préférable de programmer de façon simple sans trop chercher à décomposer les choses.

Chapitre V

Retour à d'anciennes méthodes : Pourquoi pas ?

La résolution des problèmes de taille réel d'algèbre linéaire, c'est-à-dire résoudre des systèmes formés par de très grandes matrices denses ou creuses, est une part importante du travail des mathématiciens et des informaticiens. On la retrouve dans quantité de domaines différents tels que la dynamique des fluides, la modélisation de l'atmosphère, l'analyse de structures. Dans le chapitre IV, nous avons montré que les méthodes dites directes étaient très vite limitées par la taille des systèmes à résoudre. C'est pourquoi, avec l'avènement du parallélisme et des ordinateurs possédant un grand nombre de processeurs, les méthodes itératives comme celle du Gradient Conjugué, connaissent actuellement un fort engouement.

De nombreux travaux ont été effectués sur le Gradient Conjugué, avec ou sans préconditionnement [3, 7, 60, 70]. Cependant, sa parallélisation met en évidence le coût prohibitif des opérations de réduction pour le calcul des paramètres optimaux à chaque itération. Dans ce chapitre, à partir des travaux effectués par L. Desbat [41], nous proposons une analyse de la méthode dans laquelle on remplace les paramètres du Gradient Conjugué par des constantes et rappelons le lien avec la méthode de Richardson du second ordre. Nous comparons les deux méthodes par des expérimentations séquentielles. Puis, nous étudions leur parallélisation et proposons une expérimentation dans le cas creux et régulier du laplacien sur la Connection Machine. Ce travail a été réalisé en collaboration avec L. Desbat et D. Trystram [34].

1. Introduction

Nous étudions dans ce chapitre une parallélisation de la méthode du Gradient Conjugué pour résoudre des grands systèmes linéaires creux tels que ceux qui proviennent de la discrétisation par éléments finis. La parallélisation des méthodes de résolution de ces systèmes a suscité récemment beaucoup de recherches [5, 49, 56, 58, 76, 93]. Dans la section 2, nous remplaçons les paramètres du Gradient Conjugué nécessitant des calculs de réduction (produits scalaires) par des constantes. Nous calculons la borne théorique de l'erreur de cette méthode (le taux de convergence associé aux paramètres constants optimaux correspond à la borne classique donnée pour le Gradient Conjugué). Le lien entre le Gradient Conjugué et les méthodes d'accélération de Tchebycheff sont bien connus. La méthode à paramètres constants optimaux est simplement la méthode de Richardson du second ordre. Nous montrons par des expérimentations que ses performances sont comparables à celles du Gradient Conjugué. Dans la section 3, nous analysons les deux algorithmes et proposons une parallélisation efficace pour les matrices provenant de schémas réguliers. Nous présentons dans la section 4, une implantation sur Connection Machine et comparons les deux méthodes en fonction de la précision.

1.1. Description de la méthode du Gradient Conjugué

La méthode du Gradient Conjugué introduite dans [62], est une méthode très largement utilisée pour la résolution de systèmes linéaires. Elle est particulièrement bien adaptée au cas des larges systèmes creux [5]. Ses propriétés fondamentales ont été étudiées par [56] et ont fait l'objet d'une très large recherche [55, 75].

Considérons le problème de la résolution du système linéaire suivant :

$$A.x = b$$

où A est symétrique définie positive de dimension n et b un vecteur de IR^n.

Le principe de la méthode du Gradient Conjugué est de considérer la résolution du système $Ax = b$ comme le calcul du minimum de la fonction F définie par :

$$F(x) = \frac{1}{2}x^t.A.x - x^t.b$$

En effet, le minimum de F est obtenu au point $x = A^{-1}b$. La solution pour minimiser F est d'utiliser une méthode usuelle de descente. Cela consiste à construire une suite de points x_k qui font décroître la fonction F.
On définit un point de départ x_0. Supposons connaître x_k, alors l'itéré suivant

x_{k+1} est défini par la direction de descente: p_k, et la distance de déplacement le long de la pente: λ_k.
Pour le Gradient, la direction de pente choisie est telle que à chaque étape, p_k indique la plus grande pente, c'est-à-dire la direction opposé au sens du gradient.
Pour le Gradient Conjugué, elle est choisie de telle manière qu'on ajoute à l'opposé du gradient une quantité qui dépend de la direction précédente p_{k-1}.
Le paramètre λ_k est alors défini de manière optimale par:

$$\lambda_k = \frac{r_k^t . r_k}{(A.p_k)^t . p_k}$$

où r_k désigne le vecteur gradient de x calculé pour chaque itéré.

La figure V.28 décrit le choix des deux paramètres P_k et λ_k lors de chaque itération.

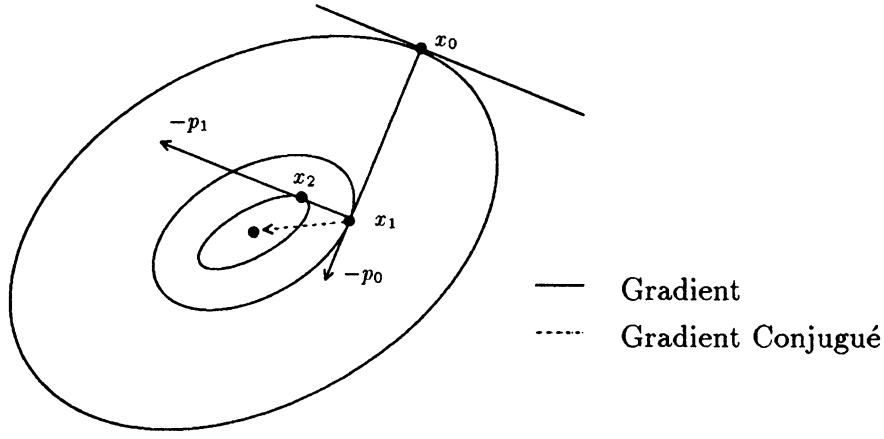


Figure V.28. Principe de la méthode du Gradient Conjugué dans $\mathbb{R}^2$.

L'optimalité de ces deux paramètres induit la propriété fondamentale du Gradient Conjugué.

Propriété 1 : Les directions de descentes successives sont conjuguées 2 à 2 [56].

Cette propriété garantit une convergence de la méthode en au plus n itérations [55, 6].

Notons d_i ses valeurs propres ordonnées comme suit: $0 < d_1 \leq d_2 \leq \ldots \leq d_n$, et $\kappa(A) = d_n/d_1$ désigne le conditionnement du système.

Rappelons l'itération du Gradient Conjugué associé à la matrice A. A partir des vecteurs initiaux: x_0 donné et $p_0 = -r_0 \stackrel{def}{=} b - Ax_0$:

Algorithme V.1 : Gradient Conjugué

$$
\begin{aligned}
&\lambda_k = \frac{\|r_k\|^2}{<Ap_k, p_k>} \\
&x_{k+1} = x_k + \lambda_k p_k \\
&r_{k+1} = r_k + \lambda_k A p_k \\
&\beta_{k+1} = \frac{\|r_{k+1}\|^2}{\|r_k\|^2} \\
&p_{k+1} = -r_{k+1} + \beta_{k+1} p_k \\
&\textit{pour } k = 1, 2, \ldots \textit{ jusqu'a convergence.}
\end{aligned}
\tag{4}
$$

Dans cet algorithme, deux paramètres sont calculés à partir de produits scalaires :

λ_k le paramètre optimal de descente le long de la direction p_k,

et β_k le paramètre optimal de choix d'une direction de descente dans le plan défini par le gradient et la direction de descente précédente.

De nombreux résultats, tant expérimentaux que théoriques, montrent que la vitesse de convergence de la méthode dépend de l'inverse du conditionnement de la matrice $\kappa(A)$. Plus $\kappa(A)$ est proche de 1, meilleure sera la convergence. On peut montrer plus précisément que la convergence est liée à la répartition des valeurs propres [55].

Une amélioration peut être apportée en préparant la matrice pour obtenir une matrice mieux conditionnée. On résout alors le système suivant :

$$L^{-1} A L^t x = L^{-1} b$$

où $L^{-1}AL^t$ est mieux conditionnée que A.

Il existe differents type de préconditionnements, les plus utilisés sont ceux de Cholesky incomplet [56], polynomiaux [93] et diagonaux. L'inconvénient de la méthode du Gradient Conjugué préconditionné est qu'elle demande en plus, lors chaque itération, la résolution d'un système triangulaire. Or on a vu dans le chapitre précédent que les performances chutent rapidement avec la taille du problème à résoudre. Il semble donc inutile, en tout cas en parallélisme massif, d'utiliser ce type de méthode pour accélérer la résolution. Dans [10], une étude plus détaillée est faite dans le cadre d'une application réelle au problème de répartition de charges dans les réseaux.

[57] montre, dans le cas bien particulier de la structure bande, que certains préconditionnement peuvent être employés. Ils comparent le préconditionnement Diagonal et celui de Cholesky incomplet. Pour le préconditionnement de Cholesky incomplet, ils remplacent la résolution triangulaire par une multiplication matrice vecteur, car profitant de la structure bande de la matrice, il est facile de calculer explicitement l'inverse de la matrice comme la somme des puissances :

$$L^{-1} \approx \tilde{L} = (I + L + L^2 + \cdots + L^k)$$

où $k \ll n$.
Ils montrent que sur certains types de machines comme la nouvelle Connection Machine CM5, on retrouve les même caractéristiques qu'en séquentiel, à savoir que le préconditionnement de Cholesky incomplet est nettement meilleur que le Diagonal, même s'il nécessite des opérations supplémentaires, et qu'il réduit considérablement le nombre d'itérations nécessaires pour obtenir la solution. Pour terminer, ils démontrent que ce type d'algorithme s'adapte facilement aux machines parallèles, car il garde une très bonne efficacité même si la taille ou le nombre de processeurs augmente.

2. Méthode à paramètres constants

L'idée que nous développons dans la suite repose sur la remarque suivante : dans l'algorithme du gradient à pas optimal (méthode dite de la plus profonde descente), si nous remplaçons le pas de descente localement optimal à chaque étape par un pas constant optimal, on obtient une méthode itérative dont la vitesse de convergence est la meilleure borne que l'on sache donner au gradient à pas optimal par l'inégalité de Kantorovitch [75]. Les liens entre la méthode du gradient conjugué préconditionné et les méthodes d'accélération de Tchebycheff [24, 56, 59], indiquent que le même phénomène que celui rencontré dans le gradient à pas optimal se produit ici encore : la meilleure borne donnée par les polynômes de Tchebycheff est quasiment atteinte lorsqu'on remplace les paramètres λ_k et β_k par des constantes optimales. Nous rappelons ce résultat dans le paragraphe suivant.

2.1. λ et β constants

De simples manipulations algébriques à partir d'une itération du Gradient Conjugué donnent :

$$r_{k+1} = ((1 + \lambda_k\beta_k/\lambda_{k-1})I - \lambda_k A)r_k - \lambda_k\beta_k/\lambda_{k-1}r_{k-1}$$

Où I est la matrice identité d'ordre n.

Si nous remplaçons respectivement dans les formules de l'algorithme (4) les paramètres λ_k et β_k par des constantes λ et β, on obtient pour l'équation précédente la relation récurrente d'ordre 2 :

$$r_{k+1} = ((1+\beta)I - \lambda A)r_k - \beta r_{k-1}. \tag{5}$$

Soit $A = U^t D U$ où D est diagonale et U unitaire. La suite y_k convergera vers le vecteur nul si les suites

$$y_{k+1}(i) = ((1+\beta) - \lambda d_i)y_k(i) - \beta y_{k-1}(i) \tag{6}$$

convergent vers 0, $\forall i = 1 \ldots n$, ou de manière équivalente si le rayon spectral de la matrice de l'itération suivante est inférieur à 1 :

$$\begin{bmatrix} y_{k+1} \\ y_k \end{bmatrix} = \begin{bmatrix} (1+\beta)I - \lambda D & -\beta \\ I & 0 \end{bmatrix} \begin{bmatrix} y_k \\ y_{k-1} \end{bmatrix} \tag{7}$$

La proposition suivante donne la surface de convergence en β et λ de l'itération (4) à paramètres constants.

Propriété 2 : l'itération (5) converge pour les couples de paramètres constants (λ, β) tels que :

$$-1 < \beta < 1$$
$$0 < \lambda < \frac{2(1+\beta)}{d_n}$$

Preuve: Nous ne présentons que les grandes lignes de la démonstration, les détails se trouvent dans [41].

Soit $\Delta_i = (1+\beta-\lambda d_i)^2 - 4\beta$ le discriminant de l'équation caractéristique E_i : $f^2 - (1+\beta-\lambda d_i)f + \beta = 0$ associée à (6):

- Lorsque $\beta < 0$, on peut remarquer que $\forall i = 1 \ldots n$, $\Delta_i > 0$, donc E_i a deux solutions. Notons $f_i(\lambda, \beta)$ le maximum du module des solutions de E_i, alors

$$f_i(\lambda,\beta) = \begin{cases} f_1^i(\lambda,\beta) \stackrel{def}{=} (1+\beta-\lambda d_i + \sqrt{\Delta_i})/2 \text{ si } 1+\beta-\lambda d_i \geq 0 \\ f_2^i(\lambda,\beta) \stackrel{def}{=} (\lambda d_i - 1 - \beta + \sqrt{\Delta_i})/2 \text{ si } 1+\beta-\lambda d_i \leq 0 \end{cases} \tag{8}$$

  $f_i(\lambda,\beta) = \max(f_1^i, f_2^i) = 1/2(|1+\beta-\lambda d_i| + \sqrt{\Delta_i})$. Il est facile de montrer que lorsque $-1 < \beta < 0$, la vitesse de convergence est plus mauvaise que celle le gradient à pas constant optimal [75].

- Lorsque $\beta \geq 0$ alors :

$$\Delta_i > 0 \Longleftrightarrow |1+\beta-\lambda d_i| > 2\sqrt{\beta}$$

  Le maximum des modules des solutions est la fonction :

$$f_i(\lambda,\beta) = \begin{cases} f_1^i(\lambda,\beta) & \text{si } 1+\beta-\lambda d_i \geq 2\sqrt{\beta} \\ f_2^i(\lambda,\beta) & \text{si } 1+\beta-\lambda d_i \leq -2\sqrt{\beta} \\ f_{12}^i(\lambda,\beta) \stackrel{def}{=} \sqrt{\beta} & \text{si } 1+\beta-2\sqrt{\beta} \leq \lambda d_i \leq 1+\beta+2\sqrt{\beta} \end{cases} \tag{9}$$

Pour que l'itération vectorielle (5) converge, il faut et il suffit que

$$f_{max}(\lambda,\beta) \stackrel{def}{=} \max_{i=1\ldots n} f_i(\lambda,\beta) < 1,$$

avec f_i donné en (8) et (9). $f_{max}(\lambda,\beta)$ est le rayon spectral de la matrice de l'itération (7). L'étude de $f_{max}(\lambda,\beta)$ repose sur l'ordonnancement des courbes f_1^i et f_2^i. On peut montrer que:

- $f_1^i(\lambda,\beta)$ est décroissante en λ et en i sur son intervalle de définition.
  $\lambda' \leq \lambda \Longrightarrow f_1^i(\lambda,\beta) \leq f_1^i(\lambda',\beta)$ et $j \leq i \Longrightarrow f_1^i(\lambda,\beta) \leq f_1^j(\lambda,\beta)$
- $f_2^i(\lambda,\beta)$ est croissante en λ et en i sur son intervalle de définition.
- Pour $\beta > 0$, $f_1^i(\lambda,\beta) \geq \sqrt{\beta}$ et $f_2^i(\lambda,\beta) \geq \sqrt{\beta}$ sur leur intervalle de définition.
- Lorsque $\beta < 0$ fixé, les courbes $f_1^1(\lambda)$ et $f_2^n(\lambda)$ se croisent lorsque:

$$\begin{aligned} & 1+\beta-\lambda_{opt}(\beta)d_1+\sqrt{\Delta_1} = -1-\beta+\lambda_{opt}(\beta)d_n+\sqrt{\Delta_n} \\ \Longrightarrow \quad & \lambda_{opt}(\beta) = \frac{2(1+\beta)}{d_1+d_n} \end{aligned}$$

- Lorsque $\beta > 0$ fixé, elles se croisent en $\lambda_{opt}(\beta)$ si

$$\begin{aligned} & \frac{1+\beta+2\sqrt{\beta}}{d_n} \leq \frac{1+\beta-2\sqrt{\beta}}{d_1} \\ \Longleftrightarrow \quad & (1+\sqrt{\beta})^2 \leq \kappa(A)(1-\sqrt{\beta})^2 \\ \Longleftrightarrow \quad & \sqrt{\beta} \leq \frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}, \end{aligned}$$

Nous pouvons conclure des propriétés précédentes que pour β fixé:

- soit $\beta \leq \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2$

$$f_{max}(\lambda) = \begin{cases} f_1^1(\lambda) & \text{si } 0 < \lambda \leq \lambda_{opt}(\beta) \\ f_2^n(\lambda) & \text{si } \lambda_{opt}(\beta) \leq \lambda \end{cases} \tag{10}$$

- soit $\sqrt{\beta} > \frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$

$$f_{max}(\lambda) = \begin{cases} f_1^1(\lambda) & \text{si } 0 < \lambda \leq \frac{1+\beta-2\sqrt{\beta}}{d_1} \\ \sqrt{\beta} & \text{si } \frac{1+\beta-2\sqrt{\beta}}{d_1} \leq \lambda \leq \frac{1+\beta+2\sqrt{\beta}}{d_n} \\ f_2^n(\lambda) & \text{si } \frac{1+\beta+2\sqrt{\beta}}{d_n} \geq \lambda \end{cases} \tag{11}$$

Il est alors facile de montrer que

$$|\beta| \geq 1 \Longrightarrow f_{max}(\lambda) \geq 1,$$

et que

$$\text{Si } |\beta| < 1 \text{ alors } f_{max}(\lambda) < 1 \Longleftrightarrow 0 < \lambda < \frac{2(1+\beta)}{d_n}.$$

□

Propriété 3 : le couple de paramètres constants

$$\beta_{opt} = \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2$$

$$\lambda_{opt} = \frac{2(1+\beta_{opt})}{d_1+d_n}$$

donne une vitesse de convergence optimale

$$\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$$

Preuve: D'après ce qui précède, pour $-1 < \beta \leq \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2$ fixé, la vitesse de convergence optimale est réalisée au point $\lambda_{opt}(\beta)$ et est donnée par

$$f_1^1(\lambda_{opt}(\beta),\beta) = 1/2\left(1+\beta-\frac{2(1+\beta)}{d_1+d_n}d_1 + \sqrt{\left(1+\beta-\frac{2(1+\beta)}{d_1+d_n}d_1\right)^2 - 4\beta}\right)$$

Cette fonction est décroissante en β et atteint son minimum en la borne $\beta_{opt} = \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2$:

$$f_1^1(\lambda_{opt}(\beta_{opt}),\beta_{opt}) = \sqrt{\beta_{opt}} = \frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$$

Lorsque $\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2 < \beta < 1$ la vitesse optimale de convergence est réalisée sur l'intervalle

$$\frac{1+\beta-2\sqrt{\beta}}{d_1} \leq \lambda \leq \frac{1+\beta+2\sqrt{\beta}}{d_n}$$

et vaut $\sqrt{\beta}$. La vitesse de convergence devient donc croissante en fonction de β

□

On peut illustrer les deux propositions qui précèdent par une représentation de f_{max} à travers les courbes f_1^i, f_2^i, f_{12}^i pour différents β. Nous présentons le cas d'une matrice de spectre $\mathrm{Sp}(A) = \{2,3,4\}$, $(n=3)$.
On peut remarquer que lorsque β est relativement petit, nous sommes proche

de la situation du gradient à pas constant. On voit dans le premier graphique de la figure V.29, que lorsque $\beta < 0$, les courbes f_1^i, f_2^i sont au dessus des courbes $|1 - \lambda d_i|$. Lorsque β augmente, $f_1^1(\lambda_{opt}(\beta), \beta)$ diminue: le deuxième graphique de la figure V.29 montre que $f_{max}(\lambda_{opt}(0.02), 0.02)$ est plus petit que $f_{max}(\lambda_{opt}(-0.01), -0.01)$. La figure V.30 montre la situation pour β_{opt}. Lorsque $\beta > \beta_{opt}$, $f_{max}(\lambda_{opt}(\beta), \beta) = \sqrt{\beta}$ croît. Dans le deuxième graphique pour $\beta = 0.1 > \beta_{opt}$, nous constatons que $f_{max}(\lambda_{opt}(0.1), 0.1) > f_{max}(\lambda_{opt}(\beta_{opt}), \beta_{opt})$, ceci $\forall \lambda_{opt}(0.1) \in [(1.1 - 2\sqrt{.1})/2, (1.1 + 2\sqrt{.1})/4]$.

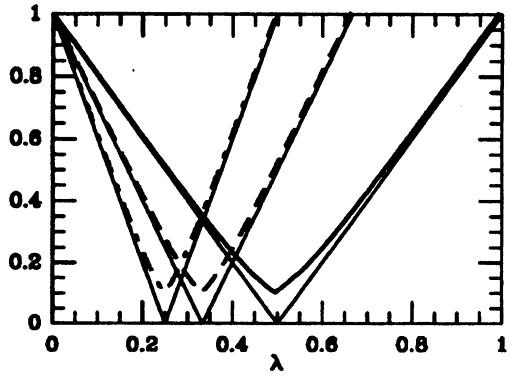
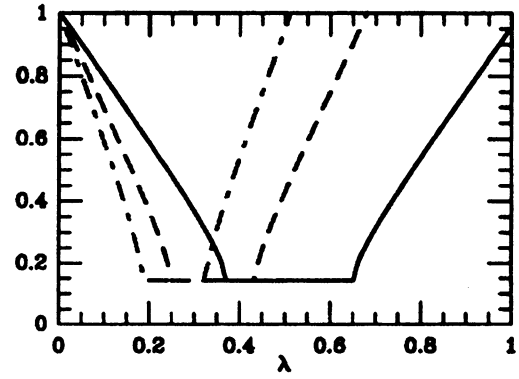


Figure V.29. $\beta = -0.01$.



$\beta = 0.02$.

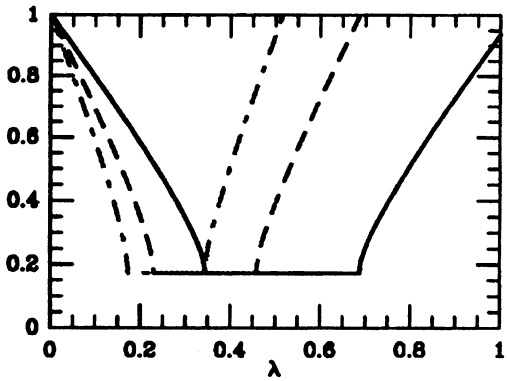
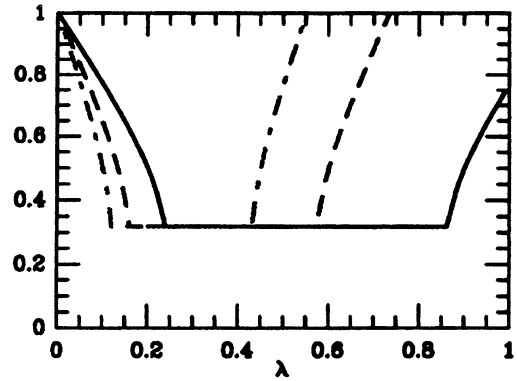


Figure V.30. $\beta = \beta_{opt}$.



$\beta = 0.1$.

2.2. Lien avec la méthode de Richardson

On peut identifier cette méthode avec celle de Richardson du deuxième ordre [56]. Elle dérive de l'accélération de Tchebycheff de la méthode du gradient à pas constant.

$$x_{k+1} = \omega_{k+1}(-\alpha r_k + x_k - x_{k+1}) + x_{k+1} \tag{12}$$

Avec

$$\omega_1 = 1, \omega_2 = \frac{2}{2-\rho^2}, \omega_{k+1} = \frac{1}{1-\omega_k \rho^2/4}$$

où ρ est le rayon spectral de la matrice d'itération de la méthode accélérée: ici $\rho(I-\alpha A) = \frac{\kappa(A)-1}{\kappa(A)+1}$ pour le choix optimal $\alpha = \frac{2}{d_1+d_n}$. On remplace les paramètres ω_k par le paramètre constant optimal:

$$\begin{aligned} \omega &= \frac{2}{1+\sqrt{1-\rho^2}} \\ &= \frac{2}{1+\sqrt{\frac{(\kappa(A)+1)^2-(\kappa(A)-1)^2}{(\kappa(A)+1)^2}}} \\ &= \frac{2(\kappa(A)+1)}{(\sqrt{\kappa(A)}+1)^2} \end{aligned}$$

En multipliant l'équation (12) par A et en retranchant b, on obtient:

$$r_{k+1} = (\omega - \omega\alpha A) r_k + (1-\omega) r_{k-1}$$

Posons donc $\beta = \omega - 1$ et $\lambda = \omega\alpha$, on obtient

$$x_{k+1} = (1 - \beta - \lambda A) r_k - \beta r_{k-1}$$

On vérifie alors que

$$\beta = \omega - 1 = \frac{2(\kappa(A)+1) - (\sqrt{\kappa(A)}+1)^2}{(\sqrt{\kappa(A)}+1)^2} = \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^2$$

$$\lambda = (1+\beta)\alpha = \frac{2(1+\beta)}{d_1+d_n}$$

Remarque 1 : Nous savons (cf. [75]) que le méthode du gradient à pas optimal a pour taux de convergence au moins $\frac{\kappa(A)-1}{\kappa(A)+1}$, plus précisément que:

$$||x_k^g - x^*||_A < \left(\frac{\kappa(A)-1}{\kappa(A)+1}\right)^k ||x_0^g - x^*||_A \tag{13}$$

avec $||x||_A = x^t A x$, x_k^g le k^{ieme} itéré de la méthode du gradient à pas optimal, x^* la solution du système linéaire à résoudre. On sait choisir λ_k constant optimal ($\lambda = 2/(d_1+d_n)$) dans la méthode du gradient à pas constant pour avoir le taux de convergence $\frac{\kappa(A)-1}{\kappa(A)+1}$. On obtient la méthode de Richardson du premier

ordre conduisant à la même majoration (13) dans laquelle on remplace x_k^g par x_k^{co}, le k^{ieme} itéré de la méthode du gradient à pas constant optimal. Dans le cas du gradient conjugué, l'optimalité sur les espaces de Krylov successifs conduit après quelques majorations au taux de convergence d'au moins $\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$ et plus précisément à la majoration :

$$||x_k^{gc} - x^*||_A < 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k ||x_0^{gc} - x^*||_A \qquad (14)$$

où x_k^{gc} est le k^{ieme} itéré de la méthode du gradient conjugué. On vient de voir que l'on sait choisir les paramètres λ_k et β_k constants optimaux dans un algorithme semblable au gradient conjugué pour que le taux de convergence soit $\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$.

La méthode de Richardson du second ordre est donc au Gradient Conjugué ce que la méthode de Richardson du premier ordre est à la plus profonde descente.

Puisque la méthode de Richardson du second ordre construit une itération dans les espaces de Krylov, le GC est plus performant en séquentiel au sens de la norme $||.||_A$. Par contre, l'introduction de paramètres constants supprime les produits scalaires et l'itération est intrinsèquement plus parallèle.

2.3. Expérimentations numériques

Nous avons comparé la méthode du Gradient Conjugué et la méthode de Richardson du second ordre (paramètres constants), dans le cas où A est la discrétisation classique du Laplacien bi-dimensionnel sur une grille carrée de différentes tailles ($n \times n$ avec $n = 60, 100, 200, 300$). Pour les expérimentations, le second membre **b** est tel que la solution de **Ax=b** soit le vecteur dont toutes les composantes sont égales à 1. Nous représentons dans les 4 courbes de la figure V.32, le nombre d'itérations en fonction de la précision relative sur le résidu r. Les courbes en pointillé correspondent au Gradient Conjugué, celles en trait plein à la méthode avec paramètres constants et optimaux. Les résultats représentent le nombre d'itérations et non pas le temps séquentiel car l'objectif est l'implémentation sur une architecture parallèle distribuée (une itération de la méthode à paramètres constants coûte en séquentiel un peu moins cher qu'une itération de gradient conjugué et en parallèle théoriquement beaucoup moins cher).

Le comportement superlinéaire caractéristique du Gradient Conjugué apparaît clairement. La méthode usuelle avec β_{opt} et λ_{opt} reste linéaire. Il faut pourtant souligner que cette méthode atteint la précision de la machine avec environ trois fois plus d'itérations seulement que le Gradient Conjugué (en double précision sur une DECstation 3100). La méthode à paramètres constants est donc très efficace. On peut probablement espérer remplacer avantageusement le Gradient Conjugué sur une machine parallèle à architecture distribuée, car elle ne néces-

site que des communications locales. En particulier si la précision désirée n'est pas très grande, on aura intérêt à l'utiliser puisque les deux méthodes ont un comportement presque similaires (au moins dans cet exemple où le spectre est uniformément réparti).

Le problème de l'estimation des valeurs propres extrêmes peut être résolu par différentes méthodes. Une première approche consiste à mettre en œuvre des méthodes hybrides analogues à celles présentées dans cet article. En général on commence par un petit nombre d'itérations du Gradient Conjugué, puis on poursuit par des itérations de type Tchebycheff. D'autre part, des méthodes concurrentes du type polynômes de moindres carrés donnent de meilleurs résultats pour la localisation des valeurs propres [64]. Une majoration de l'intervalle des valeurs propres de type Gershgorin est alors suffisante. Dans le cas du Laplacien, l'intervalle $[0, 8]$ peut-être utilisé. On peut remarquer dans la figure V.31 qu'une surestimation de β et une sous-estimation de λ est meilleur que le contraire.

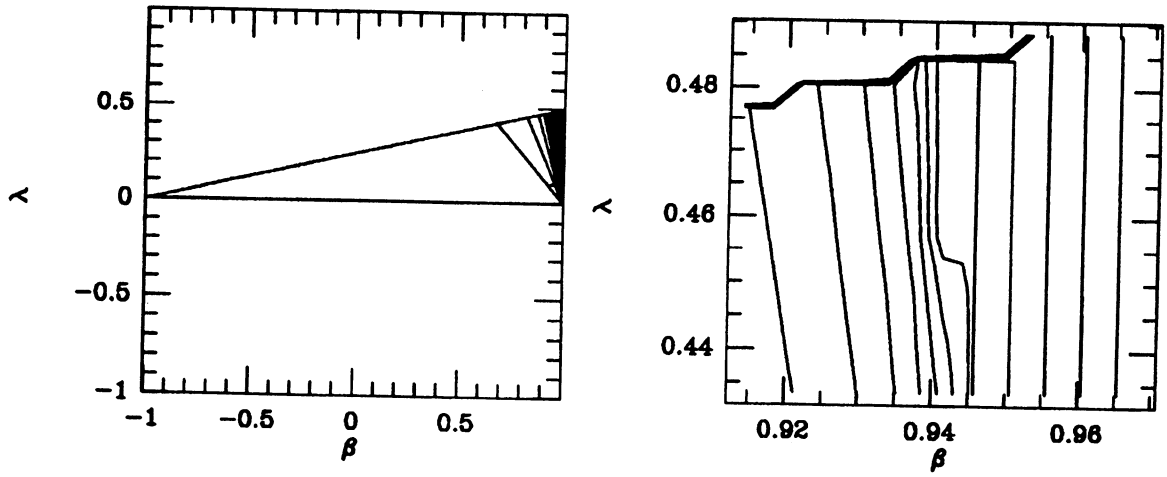


Figure V.31. $f_{max}(\beta, \lambda)$ dans le cas du Laplacien $n = 100$. A gauche nous représentons les courbes de niveau de $f_{max}(\beta, \lambda)$ de 0,97 (légèrement supérieur au minimum) à 1 par pas de 0,0025. A droite nous proposons un agrandissement autour du minimum. On peut remarquer qu'une sous-estimation de β et une surestimation λ peuvent-être dramatiques (le contraire l'étant moins).

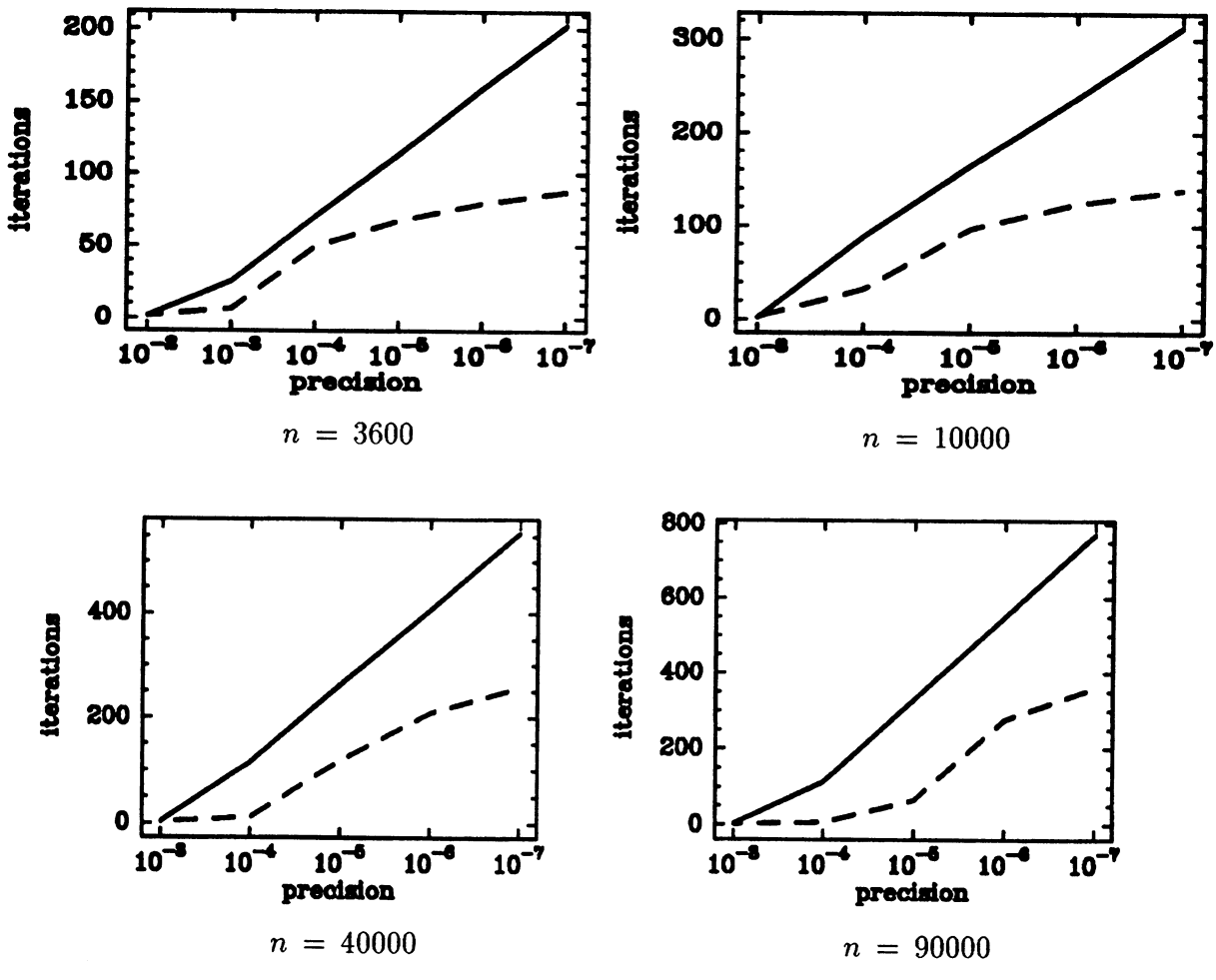


Figure V.32. Comparaison du Gradient Conjugué et de la méthode à paramètres constants pour différentes valeurs n.

3. Analyse Parallèle de la méthode à paramètres constants

3.1. Décomposition de l'algorithme du Gradient conjugué.

Le projet LINPACK (LINear algebra PACKage) [40] créé il y a une vingtaine d'années, a pour objectif de fournir à la fois des outils de mesures des programmes scientifiques et des outils généraux de production de gros logiciels. Il a conduit à la création d'une bibliothèque de procédures en algèbre linéaire.

On distingue 3 niveaux d'opérations fondamentales (noyaux des méthodes numériques) notées BLAS1, BLAS2 et BLAS3 en fonction de l'ordre du nombre d'opérations effectuées (n, n^2, n^3)[43]. Cette identification permet une implantation efficace et simple à mettre en œuvre grâce à sa conception modulaire. Cette bibliothèque comprend tous les éléments de base de l'algèbre linéaire: le Dot, le AXPY (BLAS1), le produit matrice-vecteur ou la modification de rang 1 (BLAS2), l'élimination de Gauss ou encore le produit matrice matrice (BLAS3). Le Dot et le AXPY sont les procédures de base de l'algèbre linéaire.
Leurs définitions sont les suivantes:

Algorithme V.2 : Dot

$s \leftarrow 0$
pour i $\leftarrow$ 1 jusqu'a n
    $s \leftarrow s + x(i) * y(i)$

Algorithme V.3 : AXPY

pour i $\leftarrow$ 1 jusqu'a n
    $s(i) \leftarrow a * x(i) + y(i)$

Ces deux opérations requièrent le même nombre d'opérations en séquentiel. On peut décrire l'algorithme du Gradient Conjugué en utilisant la notion de BLAS et ainsi le décomposer en procédures qui se parallélisent facilement [44]. L'itération de base de l'algorithme du Gradient Conjugué s'écrit alors:

Algorithme V.4 : Version BLAS du Gradient Conjugué

1 Produit Matrice-vecteur: Calcul de $A * p_k$
1 Dot suivi d'un calcul scalaire
2 AXPY pour la mise a jour de x_k et r_k
1 Dot suivi d'un calcul scalaire
1 AXPY pour la mise a jour de p_{k+1}

Excepté le produit matrice-vecteur, toutes les procédures utilisées sont de niveau BLAS1. Ce produit qui appartient au niveau BLAS2 peut être lui-même décomposé en n étapes de niveau 1 (cela dépend du stockage employé).

3.2. Etat de l'art

Le Gradient Conjugué est très utile pour résoudre de grands systèmes linéaires. Il permet d'utiliser au mieux la structure creuse des matrices contrairement aux méthodes directes comme l'élimination de Gauss qui ne permettent pas en général de respecter le stockage initial. Plusieurs solutions ont été proposées pour paralléliser le Gradient Conjugué. La plupart de ces travaux portent sur un découpage macroscopique en blocs sur des machines à gros grain. Soit par exemple en utilisant une décomposition locale en sous-domaines [49, 58, 76], soit avec une approche plus centralisée qui correspond simplement à la parallélisation du produit matrice-vecteur [10, 48, 106].

Saad a proposé une approche plus fine [93]. Il a mis en évidence que la chute de l'efficacité d'une itération du Gradient Conjugué était dû aux deux points de synchronisation (conséquence des 2 Dot). Des manipulations algébriques sur les formules de base conduisent à une solution où l'on peut remplacer les 2 Dot par trois consécutifs soit un seul point de synchronisation par itération. Cependant, cette méthode n'est pas stable numériquement.
Une autre solution est proposée par Chronopoulos et Gear [19] ainsi que Adams [1]. Elle consiste en une méthode plus générale: La méthode s-pas du gradient conjugué [50]. Cette méthode revient à minimiser l'erreur fonctionnelle sur des plans affines « direction » bien précis (cf [19]), et non plus dans un seul vecteur direction comme le fait la méthode du Gradient Conjugué (remarque: une étape de cette méthode est équivalente à s étape du Gradient Conjugué). De plus, cette méthode est stable numériquement pour toute valeur de $s \leq 10$. Un de ses avantages est de diminuer les communications par un placement local plus efficace qui lors d'expérimentations fait que l'on observe une forte potentialité en parallélisme. Les résultats se trouve dans [18, 17]. Ils montrent que sur des machines parallèles mais surtout possédant au niveau du nœud un processeur vectoriel très puissant (Cray, Alliant) alors les performances sont meilleures, un rapport 1.3 entre les deux. En contre partie, cette méthodes demande un surcoût en nombre d'opérations à effectuer, comme par exemple la résolution du mini-système $s \times s$ pour obtenir les coefficients définissant le plan de descente. Ce coût supplémentaire devient trop important lorsqu'on diminue le niveau du grain de parallélisme comme nous le verrons par la suite.

En parallélisme massif distribué où chaque élément de la matrice est placé sur un processeur différent, seule une étude microscopique peut être employée. Si l'on analyse le niveau BLAS1, on s'aperçoit que le Dot nécessite des communications

entre les processeurs pour rassembler toute l'information, entraînant ainsi un coût supplémentaire. Par contre, un AXPY ne fait que des opérations locales à un processeur sans communication. Un Dot est donc plus coûteux en parallèle qu'un AXPY en raison des communications supplémentaires.

3.3. Parallélisation de la méthode à paramètres constants

La méthode à paramètres constants évite les deux Dot en supprimant dans l'itération de base les calculs des scalaires λ_k et β_k, et en les remplaçant par des paramètres constants sans augmenter le volume de calcul.

Par contre, il subsiste le problème du calcul du produit matrice-vecteur. Dans le cadre du parallélisme massif sur une grille à deux dimension, le produit matrice-vecteur plein demande l'utilisation de n Dot réalisés en parallèle. Comme de plus, le vecteur résultat n'est pas en général, placé de la même manière que le vecteur initial, il est alors nécessaire de déplacer des éléments pour effectuer les AXPY ce qui entraîne des communications supplémentaires, et un coût plus important qu'un Dot. Le cas du produit matrice-vecteur plein devient donc inutile à étudier car le coût en communication est trop élevé, et le gain obtenu sur les deux points de synchronisation sera négligeable.
Le cas creux est intéressant à étudier pour trouver un stockage approprié avec la structure à la fois du problème à traiter mais aussi de l'architecture de la machine cible qui obtienne un gain de temps sensible [97].

4. Implantation

4.1. Description

Tout d'abord, nous avons mesuré le temps d'exécution des deux méthodes: Gradient Conjugué et Richardson, dans le cas d'une matrice pleine. Là, on est obligé d'utiliser le produit matrice vecteur décrit dans le chapitre III. Les résultats en ***Lisp** sont les suivant :

Taille	Gradient Conjugué	Richardson
128	0.86	0.81
256	3.48	2.6
512	12.47	9.95
1024	47.75	39.24

Tableau V.8. Temps en seconde dans le cas plein.

On remarque que le gain obtenu entre les deux méthodes est très faible. Cela s'explique facilement par la prépondérance du produit matrice vecteur dans le

coût d'une itération. En effet, le fait de supprimer trois produits scalaires a peu d'influence sur les n nécessaire pour effectuer le produit. Le cas d'une matrice pleine ne permettant pas d'espérer un gain de temps, nous nous sommes uniquement intéressés au cas creux qui représente l'essentiel des utilisations de la méthode du Gradient Conjugué. Nous considérons la résolution du problème du Laplacien dans les mêmes conditions que dans le cas séquentiel.

Il faut donç dans le cas creux, créer un nouveau produit matrice-vecteur qui soit plus rapide qu'un produit scalaire. Dans le cas du Laplacien, la matrice est issue du maillage en grille du problème considéré. A chaque pas, un nœud du maillage reçoit une valeur de ces 4 voisins pour calculer sa nouvelle valeur comme le montre la figure suivante.

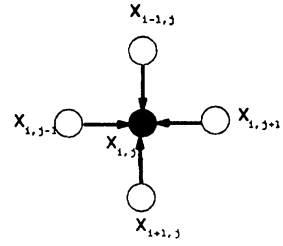


Figure V.33. calcul au niveau d'un nœud

C'est ce schéma que nous avons directement reproduit sur la Connection Machine configurée en grille 2-D. Un processeur correspond à un nœud du maillage. Cette méthode a deux avantages importants: un gain de place et un gain de temps.

- Le gain de place se traduit par une absence complète de matrice, il ne reste que le vecteur à stocker. Cela permet de résoudre des problèmes de taille très supérieure et donc de pouvoir traiter des problèmes réels.
- Le gain de temps est dû à la structure régulière du maillage du Laplacien. Une itération du produit matrice-vecteur demande un échange de messages entre tous les voisins sur la grille. Ce type de communication correspond au niveau le plus rapide: les communications NEWS sur la Connection Machine.

De manière plus générale, on peut appliquer le même raisonnement sur tous les schémas "réguliers". Des exemples d'implantations se trouvent dans [82]. Le seul problème à signaler provient du caractère S.I.M.D. de la machine qui empêche de réaliser cette communication en une seule instruction. En effet, ne pouvant effectuer qu'une opération à la fois, il faut la décomposer en quatre communications, une selon chaque point cardinal, soit quatre instructions.

L'algorithme du produit matrice-vecteur creux s'écrit:

Algorithme V.5 : Produit Matrice-Vecteur du Laplacien

$x_{k+1} = 4{*}x_k$
si le nœud n'appartient pas au bord Nord
alors reçoit x_k du voisin nord dans x_{trans}
$x_{k+1} = x_{k+1} - x_{trans}$

si le nœud n'appartient pas au bord Sud
alors reçoit x_k du voisin sud dans x_{trans}
$x_{k+1} = x_{k+1} - x_{trans}$

si le nœud n'appartient pas au bord Ouest
alors reçoit x_k du voisin ouest dans x_{trans}
$x_{k+1} = x_{k+1} - x_{trans}$

si le nœud n'appartient pas au bord Est
alors reçoit x_k du voisin Est dans x_{trans}
$x_{k+1} = x_{k+1} - x_{trans}$

Les temps obtenus dans le cas creux sont les suivants :

Taille	Gradient Conjugué	Richardson
128	20.76	8.84
256	49.36	25.2
512	157.84	91.8
1024	568.6	353.44

Tableau V.9. Temps en seconde dans le cas creux.

Les temps sont plus importants que dans le cas plein car dans le cas creux on traite un problème de taille n^2; mais proportionnellement, le gain de temps pour obtenir une composante du vecteur solution est très important. De plus, comme nous l'avons décrit précédemment (cf A.2), les communications NEWS sont les plus rapides sur la machine car elles n'entraînent aucun conflit sur les liens. Le gain de temps obtenu entre les deux méthodes est sensible par rapport au cas plein qui nécessite le calcul de n produits scalaires (communication avec recombinaison).

Mais une question importante subsiste: une itération de Richardson coûte-elle beaucoup moins chère qu'une itération du Gradient Conjugué et peut-elle compenser l'économie du produit scalaire?

Avant de répondre à cette question par les expérimentations, il faut remarquer qu'il existe une différence pour réaliser un Dot entre le cas d'un stockage plein et le cas creux. N'ayant plus de matrice à conserver, le vecteur X est stocké un élément par nœud de la grille. Le produit scalaire s'effectue alors en deux étapes: un premier produit scalaire sur les lignes suivi d'un deuxième sur les colonnes pour obtenir le résultat final.

4.2. Expérimentations

La programmation des deux méthodes a été réalisée en premier en ***Lisp**, puis comme nous le montrent les deux figures suivantes, nous sommes passé au langage **ParIS** car les performances sont sensiblement accrues d'un langage à l'autre.
Les commentaires sont identiques entre le cas creux et le cas plein. Une comparaison rapide des deux courbes permet de montrer une accélération sensible des performances entre les deux langages. Il parait assez clair que le langage *Lisp nécessite des tests supplémentaires vu son caractère de langage de haut niveau. Cette constatation est encore plus flagrante en regardant uniquement la courbe de la méthode de Richardson du second ordre qui est proportionnellement meilleure (facteur 4) (Cf figures V.34 et V.35). C'est pourquoi nous restreindrons nos commentaires au cas des programmes écrits en ParIS.
Les expérimentations ont été faites en fonction de la taille du problème à traiter. Les résultats obtenus à nombre d'itérations identiques ($n_i = 2000$), sont donnés dans les figures V.34 et V.35.

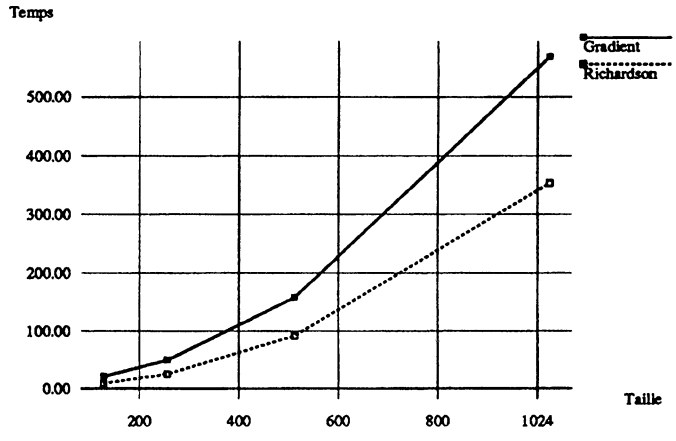


Figure V.34. Comparaison des temps d'exécution en *Lisp

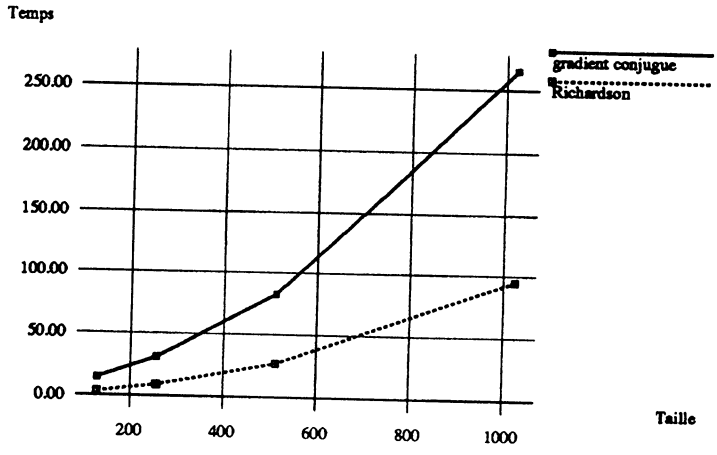


Figure V.35. Comparaison en ParIS

On remarque que dans les deux cas, la méthode à paramètres constants est meilleure que celle du Gradient Conjugué. Le gain est nettement supérieur à celui obtenu dans le cas plein. Cela s'explique par la fait que le produit matrice-vecteur creux prend beaucoup moins de temps que dans la cas plein (1 produit scalaire + communications pour remettre le vecteur en place).

On observe également que proportionnellement le rapport entre les deux méthodes sur les deux figures V.34 et V.35 diminue avec la taille. Ce phénomène est dû au VP-ratio de la CM2. Son influence est d'ordre linéaire sur le produit matrice vecteur tandis qu'elle est d'ordre logarithmique sur les produits scalaires.

Sachant que les deux méthodes n'ont pas le même type de convergence, la méthode à paramètres constants est-elle meilleure que le Gradient Conjugué pour ce type d'implantation sur des schémas réguliers?

Pour répondre à cette question, nous avons tout d'abord étudié la convergence des deux méthodes pour différentes précisions. Connaissant le résultat exact, il est facile de compter le nombre d'itérations nécessaires pour obtenir une précision donnée en utilisant la norme infinie.

$$precision = \| x_k - x^* \|_\infty$$

Les résultats obtenus sont présentés sous la forme du rapport du nombre d'itérations obtenues pour la méthode à paramètres constants sur celui de la méthode du Gradient Conjugué.

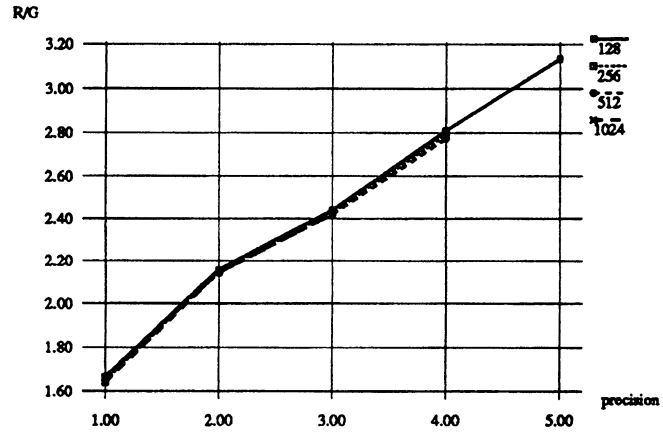


Figure V.36. Rapport entre les deux méthodes à itération fixée

On remarque que comme dans le cas séquentiel pour une faible précision, le rapport entre les deux méthodes reste faible, et plus on désire une précision élevée plus le rapport devient important. On constate de plus que la taille du problème considéré n'influe en rien sur la convergence des méthodes.
Cette étude de la convergence étant faite, observons ces influences sur le gain de temps obtenu entre les deux méthodes. En regroupant les deux dernières figures V.35 et V.36, on obtient la courbe V.37 :

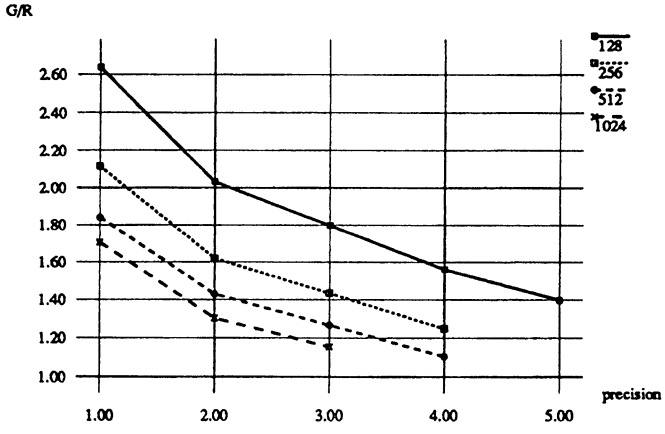


Figure V.37. Gain final obtenu par la méthode à paramètres constants

L'influence de la taille du problème à traiter (VP-ratio), observée dans la figure V.35, est encore plus flagrante sur cette courbe. Excepté le cas 128 qui permet d'obtenir un gain satisfaisant, quelle que soit la précision demandée, les performances se dégradent rapidement avec la taille. Des expérimentations sur une CM2 de taille plus importante, atténueraient certainement cette dégradation. On remarque cependant que pour toutes les mesures effectuées, la méthode à paramètres constants reste meilleure que celle du Gradient Conjugué.

Pour finir, lors des expérimentations sur la Connection Machine, nous avons constaté qu'en raison à la fois de la taille du problème et surtout de la taille du processeur flottant (32-bit), la précision maximum ne pouvait dépasser 10^{-5} dans le meilleur des cas. Ayant accès à une autre machine ayant elle des processeurs flottants de 64 bits, il a été possible de tester avec un peu plus de précision. Le gain reste néanmoins assez faible. En moyenne, nous avons seulement gagné une décimale supplémentaire, parfois deux.
Mais quelle est l'influence des processeurs flottants double précision ? C'est à cette question que nous allons maintenant essayer de répondre. Les temps obtenus sur la Connection Machine avec 4K-processeurs sont les suivants :

Nbre d'itérations	Cas Plein	Cas Creux
50	0.42	0.39
100	0.92	0.77
250	2.5	1.92
500	5.15	3.83
1000	10.5	7.66
5000	54.11	38.27
8192	89.12	62.73

Tableau V.10. Comparaison en seconde du cas plein et du cas creux avec des processeurs flottants 64 bits.

Les commentaires sont identiques quelque soit la taille des processeurs flottants. La seule différence concerne le temps d'exécution d'une itération. En comparant à Vp-ratio identique, le rapport entre les deux exécutions est de l'ordre de 1.7. Donc pour estimer le temps d'exécution de la méthode de Richardson sur une machine de taille supérieure mais en double précision, il ne faut pas tout à fait doubler les temps obtenus.

5. Conclusion

De toutes les expérimentations effectuées sur la Connection Machine dans le cas du Laplacien et pour tous schémas réguliers, on peut extraire deux grandes lignes : premièrement le manque de précision de la machine, et deuxièmement l'effet du VP-ratio.
Le phénomène de précision est dû à l'architecture même de la Connection Machine et ses processeurs flottants 32 bits, et peut être pondéré comme on vient de le voir, par l'utilisation de processeurs 64 bits, même si le temps d'exécution est presque doublé.
L'effet du VP-ratio est aussi lié au problème de l'architecture des machines S.I.M.D., mais son atténuation semble possible avec les projets de construction

de machines avec un million de processeurs. On peut donc envisager l'avenir avec beaucoup d'espoir. Mais pour l'instant la méthode à paramètres constants est très efficace pour une classe de problèmes correspondant à la résolution de grands systèmes linéaires creux avec relativement faible précision comme en traitement du signal ou comme noyau de base de méthodes non-linéaires.
[57] montrent, dans le cas régulier d'une matrice penta-diagonale obtenue par discrétisation en dimensions d'un problème d'équation différentielle partielle, l'efficacité du préconditionnement de Cholesky incomplet si le produit matrice vecteur s'exécute de façon rapide sans recouvrement. Qu'en est-il s'ils utilisent au mieux la structure bande de la matrice, et définissent un nouveau produit matrice vecteur encore plus efficace ?

Le seul inconvénient de ce type de programmation est qu'il demande un réseau de communication entre voisins très efficace et très important, ce qui n'est pas encore toujours le cas pour les machines actuelles. Efficace car par exemple, dans le cas du Laplacien, il serait intéressant de pouvoir regrouper les quatre voisins en un seul top. Important, pour pouvoir modéliser plus facilement et plus rapidement les différents maillages issus des problèmes de différences finies ou d'éléments finis. Le réseau Xnet de la MasPar semble être un début de réponses à ces demandes [88].

Chapitre VI

Traitement du Signal et Parallélisme

1. Introduction

Dans ce chapitre, nous nous intéresserons au problème de la modélisation d'un signal, c'est-à-dire à la paramétrisation de la source afin de pouvoir par exemple prédire son évolution, ou la contrôler. Plus particulièrement, nous étudierons le problème de la prédiction linéaire d'un signal. Il existe deux grandes classes de solutions : soit on utilise directement le signal discrétisé (méthode de type "Treillis") [81], soit on résoud un grand système linéaire (méthode d'autocorrélation) [39]. L'algorithme de Burg [14] appartient à la première classe de solutions. Sa structure typique dans l'ensemble des problèmes qu'il est possible de rencontrer en traitement du signal a souvent été étudiée.
L'étude et l'implémentation parallèle de cet algorithme ont été réalisées par [92, 98] sur de nombreuses machines existantes qu'elles soient massivement parallèles comme la Maspar ou le NASA/GoodyearMPP, ou M.I.M.D. comme l'iPSC/2, le Cray X-MP/48 ou encore le nCube. Leurs conclusions montrent une certaine supériorité du calcul massivement parallèle.
Mais, en général, les analyses ne portent que sur le calcul des coefficients de réflexion. Nous regarderons plus particulièrement les calculs des coefficients autorégressifs qui font appel à des permutations. En effet, ces permutations nécessitent des communications entre les processeurs. Il est donc intéressant de savoir si en ajoutant des communications, les conclusions restent comparables ou au moins du même ordre de grandeur.
C'est pourquoi après un rappel sur les méthodes de prédiction linéaire, nous étudierons dans la section 4 le problème des permutations sur différentes topologies régulières (hypercube, anneau), avant de les appliquer au cas de l'algorithme de Burg. Nous finirons par des expérimentations sur une Connection Machine CM2

et les comparerons avec les résultats obtenus sur MASPAR (et autres machines parallèles) [92, 98].

2. Rappels sur les méthodes de prédiction linéaire

Définition 1 : Un signal S est dit signal autorégressif, si sa sortie au temps t ne dépend que des sorties antérieures et non plus des entrées. Il s'écrit sous la forme :

$$S_t = \sum_{k=1}^{p} a_k S_{t-k}$$

Un tel modèle est appelé modèle "tous pôles" et les coefficients a_k (nombres réels) ainsi définis, sont les coefficients prédicteurs du signal S.

Dans la pratique, on ne retrouve pas exactement le signal de sortie, on cherche à le décrire au mieux. C'est le problème de la prédiction linéaire d'ordre p qui consiste à trouver les coefficients a_k (pour $k = 1, \ldots, p$) qui minimisent l'erreur commise entre la valeur "exacte" du signal de sortie et celle obtenue par le modèle.

Soit e_t est un bruit blanc centré gaussien représentant l'erreur commise dans l'estimation du signal (i.e. un bruit parasite altérant le signal principal).
Si on pose $a_0 = 1$ alors le signal peut s'écrire :

$$e_t = \sum_{k=0}^{p} a_k S_{t-k}$$

Il existe deux classes de méthodes pour résoudre ce type de problème : les méthodes d'autocorrélation qui nécessitent la résolution d'un système linéaire dont les coefficients doivent être calculés au préalable, et les méthodes de type "Treillis" qui utilisent directement le signal de sortie discrétisé à partir de son échantillonnage. Décrivons plus en détails ces méthodes.

2.1. Les méthodes d'autocorrélation.

Si on note :

$$R_i = \sum_{k=0}^{N-1} S_k \, S_{N-i}$$

la résolution du problème de minimisation donne par application de la méthode des moindres carrés, le système linéaire suivant :

$$\begin{pmatrix} R_0 & R_1 & \dots & R_p \\ R_1 & R_0 & \dots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \dots & R_0 \end{pmatrix} \begin{pmatrix} 1 \\ a_1 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} \sigma^2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

avec $\sigma^2 = E(e^2)$ (i.e la variance de e_t) où $E(x)$ est l'espérance mathématique de x. Ce sont les équations de Yule-Walker [15] et la matrice R ainsi créée est une matrice Tœplitz réelle définie positive.

Il existe deux méthodes différentes pour trouver la solution d'un système linéaire de ce type qui en réalité n'en forment qu'une seule. Ces deux méthodes sont la méthode de Levinson et celle de Schur [73]. Toutes deux utilisent des relations de récurrence qui permettent de calculer les coefficients prédicteurs a_k à partir de l'évaluation de nouveaux coefficients notés ρ_k appelés coefficients de réflexion ou encore coefficients de corrélation partielle.
Dans beaucoup d'applications, ces coefficients de réflexion sont un bon substitut aux coefficients de prédiction. L'évaluation de ces coefficients se fait à partir de relations de récurrence obtenues en calculant successivement les sous-systèmes obtenus pour chaque sous-matrice Tœplitz R_k pour $k = 1, \dots, p$.

Dans le cas de la méthode de Levinson la relation de récurrence est de la forme :

$$a_{k,i} = a_{k-1,i} + \rho_k a_{k-1,k-i} \tag{15}$$

où k désigne l'étape de la récurrence et i varie de 1 jusqu'à $k-1$.

Soit le changement de variable :

$$e_{k,j} = \sum_{i=0}^{p} R_{|i-j|} a_{k,i} \; pour j = k+1-n, \dots, n. \tag{16}$$

En l'appliquant á la relation de récurrence de Levinson (15), on obtient alors la relation de base de l'algorithme de Schur.

$$e_{k,j} = e_{k-1,-j} + \rho_k e_{k-1,k-j} \tag{17}$$

où j varie de 0 jusqu'à $n-k-1$.

Les deux méthodes que nous venons de rappeler brièvement sont détaillées dans [39] et ont une complexité équivalente en séquentiel ($\mathcal{O}(n)$ opérations). Au point de vue numérique (i.e. en virgule flottante), leurs solutions peuvent être erronées. En effet, le calcul des coefficients R_i de la matrice R est soumis à des erreurs d'arrondis qui peuvent entraîner une instabilité sur la solution finale [90]. Le

problème de stabilité exige un bon conditionnement de la matrice R du problème global [22]. Il est à noter que dans le cas d'un calcul à virgule fixe, la méthode de Schur reste la méthode la plus souvent employée.

2.2. Les méthodes de type "Treillis".

A l'inverse des deux méthodes précédentes, les méthodes de type "Treillis" calculent directement les paramètres de sortie à partir des entrées du système: les S_i. Dans ce cas, la propagation d'erreur diminue.

On note $f_k(t)$ (respectivement $b_k(t)$) l'erreur de prédiction directe (resp. rétrograde), et on a:

$$f_k(t) = \sum_{i=0}^{k} a_{k,i} S_{t-i} \tag{18}$$

$$b_k(t) = \sum_{i=0}^{k} a_{k,k-i} S_{t-i} \tag{19}$$

Le principe de ces méthodes est de minimiser soit la norme de l'erreur de prédiction directe $f_k(t)$, soit la norme de l'erreur de prédiction rétrograde $b_k(t)$, soit une combinaison des deux. Dans [81], une présentation des différents types de méthodes est faite. Nous étudions seulement le cas de l'algorithme de Burg qui utilise la troisième solution.
En utilisant les relations (15), (18) et (19), et en posant $f_0(n) = b_0(n) = S(n)$, on obtient les relations de récurrence entre les deux erreurs de prédiction qui nous permettent à chaque étape k de calculer le coefficient de réflexion ρ_k :

$$\rho_{k+1} = -2 \frac{\sum_{i=n-k+1}^{n} f_k(i) b_{k-1}(i)}{\sum_{i=n-k+1}^{n} f_k^2(i) + b_{k-1}^2(i)} \tag{20}$$

$$f_{k+1}(n) = f_k(n) + \rho_{k+1} b_k(n-1) \tag{21}$$

$$b_{k+1}(n) = \rho_{k+1} f_k(n) + b_k(n-1) \tag{22}$$

La démarche complète pour calculer l'ensemble des coefficients de réflexion consiste à itérer k fois les trois relations ci-dessus.
La complexité de cet algorithme est alors en $k(10n - 4(k+1))$. Cette complexité est plus élevée que pour les méthodes précédentes au niveau d'une itération mais ne demande pas de calculs supplémentaires pour le calcul des éléments R_i de la matrice. Nous nous proposons d'étudier la parallélisation de cette méthode.

3. Etude de l'algorithme de Burg.

En reprenant les formules (20) à (22), il est facile de construire l'algorithme de Burg pour le problème de prédiction linéaire d'ordre k [14].

Algorithme VI.1 : algorithme séquentiel de Burg

```
Pour i=1... N
    e(i) = S(i); b(i) = S(i)

Pour i=1... k
    Som1=0; Som2=0
    Pour j=1... N
        Som1 = Som1 + e(j) × b(j-i)
        Som2 = Som2 + e(j)² + b(j-i)²

    ρ(i) = -2.0 × Som1/Som2
    Si i > 1 alors
        Pour j=1... i-1
            al(j) = a(j) + ρ(i) × al(i-j)

        Pour j=1...i-1
            a(j) = al(j)

    Pour j=1... N
        temp = e(j) + ρ(i) × b(i-j)
        b(j-i)= b(j-i) + ρ(i) × e(j)
        e(j) = temp x
```

Dans cet algorithme séquentiel, on s'aperçoit qu'il est nécessaire d'utiliser une variable temporaire pour la mise à jour des erreurs de prédiction $e(i)$ et $b(j-i)$. Mais surtout il faut un vecteur complet pour calculer le nouveau vecteur a contenant les coefficients prédicteurs d'ordre i car la contrainte temporelle nous oblige à conserver le début de l'ancien vecteur pour calculer la fin du nouveau.
En fait, ce calcul de a est tout simplement une permutation miroir de taille $i-1$ suivie d'un calcul simple, c'est-à-dire une permutation où l'élément $a[j]$ doit rencontrer l'élément $a[i-j]$ pour obtenir la mise à jour de a (permutation miroir sur $i-1$ éléments).
Une solution vectorielle peut être apportée à ce problème. En effet, il suffit de recopier a dans un deuxième vecteur b, puis de les mettre en entrée de l'unité vectorielle, un dans le sens normal, l'autre "tête bêche". Ce calcul devient alors

une opération usuelle sur les processeurs vectoriels. [71, 107]

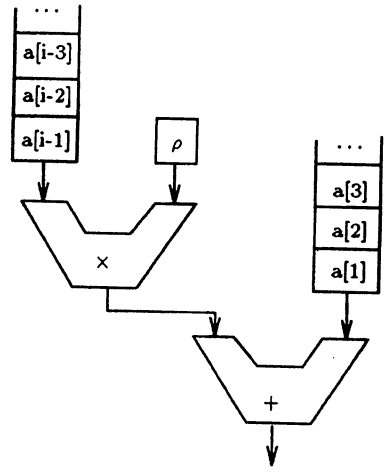


Figure VI.38. solution vectorielle à la permutation

Pour une étude parallèle de cet algorithme, nous allons le décomposer en utilisant les notations BLAS comme dans le chapitre précédent. Les BLAS par leur modularité, permettent une implantation simple à mettre en œuvre [40]. Le corps principal de l'algorithme s'écrit alors sous la forme suivante :

Algorithme VI.2 : Version BLAS

Pour i=1 ... k
  Som1=0 ; Som2=0
  Som1 = DOT du produit de e(i) et b(j-i)
  Som2 = DOT de la somme de $e(i)^2$ et $b(j-i)^2$
  $\rho(i)$ = -2.0 $\times \frac{Som1}{Som2}$
  Permutation de a et stockage dans al
  AXPY pour le calcul de a
  temp = AX pour commencer le calcul de e
  AXPY pour le calcul de b
  XPY = pour finir le calcul de e

L'utilisation d'une variable temporaire est toujours nécessaire à la mise à jour des deux erreurs car il faut séparer un AXPY en deux étapes afin de ne pas perdre les informations contenues initialement dans le vecteur e. Ceci n'a aucune répercussion sur le temps d'exécution si les processeurs n'ont pas d'unité de

calcul enchaînant efficacement les deux opérations, comme par exemple sur une Connection Machine.

Par contre la permutation revient à effectuer une communication générale de type **send** utilisant l'unité matérielle de routage. Elle ne demande donc aucun vecteur supplémentaire [25]. Au même instant tous les éléments devant permuter sont envoyés vers leur processeur destination en ne produisant aucune perte d'information.
La mise à jour de a s'effectue avec le message qui arrive sur le processeur. Le parallélisme permet donc un gain de place par rapport à l'algorithme séquentiel. Il nous faut maintenant étudier le coût supplémentaire qu'entraîne l'utilisation des communications pour effectuer la permutation. Une autre version parallèle de l'algorithme de Burg tenant compte de la tolérance aux erreurs se trouve dans [83].

4. Les permutations

Les permutations jouent un rôle important au niveau des mouvements de données lors de l'étude d'algorithme parallèles. On les retrouve souvent dans bon nombre de schémas parallèles classiques tels que les "shuffles" avec la transformée de Fourier (permutation miroir) [101], la transposition de matrice [67] ou encore la conversion entre divers stockages [80]. Mais aussi dans bien d'autres domaines tels que la résolution d'équations aux dérivées partielles, ou l'évaluation d'expressions arithmétiques en calcul formel.

4.1. Quelques rappels sur les permutations.

Rappelons tout d'abord quelques définitions et propriétés pour mieux comprendre l'intérêt des permutations [4, 23].

Définition 2 : Soit E un ensemble, l'ensemble des bijections de E dans lui-même, s'appelle l'ensemble des permutations de E. on le note $\mathfrak{S}(E)$.

Cet ensemble est non vide car au moins $\mathrm{Id}_E \in$ E.
Soit la loi o, loi de composition interne, telle que $\forall \sigma, \delta \in \mathfrak{S}(E)$, $\sigma \circ \delta \in \mathfrak{S}(E)$
$\mathfrak{S}(E)$ munit de cette loi est un groupe non commutatif.

Définition 3 : On appellera *taille* de la permutation le cardinal de l'ensemble de départ, c'est-à-dire le nombre d'éléments sur lesquels porte la permutation.

Si E est un ensemble fini de cardinal $n \geq 1$, la groupe $\mathfrak{S}(E)$ est un groupe de cardinal $n!$.

Il existe de nombreuses façons de représenter les permutations qui illustrent parfaitement les différents domaines les utilisant. Par exemple, en algèbre, la représentation est la suivante: une matrice $2n$ où la première ligne décrit l'ensemble de départ E, et celle en dessous son image par la permutation σ. Ainsi $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 6 & 4 & 2 & 1 \end{pmatrix}$ représente $\sigma \in \mathfrak{S}(E)$, où E=$\{1,2,3,4,5,6\}$, et $\sigma(1) = 3$, $\sigma(2) = 5$, $\sigma(3) = 6$, $\sigma(4) = 4$, $\sigma(5) = 2$, et $\sigma(6) = 1$.

On utilise aussi souvent une matrice carrée **B** $n \times n$ où n est la taille de la permutation. Elle est définie par:

$$b_{i,j} = \begin{cases} 1 \text{ si } j = \sigma(i), \\ 0 \text{ sinon.} \end{cases}$$

une telle matrice est appelée *matrice de permutation.*
Dans l'exemple ci-dessus, on obtient la matrice suivante:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Une représentation équivalente peut être faite à partir d'une grille $n \times n$ comme le montre la figure VI.39 sur l'exemple de la même permutation.

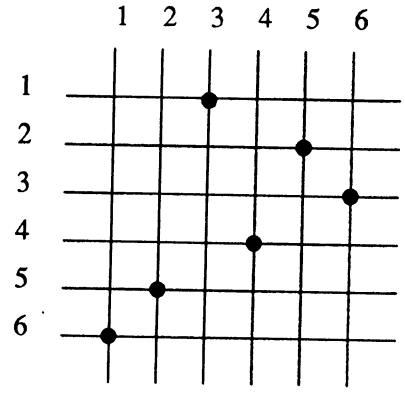


Figure VI.39. représentation de la permutation σ sur une grille $n \times n$.

Remarque 17 : Une relation binaire sur E est associée à une permutation si et seulement si, toutes ses coupes (horizontales et verticales) ont un et un seul élément.

Enfin, il est possible d'associer la permutation σ à un graphe orienté. Le graphe d'une permutation est défini par :

$$\text{Il existe un arc } \vec{xy} \text{ si, et seulement si } \begin{cases} y = \sigma(x) \\ y \neq x \end{cases}$$

la figure VI.40 reprend l'exemple.

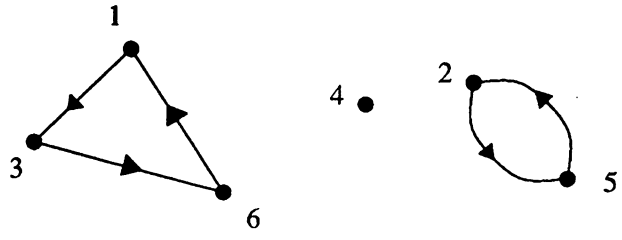


Figure VI.40. Représentation d'une permutation par un graphe orienté.

Dans l'algorithme de Burg, l'intérêt des permutations réside dans la mise à jour du vecteur des coefficients prédicteurs a_k. Mais, il utilise seulement des sous-permutations (i.e. des permutations ne faisant intervenir qu'un sous ensemble de l'ensemble de départ) fortement liées d'une itération à l'autre.
Dans la pratique, en séquentiel, cette mise à jour est effectuée par une simple inversion suivie d'une duplication du vecteur des a_k pour ensuite finir par une opération de type AXPY entre deux vecteurs, opération facilement vectorisable. Nous allons dans cette section montrer l'apport du parallélisme pour la réalisation des permutations et plus particulièrement pour l'algorithme de Burg.

4.2. Parallélisme et permutations

Lorsqu'on connaît à l'avance, une partie ou la totalité des communications qui interviennent lors de l'exécution d'un programme, il est avantageux de précalculer une solution au routage. Dans ce cas, on parle de communication "*off-line*" [77].
Pour une permutation complète, le schéma de communication peut devenir complexe. Si le schéma est utilisé plusieurs fois, alors il n'est pas aberrant de chercher une solution optimale, même si cela devient coûteux. Le "Communication Compiler" de Dahl [33] est un exemple de solutions à apporter.

Un grand débat existe sur l'influence de la topologie sur la réalisation d'une permutation : anneau [77], grille [72], hypercube [67, 100, 103], réseau de benes̃ [84], etc....
Nous étudierons plus particulièrement le cas des deux topologies qui sont à la base des architectures actuelles : l'anneau et l'hypercube.
Il est évident que, si on désire exécuter une permutation sur ces deux topologies l'hypercube semble le plus efficace.

4.2.1. L'hypercube

Conjecture 1 (Szymanski) : Dans un hypercube de dimension supérieur à 3, il est toujours possible d'effectuer n'importe quelle permutation en utilisant les plus courts chemins (distance de Hamming). C'est-à-dire trouver un chemin orienté de plus courte distance entre l'élément et son image par la permutation, et surtout à arêtes disjointes avec tous les autres. [103]

Ceci est équivalent à : toute permutation peut être effectuer sur un hypercube en au plus $\log_2(n)$ étapes de communication.

La conjecture précédente a été mise en défaut sur certains contre-exemples [45]. Beaucoup de travaux ont été effectués pour caractériser les permutations réalisables en au plus $\log_2(n)$ étapes [100]. Par exemple : Ho nous propose un algorithme récursif pour effectuer en $\log_2(n)$ étapes une transposition de matrice sur un hypercube avec un placement des données en grille [67]. Liu et You ont démontré que les permutations de Lee (problème d'allocation mémoire d'une matrice $n \times n$ sur une machine à n processeurs) peuvent être réalisées sur une machine parallèle de type hypercube à mémoire distribuée en $N = \log_2(n)$ étapes [80]. Une présentation de tous ses travaux se trouvent dans [38].

Nous étudierons plus en détail des permutations particulières bien plus simple : la permutation miroir du "perfect shuffle" et ses sous permutations. Si on place les données sur un anneau plongé dans l'hypercube en utilisant les codes de Gray réfléchis, alors la permutation miroir (i.e. la permutation de $a[i]$ avec $a[N-i]$) est effectuée en un seul "top" de communication. Par exemple, sur la figure VI.41, la communication se fera sur la troisième dimension de l'hypercube. En général, la communication s'effectue toujours sur la dernière dimension créée par la définition récursive de l'hypercube.

Ce placement des données sera utilisé tout au long de cette section.

Dans le cas de l'algorithme de Burg, les permutations sont des sous-permutations de tailles très inférieures à la permutation complète, et sont du type miroir. Nous sommes obligés de les considérer une à une car elles sont différentes à chaque étape. Néanmoins, elles sont fortement liées par le fait qu'il suffit de rajouter un

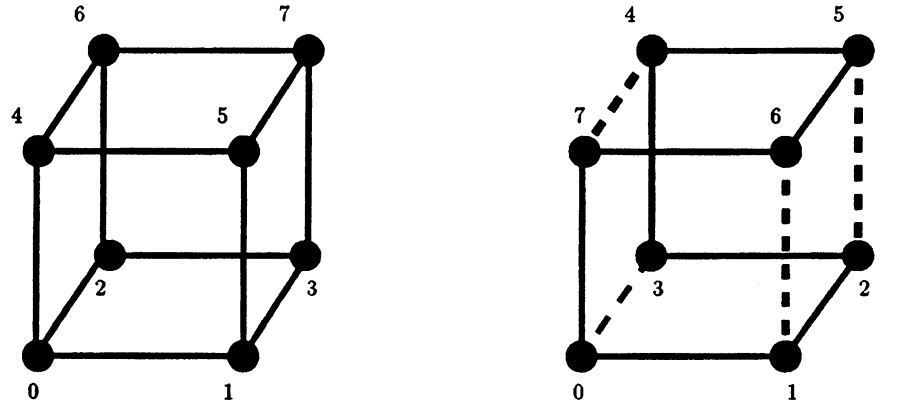


Figure VI.41. comparaison entre les placements de données

élément à chaque itération. C'est pourquoi, en utilisant la propriété de récursivité du plongement d'un anneau dans un hypercube, on déduit la propriété suivante :

Propriété 1 :

- Si la permutation est de taille $\log_2(n)$ alors elle se réalisera sur un sous-cube en une seule étape de communication. En effet, le placement des données est tel que l'on se place dans le cas d'une permutation miroir de taille $\log_2(n)$.

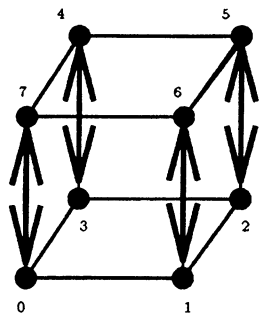


Figure VI.42. permutation miroir sur un sous-cube de dimension 3

- Sinon elle se réalisera en au plus $\lceil \log_2(n) \rceil$ étapes de communication où n représente la taille de la permutation. En effet, il existe pour tous les éléments de la permutation, un chemin pour envoyer sa valeur au processeur destination. Ces chemins sont distincts dans le temps et en au plus $\log_2(n)$.

- Distincts par construction car ces chemins sont construits en suivant l'anneau plongé dans un hypercube,
- et en au plus $\log_2(n)$ par la construction récursive de l'anneau qui permet de supprimer deux directions identiques dans la séquence des directions à suivre pour former le chemin. On aura dans la séquence finale au plus une apparition de chaque direction.

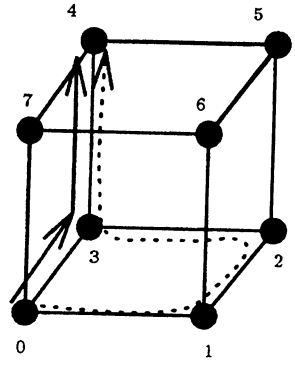


Figure VI.43. simplification d'un chemin

Par exemple dans la figure VI.43, on remarque que la séquence de directions formant le chemin reliant le processeur 0 et le processeur 4 est 1213. On peut la remplacée par 23.
Cette propriété de simplification est vrai pour tous les chemins à faire lors de la permutation. De plus, il semble que grâce à la propriété du plongement de l'anneau dans l'hypercube tous ces chemins soient à arêtes disjointes dans le temps. En fait, on utilise les liens laissés libre par le plongement.

4.2.2. L'anneau

L'anneau de base ne se prête pas bien aux permutations car une permutation simple, pour échanger deux valeurs, devient d'autant plus coûteuse que la distance entre les deux éléments augmente. De plus des phénomènes de rétentions de messages dû au faible degré (2) d'un anneau se produisent empêchant ainsi d'effectuer une permutation générale en un nombre d'étapes égale à la taille du diamètre.

Théorème 3 : Sur une ligne de n processeurs, toute permutation de taille nE peut être effectuer en n étapes avec au plus un message par processeur à chaque instant.

La démonstration est basée sur le fait qu'un tri par transpositions paires-impaires réalise n'importe quelle permutation (cf [77]).
Par exemple, pour la permutation donnée en 4.1, sur la figure VI.44 on remarque que la borne supérieure est atteinte en 6 étapes.

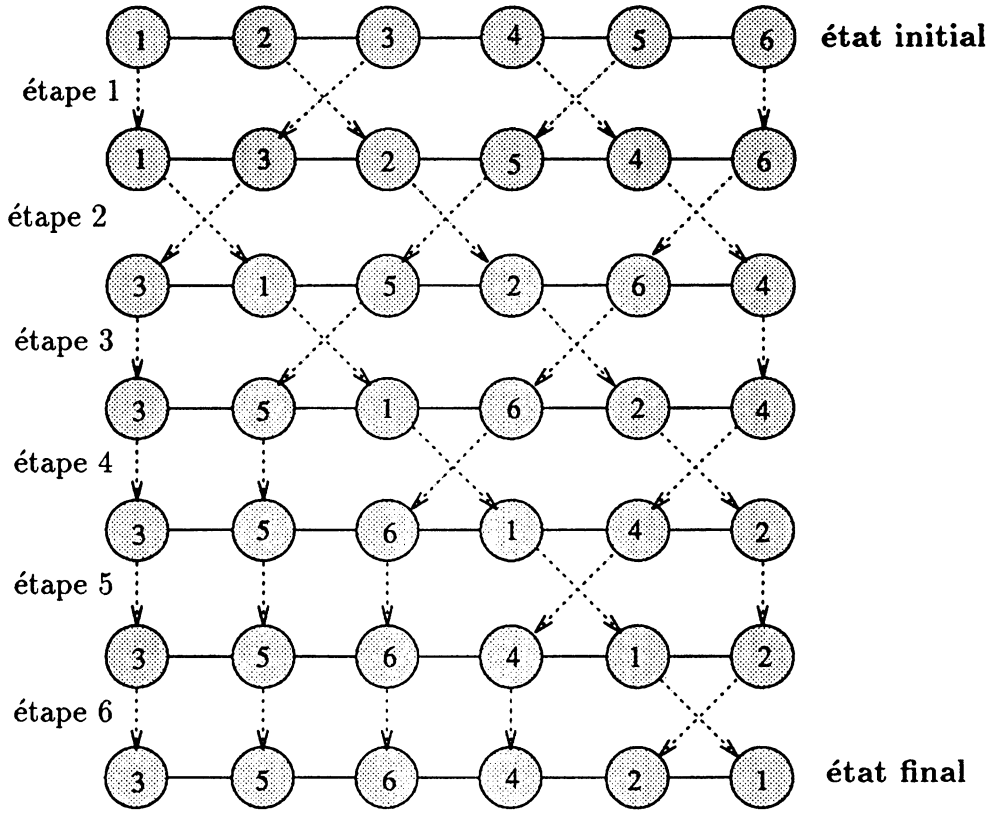


Figure VI.44. Décomposition de la permutation σ en 6 étapes.

Mais pour l'algorithme de Burg il faut tenir compte de sa structure bien particulière et surtout de la faible taille des permutations. Comme nous l'avons dit précédemment, elles sont liées de façon itérative. On se rend très vite compte que l'on peut utiliser cette liaison pour diminuer le coût global des communications. En effet, le coefficient $a[1]$ doit rencontrer successivement dans l'ordre croissant, tous les autres éléments d'indice supérieur jusqu'à l'ordre k. Pour cela, il faut à chaque étape le faire progresser d'un élément vers la droite. (voir figures VI.45 et VI.46)

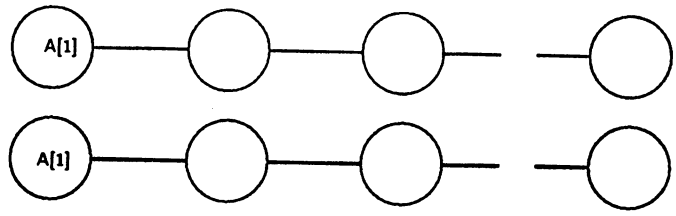


Figure VI.45. première étape

Il faut faire de même avec le deuxième élément $a[2]$ avec un décalage d'une étape, et ainsi de suite ...

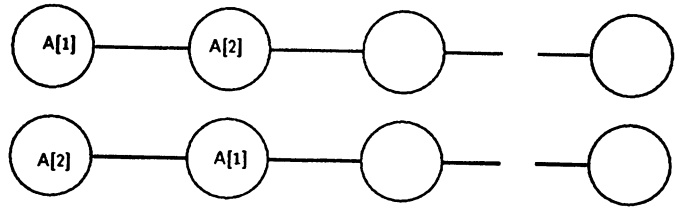


Figure VI.46. deuxième étape

En conclusion, il suffit de créer un deuxième anneau contenant les mêmes données qui va remonter en sens inverse par rapport au premier anneau. Ainsi à chaque étape on peut, sur tous les processeurs formant l'anneau, calculer la mise à jour des coefficients de prédiction.

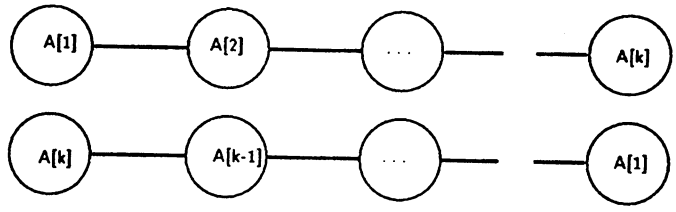


Figure VI.47. dernière étape

Pour pouvoir itérer ce principe, il faut décaler à chaque étape le deuxième anneau d'un cran vers les indices croissants. Mais pour que le processeur voisin puisse à son tour, au cours de l'étape suivante, calculer la mise à jour de a, il est nécessaire d'effectuer un double calcul sur chaque processeur avant le décalage.

L'algorithme demande une duplication des données du vecteur a mais qui reste faible lorsque l'on sait que le nombre de coefficients recherchés est faible par rapport au nombre de données initiales. Le fait le plus important reste néanmoins

le faible coût en communications car leur nombre est considérablement réduit par rapport à la topologie en hypercube. Le coût global des communication est seulement de (k-1) communications entre deux voisins.

Reste le problème de la mise à jour des deux anneaux qui entraîne un double calcul. Ce double calcul coûte-t-il plus cher que le nombre de communications supplémentaires engendrées par le premier algorithme? C'est à cette question que nous répondons par les expérimentations sur une Connection Machine dans la section 5.

5. Résultats expérimentaux

5.1. Programmation de l'algorithme de Burg

La programmation du corps principal de l'algorithme, excepté le problème des permutations, est inspiré des travaux de [92, 98]. En effet, sur une machine massivement parallèle, il n'existe pas d'autres alternatives qui permettent une programmation efficace. Nous utilisons le placement des données de la figure VI.41 qui permet un maximum de calculs internes aux processeurs et limite le nombre de communications à un décalage du vecteur des erreurs de prédiction rétrograde b. On peut ainsi calculer les sommes $Som1$ et $Som2$ en $log_2(n)$ en utilisant les communications avec recombinaison sur Connection Machine.

Dans un premier temps, nous nous sommes intéressés à la comparaison des deux algorithmes de permutation. La programmation de l'ensemble des algorithmes a été réalisée en CParis afin de s'adapter au mieux à la Connection Machine. Un regret subsiste : il nous est impossible dans ce langage de décider du chemin suivi par les messages lors d'une communication de type **"send"**. En effet, elle utilise l'unité matérielle de routage. Nous ne sommes donc pas sûrs de suivre le schéma optimal sur un hypercube.

Remarque 18 : L'emploi d'un outil tel que le "communication compiler" [33] est inutile dans le cas de l'algorithme de Burg. Le schéma évoluant à chaque itération, le coût de recherche d'une solution optimale serait prohibitif.

P	8192		16384		65536		262144	
m	send	news	send	news	send	news	send	news
50	0.21	0.18	0.25	0.23	0.61	0.48	2.0	1.53
100	0.44	0.37	0.56	0.45	1.29	0.97	4.12	3.09
250	1.15	0.92	1.52	1.13				
500	2.33	1.84	3.15	2.26	10.35	4.84	22.92	15.31
750	3.52	2.75	4.79	3.4				
1000	4.71	3.67	6.49	4.53	23.27	9.69	58.82	30.63
5000	24.04	18.44	33.97	22.71	130.60	48.56	439.57	152.87

Tableau VI.11. Mesures en seconde en fonction du nombre de coefficients à trouver et le nombre de processeurs utilisés.

On note **send** les performances de l'algorithmes pour une permutation utilisant le router, **news** celles utilisant un deuxième anneau, **m** l'ordre de la prédiction linéaire et **P** le nombre de processeurs. Les résultats obtenus sont les suivants :

On remarque que dans tous les cas, les performances sont meilleures pour le deuxième algorithme. Le surcoût de calcul entraîné par le double calcul ne compense pas la différence trop importante entre les deux types de communication sur une Connection Machine.
Il faut donc lorsqu'on étudie un algorithme massivement parallèle ne pas exclure un surcoût en calcul pour un changement de type de communication.

De plus, les expérimentations ont été effectuées pour des ordres de prédiction linéaire nettement supérieurs à ceux couramment employés qui sont de l'ordre de 100 à 250. Cela a été réalisé dans le but de comparer les deux méthodes en fonction du nombre de processeurs.
On observe une croissance linéaire du temps d'exécution en fonction de l'ordre de prédiction linéaire. Ce phénomène est normal pour le deuxième algorithme car chaque itération coûte le même prix. Il l'est moins pour le premier car le nombre de communications dépend du numéro de l'itération. Les figures VI.48 et VI.49 nous permettent d'étudier le rapport **send / news** en fonction du VP-ratio.

On observe que le gain tend asymptotiquement vers une limite finie lorsque l'ordre de prédiction croît pour un nombre fixé de processeurs. Dans le cas contraire, c'est-à-dire à ordre de prédiction fixé et nombre de processeurs variable, on remarque qu'il existe un VP-ratio pour lequel le rapport entre les deux méthodes est le meilleur.

De plus, il semble que si le VP-ratio est trop élevé, alors le rapport tende vers une limite finie. Ce phénomène est dû au stockage des données de la Connection

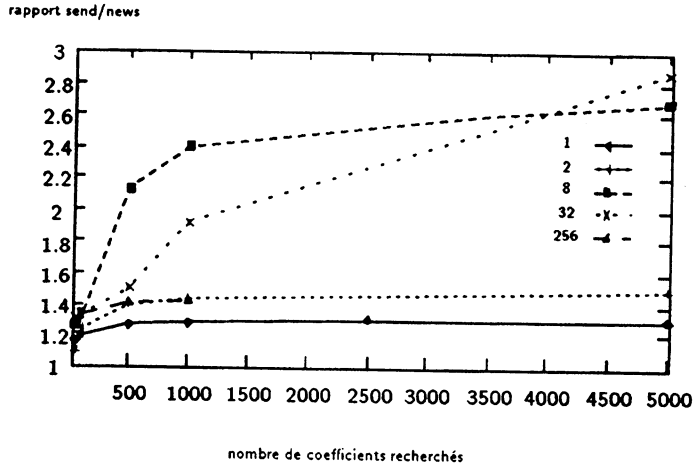


Figure VI.48. Comparaison des deux types de permutations en fonction du nombre de processeurs et le nombres de coefficients recherchés.

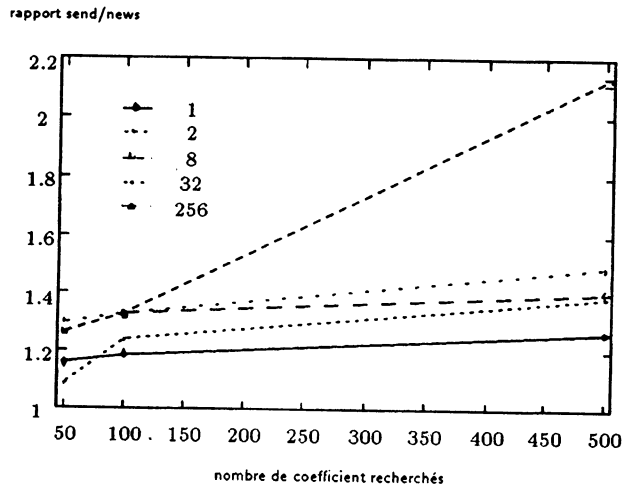


Figure VI.49. Détail sur la zone (50-500)

Machine qui va placer sur un même processeur physique les processeurs virtuels voisins.
Par exemple dans le cas d'un VP-ratio de 256, le processeur 1 va contenir les 256 premiers processeurs virtuels, et ainsi les 1000 premiers coefficients de réflexion seront stockés sur les 4 premiers processeurs physiques diminuant ainsi le coût en communication. Pour 100 coefficients et un VP-ratio de 256, les coefficients étant tous sur un seul processeur, il n'existe plus de communications, elles sont remplacés par des échanges internes à la mémoire.

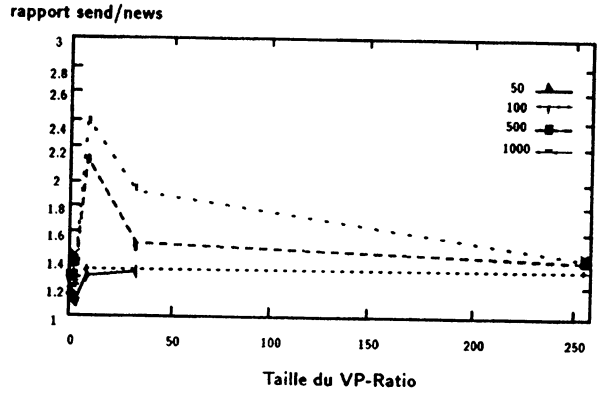


Figure VI.50. Effets du Vp-ratio à taille fixée

	8K - 32bits		4K - 64bits	
m	send	news	send	news
50	0.21	0.18	0.42	0.39
100	0.44	0.37	0.92	0.77
250	1.15	0.92	2.5	1.92
500	2.33	1.84	5.15	3.83
1000	4.71	3.67	10.50	7.66
2500	11.95	9.21	54.11	38.27
8192	39.53	30.36	89.12	62.73

Tableau VI.12. Comparaison en fonction de la taille des flottants.

De plus, ayant accès à une Connection Machine travaillant avec des processeurs flottants double précision, il est intéressant de comparer les temps obtenus pour un nombre de processeurs identiques, mais avec VP-ratio différents.

On remarque que le rapport entre les deux machines est identique à celui des VP-ratio. Donc la taille des processeurs n'influence pas le gain. Une étude complémentaire de la stabilité de la solution montre un avantage certain pour la Connection Machine à processeurs flottants double précision, mais là n'est pas notre but.

6. Conclusion

La méthode classique devrait être meilleure en théorie car elle demande moins de calculs, mais dans la pratique, le rapport entre cette dernière et la méthode à base de communication entre voisins reste supérieur à 1. Donc on se rend compte que l'appel aux communications générales de la Connection machine est plus coûteux que celui entre voisins.

Par une étude plus approfondie de l'algorithme de Burg, on remarque que le calcul de $\rho(i)$ n'utilise pas l'ensemble des vecteurs e et b lors de la réalisation des produits scalaires. Leur mise à jour ne s'effectuent que sur la partie finale de notre anneau. Or le calcul des coefficients prédicteurs est le même que celui de la mise à jour de e. Il est alors possible de les réaliser en même temps sur des processeurs différents, en plaçant le vecteur a sur les processeurs libérés à chaque itération.
La seule précaution à prendre est de faire uniquement le calcul des sommes partielles seulement sur les processeurs contenant e et b. Ceci est possible sur une Connection Machine en utilisant le "Scan segmenté" (Cf A.2.4.1) qui permet d'effectuer ces sommes partielles en utilisant les communications avec recombinaison sur une partie des processeurs.

Ainsi, il est possible de diminuer sensiblement le temps total de l'algorithme de Burg car les deux décalages de l'anneau n'en font plus qu'un comme les deux mises à jour.
Dans la pratique, ce nouveau placement des données permet de regrouper deux boucles de l'algorithme séquentiel. Mais son avantage primordial est de pouvoir calculer dans le même temps que celui proposé par [92, 98] à la fois les coefficients prédicteurs et ceux de réflexion.

Chapitre VII

Conclusion

Nous nous sommes intéressés tout au long de cette thèse à tester le mode S.I.M.D. pour la résolution de problèmes d'algèbre linéaire au travers de la Connection Machine 2 de T.M.C. Pour ce faire, nous avons utilisé les trois approches décrites dans [47].
La *première*, consistant à chercher et à atteindre la puissance maximale de crête, s'est avérée être la plus passionnante, mais aussi la plus difficile. Pour parvenir à ces performances importantes, il a fallu descendre au plus bas niveau de l'architecture. Ce n'est pas une caractéristique de la Connection Machine ou du mode S.I.M.D., on retrouve ce phénomène sur l'ensemble des machines et quel que soit leur mode de programmation (M.I.M.D., S.I.M.D., S.P.M.D.). L'apport principal de cette approche est avant tout une connaissance très fine de la Connection Machine et du mode synchrone. Elle nous a permis de développer des procédures de communications optimales en mode synchrone [37], qui le reste en mode asynchrone [53]. Pour ce faire, nous avons, dans [37], créé un nouvel algorithme de construction d'arbres équilibrés. Associées à un placement judicieux des données utilisant la décomposition en produit cartésien d'un hypercube de dimension n en deux hypercubes de dimension $\frac{n}{2}$, ces procédures ont permis d'atteindre des performances supérieures au Gigaflops, c'est-à-dire l'exécution de plus d'un milliard d'opérations par seconde sur une Connection Machine 32k-processeurs.

La *deuxième*, qui consiste à utiliser les boites noires transparentes pour un utilisateur, nous a permis, à partir de la décomposition en instructions de faible volume (i.e en $\mathcal{O}(n)$, $\mathcal{O}(n^2)$), d'adapter au mieux les différents algorithmes en distinguant les instructions qui peuvent s'exécuter facilement en parallèle. De plus, elle nous a permis de mettre en évidence la barrière de synchronisation que sont les deux produits scalaires lors d'une itération du Gradient Conjugué. Ce

phénomène entraîne une chute importante des performances. Nous avons donc proposé au chapitre V une méthode à pas constants qui dans le cas de matrices creuses structurées telles que la matrice du Laplacien, est meilleure que celle du Gradient Conjugué [34].

En ce qui concerne la *troisième*, c'est-à-dire la prise en compte de la structure des matrices à traiter, le chapitre V fournit des exemples des avantages que l'on peut espérer obtenir lorsqu'on utilise au mieux la structure des matrices. Il est intéressant de tenir compte de cette structure pour, à la fois, gagner de la place mémoire, augmenter la taille des problèmes à résoudre et surtout accroître les performances [34]. Dans [57], on trouve d'autres exemples de matrices structurées qui confirment nos résultats.

Une première conclusion s'impose. Les trois approches ne sont pas antagonistes ; bien au contraire si on désire aller le plus vite possible. Il faut tenir compte à la fois des spécificités de la machine et de celles du problème pour programmer de façon efficace. Malheureusement le mode S.I.M.D. massif n'est peut-être pas toujours le meilleur moyen d'y arriver.
En effet, la première leçon à retenir des différentes expérimentations effectuées tout au long de cette thèse est que la *règle d'or* du parallélisme à grains très fins : « *un élément de la matrice par processeur* », ne doit pas toujours forcément être appliquée. Le chapitre IV avec la résolution triangulaire est un bon exemple de ce qu'il ne faut pas faire car comme nous l'avons montré ce type de placement entraîne une faible efficacité (un nombre trop faible de processeurs peut travailler ensemble). Comme autre contre-exemple, nous pouvons citer les opérations de type *Parallel Prefix* sur hypercube comme le scan ou le spread (cf A. 2.4.1). En effet dans ce même chapitre, nous avons vu que si le VP-ratio est grand, c'est-à-dire si le nombre d'éléments physiques sur un processeur est grand, alors le coût d'un spread devient linéaire. Cela s'explique par le fait que le coût global diminue proportionnellement avec la taille. Ce phénomène est dû à la diminution du coût de communication, et à moindre importance à l'accélération des calculs locaux avec un processeur vectoriel. Dans ce cas, ce n'est pas la partie linéaire qui fait chuter les performances mais bien la partie logarithmique due aux communications. De façon plus générale, ce n'est pas parce qu'une partie d'un algorithme est en $\mathcal{O}(n^2)$ et une autre en $\mathcal{O}(n^3)$, que l'on doit nécessairement la sous-estimer lors de la programmation.

La chute du mythe de la *règle d'or* met en évidence un autre problème : celui du placement des données. Au début, il semblait naturel d'associer une matrice à un tableau à deux dimensions, d'autant plus que cela permettait une visualisation évidente des effets sur une feuille de papier. Mais la taille des problèmes augmentant, il devenait difficile de construire des machines avec un milliard de processeurs. T.M.C. a, avec le concept de virtualisation (VP-ratio), apporté un

semblant de solution. Celle-ci a le mérite d'être transparente pour un utilisateur, et d'être valable en général comme nous l'avons vu dans les chapitres précédents. Mais il y a toujours des contre-exemples comme dans le chapitre IV où les performances chutent lorsque le VP-ratio augmente, car le caractère séquentiel de la résolution triangulaire ne peut pas tirer partie de la vectorisation au niveau du processeur.
Lors de la programmation en CMIS, ce concept de virtualisation est inexistant. Nous avons dû adapter le placement à l'algorithme et non le contraire comme l'impose souvent la règle d'or. Dans le chapitre II, nous avons présenté un placement optimal où la position des éléments de la matrice dépend uniquement de celle initiale des différentes composantes du vecteur. De plus, si le vecteur est placé de façon régulière, on obtient alors un stockage de type "bloc".
Ce stockage fait appel au nouveau concept du parallélisme: *les indices sont un ensemble, plutôt qu'une suite ordonnée*, c'est-à-dire il ne faut pas figer les choses [63].

En conclusion, pour que le mode synchrone soit efficace, il est plus important d'avoir des données régulières qu'un trop grand nombre de données. La règle d'or numéro 2 s'écrit: « *Placer plusieurs éléments de votre matrice par processeur, mais pas n'importe comment. Il faut suivre le concept précédent* ».
C'est cette règle que nous avons suivie sans dans le chapitre II [37].

Il ne faut pas conclure trop vite à la disparition du mode S.I.M.D. massivement parallèle. Il existera toujours des applications qui seront optimales en mode synchrone: traitement d'images, ou des problèmes d'algèbre linéaire avec des matrices dont la structure s'adapte au mieux avec un réseau entre voisins plus développé. La dernière version de la MasPar avec ses performances accrues, a encore de beaux jours devant elle. Cependant d'après les remarques précédentes, l'expérience acquise et l'architecture des machines actuelles, il est évident que la voie à suivre est celle des machines ayant comme nœuds des processeurs très puissants, possédant si possible une unité vectorielle pour pouvoir accélérer les calculs locaux, et rendre le stockage par "blocs" encore plus efficace. Il faut de plus un nombre suffisamment important de nœuds pour avoir une bonne efficacité et ne pas trop privilégier le mode vectoriel. En outre, compte tenu des restrictions que nous avons apportées sur le mode S.I.M.D., il est avantageux, comme le font certains constructeurs, de se tourner vers un mode de programmation plutôt S.P.M.D./M.I.M.D. qui permet d'exécuter des parties de code différentes sur des processeurs voisins. T.M.C. avec la création de la Connection Machine de la nouvelle génération CM5 fournit en plus des procédures de synchronisation qui permettent théoriquement de passer un code de CM2 sur leur nouvelle machine. Dans la pratique, on est confronté à une perte très sensible des performances.

Dans le cadre de l'algèbre linéaire, une autre solution se profile à l'horizon avec

les réseaux hétérogènes de machines associés à un environnement de programmation. Ils permettent de faire travailler ensemble des machines séquentielles (Sparc station, IBM RS6000, etc..) et des machines parallèles (CM5, iPSC/2, Cray, etc..). Les résultats obtenus avec PVM [1] [42] sont de bonne augure pour l'avenir [20, 21].

[1] Parallel Virtual Machine

Annexe A

La Connection Machine

Le parallélisme est une notion que l'on retrouve un peu partout en informatique. Elle consiste à effectuer plusieurs traitements indépendants simultanément. C'est une méthode très efficace dès que l'on manipule des données régulières, comme des vecteurs ou des matrices. Le parallélisme peut prendre beaucoup de formes différentes, le plus simple est sans aucun doute de travailler de manière synchrone en effectuant la même instruction sur des données multiples. On parle ici de parallélisme de type S.I.M.D. (de l'anglais Single Instruction Multiple Data). C'est le cas de la Connection Machine. Cette contrainte permet d'obtenir un parallélisme massif.
Un programme pour une telle machine est un programme séquentiel auquel sont ajoutées des instructions portant sur un ensemble de données et qui seront exécutées par la machine parallèle. Parmi ces instructions parallèles, on retrouve l'équivalent de toutes les instructions séquentielles portant sur tout un ensemble d'opérandes.
Il n'existe que deux nouveautés principales : la possibilité de gérer des ensembles de données et la présence d'instructions de communications entre les processeurs élémentaires de la machine parallèle.

Le principe de programmation de la Connection Machine est d'associer à chaque élément d'un ensemble, une unité de contrôle et de calcul que l'on appellera (un peu abusivement) un processeur. Tous les éléments de cet ensemble peuvent être alors traités simultanément. Dans le cas d'une image, chaque pixel sera associé à un processeur. Si l'ensemble est plus grand que le nombre de processeurs, ces derniers vont devoir partager leur temps et leur espace mémoire de manière à gérer plusieurs éléments. Ce mécanisme est totalement transparent lors de la programmation pour un utilisateur usuel qui n'a qu'à spécifier le nombre de pro-

cesseurs qu'il désire. On parle alors de processeurs virtuels (ou VP). Le nombre de processeurs virtuels divisé par le nombre de processeurs physiques s'appelle le VP-ratio[1].
Les processeurs de la Connection Machine ont un fonctionnement S.I.M.D., c'est-à-dire qu'ils effectuent tous une même instruction sur des opérandes identiques. Tous les éléments gérés par un ensemble de processeurs subissent donc le même traitement. Ainsi, la mise à jour des positions d'un ensemble de particules correspond à une seule instruction $x \leftarrow x + dx$.
La seule manière pour un processeur de se distinguer par rapport aux autres est de ne pas exécuter l'instruction commune. Ceci est contrôlé par un bit (context-flag) sur chaque processeur virtuel. Si le context-flag d'un processeur est à 1, celui-ci est actif et il exécute les instructions, dans le cas contraire, il n'exécute rien (sauf pour quelques instructions particulières).
Chaque processeur possède une mémoire qui lui est propre et n'a pas directement accès aux mémoires des autres processeurs. Il est donc nécessaire d'avoir un système de communication permettant à deux processeurs d'échanger des informations.

Après une présentation générale et fonctionnelle des différents composants fondamentaux de la machine, la section 2 introduit les différents types de communications en insistant davantage sur les facilités offertes que sur la mise en œuvre pratique, et décrit un outil d'optimisation développé par Dahl [33]. La section 3 présente trois langages de programmation : *Fortran* qui peut être utilisé sans différence fondamentale par rapport à la programmation séquentielle, *ParIS* dont l'intérêt provient du fait qu'il est très proche de la machine et *StarLisp* qui est un bon compromis entre facilité de programmation et utilisation optimale de la machine. Enfin, la dernière section détaille l'assembleur de la Connection Machine CMIS qui nous a permis d'obtenir des performances supérieures au GigaFlop[2].

Remarque 19 : Ce chapitre ne tient pas compte des dernières améliorations de T.M.C. comme par exemple l'environnement *Prism* [31], environnement sous X qui fournit à l'utilisateur des outils de développement qui facilite la programmation et surtout la mise au point avec une fenêtre pour debugger ou encore les bibliothèques de procédures graphiques qui permettent une visualisation temps réel. De même, l'évolution du CM-Fortran avec le nouveau compilateur slice-wise n'est que rapidement abordé lors de l'étude de la nouvelle vision de la Connection Machine (Cf 4.3). Quant à *C**, il a été volontairement écarté. Son développement tardif, nous a empêché de l'utiliser dès le début, puis l'habitude a fini de l'écarter définitivement. Néanmoins toutes ces améliorations ne modifient en rien les caractéristiques et les orientations de la Connection Machine.

[1] Virtual Processors ratio
[2] Milliard d'opérations par seconde

1. Présentation générale

1.1. Description rapide

Dans sa configuration maximale, la Connection Machine comporte 2^{16} processeurs 1-bit, répartis en 8 cubes physiques. La technologie est sûre mais pas très performante, le temps de cycle des petits processeurs élémentaires est assez élevé (contrairement aux ordinateurs de type CRAY). Les bonnes performances sont obtenues grâce à la grande régularité des instructions et au nombre important de données. On y accède par un ordinateur frontal (ou hôte), VAX, SUN ou Symbolics. Le lien entre le frontal et la CM2 est assurée par un séquenceur parmi les quatre disponibles. L'accès s'effectue en mode mono-utilisateur (un utilisateur unique pourra travailler sur un sous-cube complet de 8, 16, 32 ou 64 Kprocesseurs).

Pratiquement, on développe (et compile éventuellement) le programme sur l'hôte. Celui-ci comporte une partie de code séquentiel, exécuté sur le frontal, et une partie de code parallèle. Ce dernier arrive dans chacun des processeurs élémentaires de la Connection Machine à travers un séquenceur. Le mode S.I.M.D. impose partout la même instruction et que les données manipulées soient à la même place en mémoire locale. Une horloge globale garantit l'ordre d'exécution des instructions.

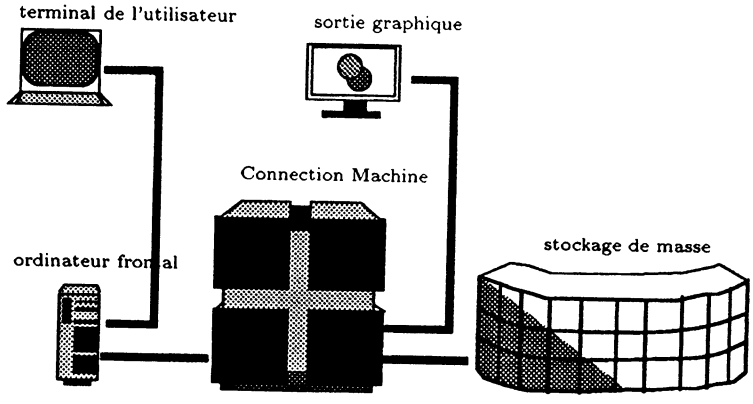


Figure A.51. Schéma global de la CM2

1.2. Architecture fonctionnelle

La Connection Machine est constituée de cartes identiques, regroupant 16 modules de 32 processeurs élémentaires 1-bit. Un tel module est détaillé dans la figure suivante. Il contient deux circuits constitués de 16 processeurs élémen-

taires plus une unité de communication, auxquels s'ajoutent une mémoire locale et une unité de calcul flottant.

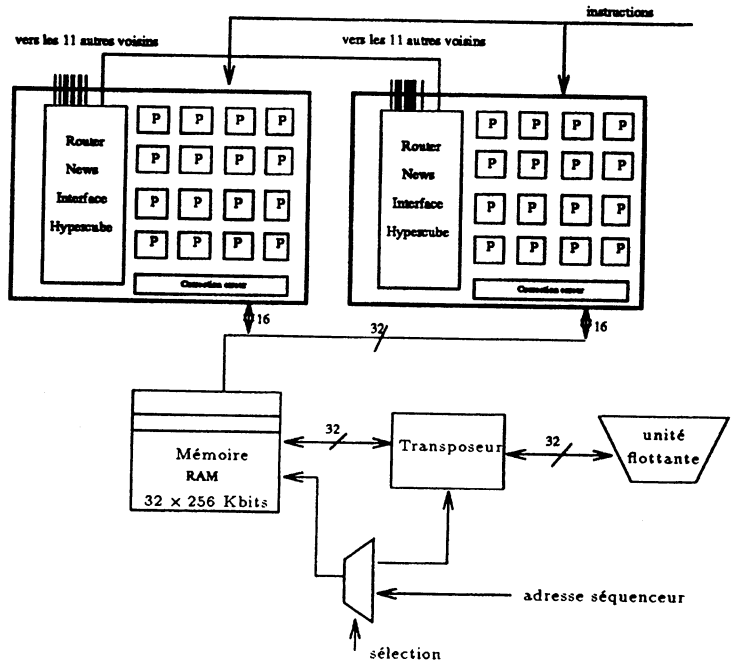


Figure A.52. Organisation d'un module.

Tous les modules de 32 processeurs sont reliés au frontal à travers un séquenceur par un bus d'instructions et un bus d'adresses. Les circuits de 16 processeurs 1-bit sont reliés entre eux, via les unités de communication, suivant un hypercube de degré 12 dans le cas de la configuration maximale. Les seize processeurs élémentaires d'un même circuit sont entièrement connectés entre eux par le biais d'un réseau d'interconnexion complet (butterfly à 4 niveaux). Etudions maintenant en détail les divers composants d'un module.

1.3. Organisation de la mémoire

Vu depuis les langages de hauts niveaux (Fortran, C*, *Lisp et ParIS), la mémoire de chaque processeur se divise en deux parties distinctes : la pile (stack) et le tas (heap). L'accès en lecture et écriture est identique pour les deux parties de la mémoire. Elles se distinguent par leur gestion des espaces libres. Le tas réserve une place où il en trouve, et ne désalloue que la zone contenant la variable. Il peut donc exister des emplacements vides au sein du tas. Au contraire dans la pile, l'allocation mémoire se fait sur le sommet de celle-ci et si l'on désalloue une

variable tout ce qui se trouve au dessus est perdu, de manière à ne pas créer d'emplacement libre.

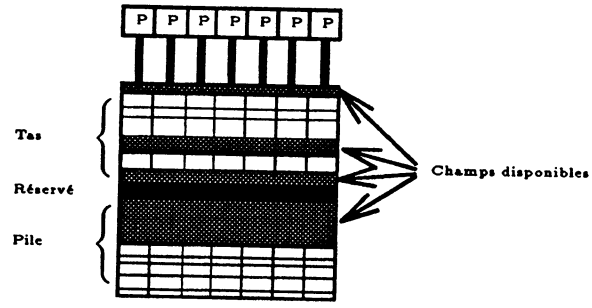


Figure A.53. Organisation d'un module de mémoire

Remarque 20 : Usuellement, les variables générales sont à mettre dans le tas, alors que les variables temporaires sont à stocker dans la pile afin de récupérer la place à la fin de leur utilisation. Il est en effet plus rapide d'allouer un emplacement dans la pile, mais la récupération de cet espace mémoire demande davantage de précautions. C'est donc à l'utilisateur de juger selon ses besoins!

1.4. Le transposeur et l'unité flottante

Afin d'accélérer les calculs numériques, T.M.C. a associé à chaque groupe de 32 processeurs une unité de calcul flottant. Cette unité travaille sous le format standard IEEE (23 bits de mantisse, 8 d'exposant et le bit de signe). Sur certaines machines, cette unité travaille en double précision (64 bits).
Au lieu d'être faite bit à bit et en parallèle sur chacun des 32 processeurs, une opération portant sur des réels (codés en simple précision) est faite séquentiellement par mot de 32 bits. On remarque que l'unité de calcul flottant a besoin de 32 bits provenant d'un même processeur élémentaire alors que la mémoire ne donne qu'un bit de chacun des processeurs à chaque appel. D'où la nécessité d'un circuit de transposition.

Ce transposeur est en fait une matrice de 32×32 bits qui sert à stocker et à arranger les opérandes de l'unité flottante (ces opérandes rentrent en série et sortent en parallèle). Notons qu'en réalité il existe trois transposeurs : un par entrée de l'unité flottante et que ces transposeurs servent également de buffers pour les communications.
Grâce à cette unité de calcul supplémentaire, les calculs portant sur des réels codés en simple précision sont environ 20 fois plus rapides.

Notons qu'à un niveau plus bas de la machine (accessible seulement en CMIS), on peut stocker les flottants en mettant un bit par processeur d'un même module et le transposeur ne sert alors que de buffer. Nous développerons ce point dans la section 1.2.

1.5. Processeurs élémentaires

Comme nous l'avons déjà mentionné, le niveau le plus bas consiste en un processeur élémentaire de 1 bit. Au cours d'un cycle élémentaire ce processeur a accès à deux bits provenant de sa mémoire externe et peut mettre à jour l'un des deux. Il peut aussi lire un de ses quatre flags internes et en mettre un autre à jour (éventuellement le même).
Ce processeur est capable d'effectuer n'importe quelle opération de $\{0,1\}^3$ dans $\{0,1\}^2$ (soit 216 opérations possibles). En pratique, ce processeur est une mémoire morte dans laquelle les opérations sont simplement tabulées.

La mémorisation du résultat est conditionnée par un bit, appelé *context-flag*. Si celui-ci est différent de 1, l'instruction n'a aucun effet (sauf pour certaines instructions particulières comme *always* ou *get*.

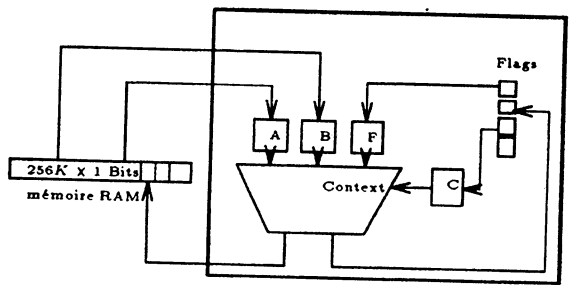


Figure A.54. Processeur élémentaire.

1.6. Routeur

Si chaque processeur de la Connection Machine peut échanger des informations avec n'importe quel autre, deux processeurs quelconques ne sont pas forcément connectés par un lien physique direct, pour des raisons évidentes de réalisation technique. Comme nous l'avons vu lors de la description de l'architecture, deux niveaux de connexion existent :

- D'une part, deux processeurs se trouvant sur un même circuit, qui en contient seize, sont directement connectés.

- D'autre part, ces circuits de 16 processeurs sont disposées en hypercube. Dans sa configuration maximale, la Connection Machine comprend 2^{16} processeurs, et donc chaque circuit est relié à 12 circuits voisins suivant un hypercube de dimension 12.

Si l'on gagne ainsi en nombre de liens de communication, certains messages vont devoir emprunter plusieurs de ces liens pour arriver à destination. Il est donc nécessaire d'avoir un routeur chargé d'acheminer les messages entre les différents groupes de 16 processeurs.
Le mécanisme complet de routage est constitué de plusieurs puces, une par groupes de seize processeurs physiques, reliées par des liens de communications. Le principe du routeur est le suivant : chaque puce reçoit des messages provenant de ses voisines, elle distribue les messages destinés à ses 16 processeurs et envoie à ses dernières les messages qui lui restent en choisissant une direction appropriée pour chacun.

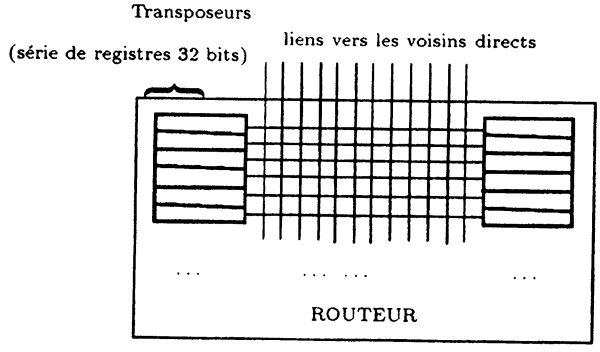


Figure A.55. Description conceptuelle du routeur : un message comporte une partie "données" plus une adresse relative. Plusieurs messages peuvent être traités simultanément, ils sont rangés par ordre de priorité et circulent de liens en liens.

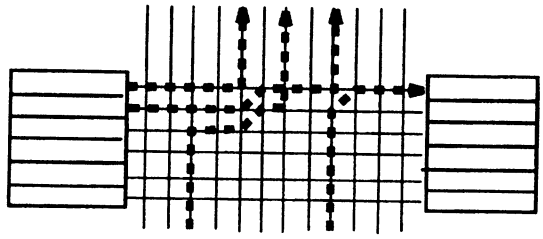


Figure A.56. Principe du routeur :les messages visitent tous les liens jusqu'à en trouver un libre et le satisfaisant.

Le message prioritaire qui peut traverser un lien, le traverse. Les messages que reçoit un module et qui ne lui sont pas destinés entrent dans le processus.
Notons que si l'on manque de place, un message peut être "bousculé" par un message entrant.

Jusqu'à présent nous n'avons parlé que des communications entre processeurs physiques. Par ordre de rapidité décroissante, voici trois cas qui peuvent se présenter lors d'une communication entre deux processeurs virtuels :

- Une communication entre deux processeurs virtuels, gérés par un même processeur élémentaire, se ramène à une simple lecture/écriture en mémoire.
- Une communication entre deux processeurs virtuels, gérés par deux processeurs élémentaires différents mais se trouvant sur un même circuit, utilise le lien direct reliant ces deux processeurs physiques.
- Une communication entre deux processeurs virtuels, gérés par deux processeurs élémentaires différents se trouvant sur deux circuits différents, utilise un ou plusieurs liens physiques entre circuits.

En conséquence, si le nombre de processeurs virtuels double, le temps mis pour un schéma de communication donné augmente sans pour autant doubler.

1.6.1. Un mot sur les périphériques rapides

La CM2 est dotée d'une grosse mémoire de masse externe à accès très rapide (le data vault). Le transfert s'effectue à un taux de 25 MegaOctets par seconde, pour 64 MegaOctets au minimum (soulignons de plus que le format des données n'est pas toujours compatible avec celui de l'hôte, ce qui nécessite un réarrangement coûteux).
Ce mécanisme devient très intéressant pour des grandes séries de données auxquelles on accède souvent (par exemple des images telles que celles provenant des satellites).

Il existe également une sortie graphique assez rapide qui permet à l'utilisateur une visualisation temps-réel de ses données (déchargement du contenu d'un champ d'un processeur dans un pixel de l'écran). Ce frame buffer est spécialement dédié à la CM2.

2. Communications

2.1. Généralités

2.1.1. Quelques remarques préliminaires sur les grilles

Comme nous l'avons vu précédemment, l'architecture physique de la Connection Machine est de type hypercube. En pratique, on raisonne la plupart du temps sur des grilles toriques, qui correspondent aux structures logiques des objets que l'on manipule.

L'anneau est une grille de dimension 1, c'est-à-dire un réseau où chaque nœud a exactement 2 voisins. De même, une grille de dimension 2, de taille $l \times m$, comprend l lignes de m processeurs, chacun ayant 4 voisins (l et m étant obligatoirement des puissances de 2, sur la Connection Machine). Il est facile de généraliser cette notion : dans une grille de dimension n, ou n-grille, de taille $\prod_{i=1}^{n} 2^{m_i}$, il y a 2^{m_i} processeurs sur chaque direction i et chaque processeur a $2n$ voisins.

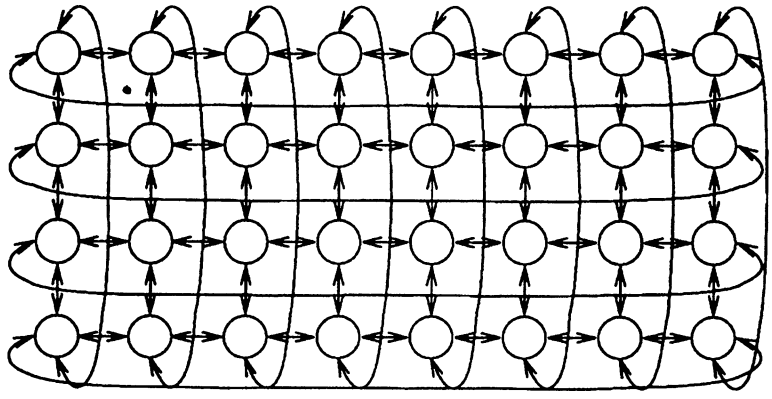


Figure A.57. Exemple de grille 8x4

Etant donné un ensemble de 2^m processeurs virtuels, n'importe quelle n-grille, de taille $\prod_{i=1}^{n} 2^{m_i}$, peut lui être associée, la seule contrainte étant qu'elle contienne autant de processeurs, c'est-à-dire que $\sum_{i=1}^{n} m_i = m$.

Propriété 1 : Soit une grille $2^i \times 2^j \times 2^k \times \cdots \times 2^q$ quelconque, le nombre de processeurs étant une puissance de 2, cette grille est alors un graphe partiel de l'hypercube de degré $(i + j + k + \cdots + q)$.

Remarque 21 : Il est possible de plonger une grille rectangulaire dans un hypercube, mais il faut alors utiliser une ou plusieurs dilatations.

Une n-grille étant associée à un ensemble de processeurs virtuels, la position d'un processeur est décrite par un n-uplet d'entiers, le i-ème étant compris entre

0 et $2^{m_i} - 1$. En raison de la structure en tore d'une n-grille, la position relative de deux processeurs le long d'un axe i se calcule modulo 2^{m_i}. La position relative de deux processeurs sur une n-grille est donc représentée par un n-uplet d'entiers, le i-ème étant compris entre 0 et $2^{m_i} - 1$.

2.1.2. Présentation des différents niveaux de communication

Le système de communication de la Connection Machine permet à chaque processeur d'échanger des informations avec n'importe quel autre processeur. Ainsi tous les processeurs actifs peuvent recevoir un message provenant d'un processeur de leur choix grâce à des instructions de type get.
Et de même, tous les processeurs actifs peuvent envoyer un message à un processeur de leur choix grâce à des instructions de type send.
Le point important est de savoir comment un processeur désigne son correspondant. Le système de communication de la Connection Machine propose deux solutions à ce problème :

- La première consiste à associer à chaque processeur une adresse qui lui est propre (Send-address). Un processeur actif désigne alors son correspondant par sa *Send-address*. Les communications invoquant ce type d'adressage sont appelées communications générales.

- La seconde solution est de disposer les processeurs sur une grille. Un processeur actif désigne alors son correspondant par sa position relative sur la grille. Les communications invoquant ce type d'adressage sont appelées communications *NEWS* par allusion avec les quatre points cardinaux (North, East, West, and South).

En fait, la différence entre les communications générales et les NEWS ne s'arrête pas à une simple différence d'adressage. En effet, lors d'une communication générale, un processeur actif peut désigner n'importe quel autre processeur comme son correspondant et ceci indépendamment des autres processeurs actifs; d'où la qualification de communication générale. Au contraire, lors d'une communication de type NEWS, la position relative d'un processeur actif vis-à-vis de son correspondant est uniforme sur toute la machine.

Nous avons donc d'un côté des communications très générales mais coûteuses en temps car il y a un risque important de conflits sur les liens physiques de communication en raison du brassage d'informations.D'un autre côté, des communications beaucoup plus spécifiques mais bien plus rapides car il n'y a qu'une translation de l'information.

Le système de la Connection Machine propose un troisième type de communication. Ces dernières consistent en une combinaison puis une redistribution des informations contenues par un ensemble de processeurs. Par exemple, la somme

d'une variable donnée sur tous les processeurs peut être diffusée à tous.

Ce type de recombinaison peut se faire sur l'ensemble des processeurs actifs, mais aussi, et c'est ce qui en fait tout l'intérêt, se faire simultanément sur des sous-ensembles de processeurs. Prenons un exemple : si les processeurs sont organisés en une grille de dimension 2, la somme sur toute une ligne de processeurs peut être diffusée à tous les processeurs de cette ligne, et ce simultanément sur toutes les lignes :

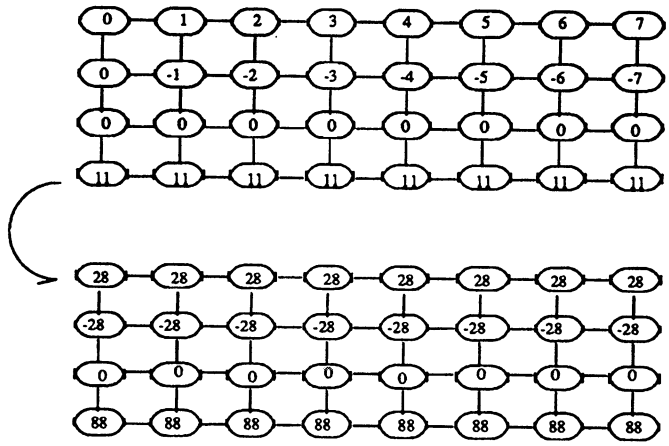


Figure A.58. Diffusion sur chaque ligne de la somme de ses composantes.

Comme les communications NEWS, les communications avec recombinaisons font intervenir l'organisation en grille des processeurs.

Par ailleurs, on peut classer parmi les communications tous les échanges d'information entre la Connection Machine et la machine hôte. Un processeur particulier peut lire ou écrire des données sur la machine hôte. De plus, des informations calculées sur l'ensemble ou une partie des processeurs peuvent être communiquées à la machine hôte. Enfin, une même valeur provenant de l'hôte peut être distribuée à tous les processeurs de la CM2.

2.2. Communications générales

Dans la suite, on appellera champ, une suite de bits consécutifs en mémoire. De plus, on prendra les conventions suivantes :

2.2.1. Send et Get

Il y a deux instructions de communication générale : *send* pour émettre et *get* pour recevoir.

Figure A.59. Représentation des processeurs

Lors d'un send, tous les processeurs actifs, et seulement ceux-ci, émettent un message vers un processeur de leur choix qui le reçoit même si il est inactif. Lors d'un get, tous les processeurs actifs, et seulement ceux-ci, reçoivent un message, provenant d'un processeur de leur choix, même si celui-ci est inactif.

Pour l'une comme pour l'autre instruction, chaque processeur a besoin de plusieurs champs : un champ *origine* contenant le message à émettre, un champ *destination* destiné à recevoir un message et un champ *adresse* contenant l'adresse du correspondant.

Lors d'une instruction *get*, chaque processeur actif p reçoit donc dans son champ *destination* la valeur du champ *origine* du processeur dont l'adresse est égale au contenu du champ *adresse* du processeur p :

$$\mathbf{destination(p) \leftarrow origine(adresse(p))}$$

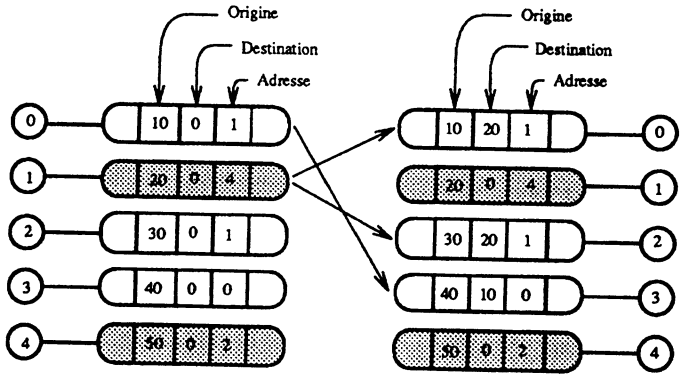


Figure A.60. Instruction get

Lors d'une instruction *send*, chaque processeur actif p envoie le contenu de son champ *origine* au processeur dont la Send-address est égale au contenu de son champ *adresse*. Celui-ci mémorise le résultat dans son champ *destination*.

destination(adresse(p)) ← origine(p)

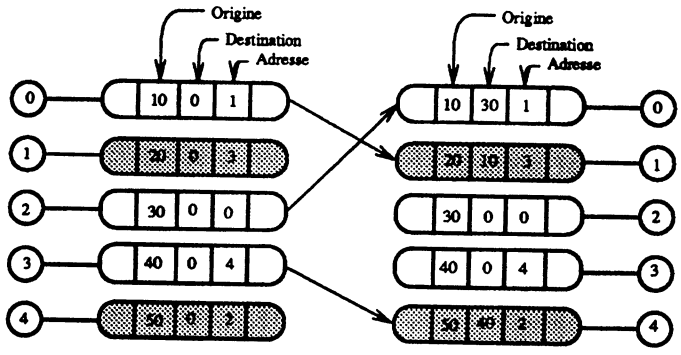


Figure A.61. Instruction send.

Remarque 22 : L'instruction *get* équivaut formellement à deux *send*. Dans un premier temps, chaque processeur désirant recevoir une donnée envoie son adresse au processeur dont il désire le contenu, dans un second temps l'information est retournée.

2.2.2. Lorsqu'un processeur reçoit plusieurs messages

Lors d'une instruction *send*, il est possible qu'un processeur reçoive plusieurs messages (Ce cas ne se présente pas lors d'une instruction *get*). Il est donc nécessaire de préciser par un paramètre le comportement de l'instruction *send* si un tel cas se présente. Il y a plusieurs solutions :

- Si l'on est sûr que le cas ne se présente pas, on peut le préciser et gagner ainsi en temps de communication car plus aucun contrôle de collision n'est fait.
- L'autre solution est de combiner tous les messages en un seul à l'aide d'une opération de combinaison. Parmi les différentes opérations de combinaison, citons l'addition et le choix d'un message.
- Dans une version à venir, les différents messages reçus pourront être stockés dans un tableau.

2.2.3. Quelques remarques

En langage ParIS, grâce à l'instruction *get-aref32*, il est possible dans le cas d'une lecture de passer outre la contrainte selon laquelle les messages ont tous la même origine dans les processeurs sources. Un processeur source peut même envoyer un message différent à chacun des processeurs demandeurs. La seule contrainte est que ces messages soient d'une longueur multiple de 32 bits.
En effet, dans l'instruction *get-aref32*, origine désigne un vecteur d'éléments et un paramètre supplémentaire, *index*, indique l'élément désiré.

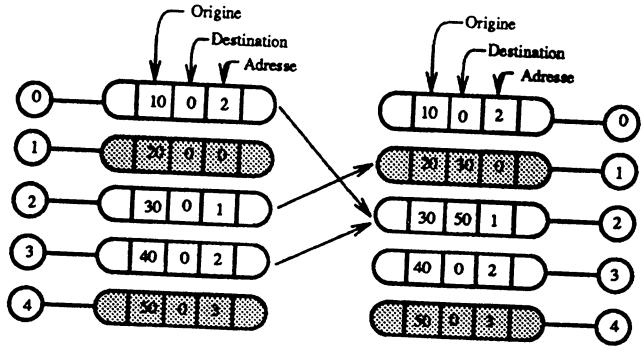


Figure A.62. Instruction send avec l'addition comme combinateur de messages.

De même pour une écriture, il est possible de passer outre la contrainte selon laquelle tous les messages arrivant dans un même processeur sont mémorisés dans le même champ destination (dans ce cas, l'instruction est *send-aref32*).

2.3. Communications de type News

2.3.1. Get-from-News et Send-to-News

Il y a deux communications de type NEWS, get-from-news et send-to-news, recevoir et émettre sur la grille. Ces deux instructions ne diffèrent de leurs homologues get et send que par le fait que chaque processeur ne choisit plus son correspondant indépendamment des autres processeurs.
Lors d'une communication NEWS, la position relative sur la n-grille, d'un processeur vis-à-vis de son correspondant, est uniforme sur la machine. Cette position relative, ou déplacement, est un paramètre de l'instruction de communication.

Lors d'un send-to-news, tous les processeurs actifs, et seulement ceux-ci, émettent un message vers le processeur de position relative "déplacement" qui le reçoit même s'il est inactif.
Lors d'un get-from-news, tous les processeurs actifs, et seulement ceux-ci, reçoivent un message, provenant du processeur de position relative "déplacement", même si celui-ci est inactif.
Pour l'une comme pour l'autre instruction, chaque processeur a besoin de deux champs :

- un champ origine contenant le message à émettre.
- un champ destination servant à recevoir un message.

Lors d'une instruction get-from-news, chaque processeur actif p reçoit donc dans son champ destination la valeur du champ origine du processeur dont l'adresse

relative sur la n-grille est égale au paramètre "déplacement" de l'instruction :
destination(p) ← origine(adresse$_{n-grille}$(p) + déplacement)

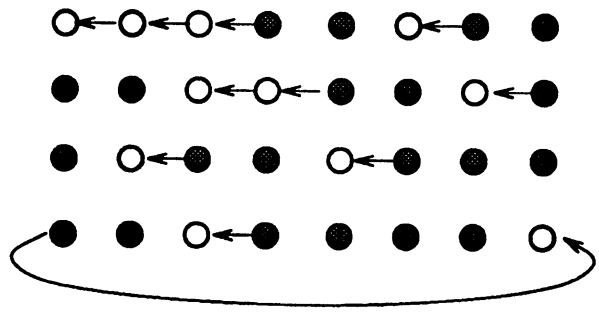


Figure A.63. Instruction get-from-news

Lors d'une instruction send-to-news, chaque processeur actif p envoie le contenu de son champ origine au processeur dont l'adresse relative sur la n-grille est égale au paramètre "déplacement" de l'instruction. Celui-ci mémorise le résultat dans son champ arrivée.

destination(adresse$_{n-grille}$(p) + déplacement) ← origine(p)

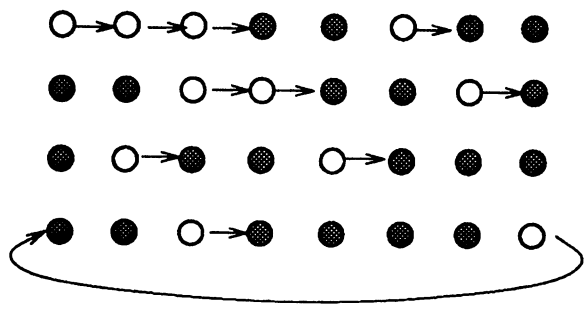


Figure A.64. Instruction send-to-news

2.3.2. n-grille associée à un ensemble de processeurs virtuels

Les instructions de communication de type NEWS font référence à une n-grille. Celle-ci doit donc être créée et associée à l'ensemble de processeurs avant toute communication de type NEWS.
Remarquons qu'il est possible de changer la n-grille associée à un ensemble de processeurs virtuels (la seule contrainte étant que le nombre de processeurs d'un vp-set donné reste inchangé). Toutefois, il n'y a pas de correspondance simple entre les positions, avant et après changement, d'un même processeur.

2.3.3. remarques

- Contrairement à une instruction send, lors d'une instruction send-to-news un processeur ne peut recevoir qu'un message. Il n'est donc pas nécessaire d'utiliser un combinateur de messages.

- Si tous les processeurs sont actifs, les deux instructions get-from-news et send-to-news sont équivalentes pour peu que leurs paramètres "déplacement" soient opposés.

2.4. Communications avec recombinaison

2.4.1. Spread, Reduce et Scan

Les communications avec recombinaison sont au nombre de trois : spread, scan et reduce, moins utile. Les processeurs sont regroupés en ensembles ordonnés que l'on appellera classes de recombinaison. Une opération de combinaison, associative, est choisie. A l'intérieur de chaque classe, l'information est combinée, par l'opération de combinaison choisie, puis redistribuée. Ces communications opèrent simultanément sur chaque classe de recombinaison. Il n'y a aucune communication entre deux processeurs s'ils ne font pas partie de la même classe.

Lors d'une instruction spread (diffusion) la combinaison de l'information de tous les processeurs d'une même classe de recombinaison est redistribuée à chacun d'eux.

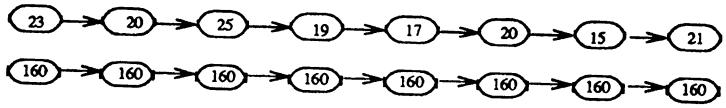


Figure A.65. spread avec addition.

Lors d'une instruction reduce, la combinaison de l'information de tous les processeurs d'une même classe de recombinaison est retournée à l'un d'entre eux et seulement un.

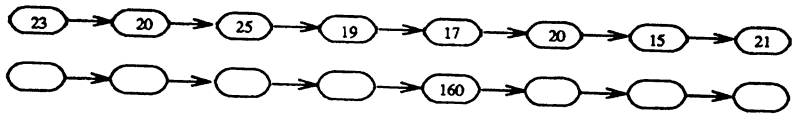


Figure A.66. reduce avec addition.

Contrairement aux instructions spread et reduce, l'instruction scan, ou parallel prefix [11], fait intervenir l'ordre des processeurs à l'intérieur de leur classe de recombinaison. Lors d'une instruction scan, chaque processeur reçoit la combinaison de l'information de tous les processeurs qui le précédent.

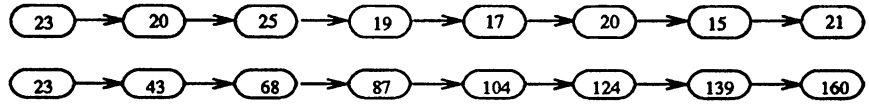


Figure A.67. scan avec addition.

Ces communications régulières utilisent en fait la topologie en hypercube avec une complexité logarithmique au lieu de linéaire.

2.4.2. Lien entre les classes de recombinaison et la grille

Les classes de recombinaison d'un ensemble de processeurs virtuels sont définies à partir d'une n-grille associée à cet ensemble de processeurs et d'un axe de cette n-grille.

Soient un ensemble de 2^m processeurs virtuels, une n-grille comprenant 2^{m_i} processeurs sur chaque axe i, avec $\sum_{i=1}^{n} m_i = m$, et un axe I. Par définition, deux processeurs appartiennent à la même classe de recombinaison suivant l'axe I si et seulement si leurs coordonnées sur la n-grille ne diffèrent que suivant la I-ème composante. A l'intérieur d'une même classe, les processeurs sont ordonnés par ordre croissant de leur I-ème coordonnée.
En d'autres mots, les communications avec recombinaison le long d'un axe I combinent l'information sur chaque ligne de processeurs parallèle à l'axe I.

Ainsi, dans le cas d'une 3-grille $(4, 2, 4)$, il y a huit classes de recombinaison suivant l'axe 1. Deux processeurs de la même classe ayant même couleur, on obtient la figure :

Les classes de recombinaison dépendant de l'axe choisi, celui-ci est un paramètre des instructions de communication avec recombinaison. Par contre, la n-grille est par défaut celle associée à l'ensemble de processeurs.

2.4.3. Regroupement des classes de recombinaison.

Dans le cas d'une instruction de diffusion (spread) il est possible de regrouper les classes de recombinaison définies par la n-grille, mais toute information d'ordre à l'intérieur de la classe est perdue (c'est pourquoi il n'existe pas de tel regroupement pour l'instruction scan).
Etant donné $I_1, \cdots, I_k$ les k axes d'une n-grille; deux processeurs appartiennent

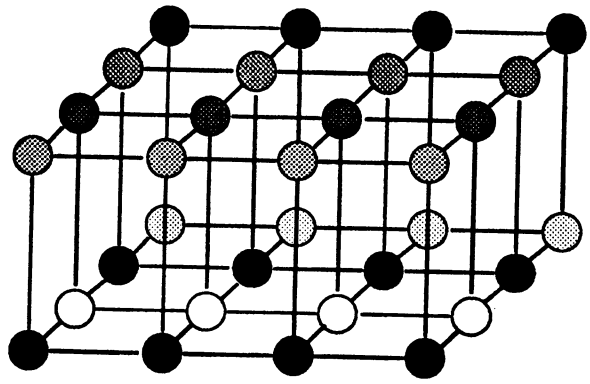


Figure A.68. Classes de recombinaison pour une grille $4 \times 2 \times 4$ le long de l'axe 1.

à la même classe de recombinaison suivant ces axes si et seulement si leurs coordonnées sur la n-grille ne différent que suivant les composantes $I_1, \cdots, I_k$. L'instruction de diffusion, invoquant de telles classes de recombinaison s'appellent multi-spread en ParIS.

2.4.4. Segmentation des classes de recombinaison.

Dans le cas d'instruction scan, il est possible de fragmenter les classes de recombinaison définies par la n-grille. Etant donné un champ segment par processeur, ce champ contenant soit 0 soit 1, un processeur est le premier élément d'une classe segmentée le long de l'axe I si et seulement si il est le premier élément d'une classe le long de l'axe I ou si son champ segment est égal à 1. L'ordre des processeurs dans les classes segmentées est le même que dans les classes non segmentées.

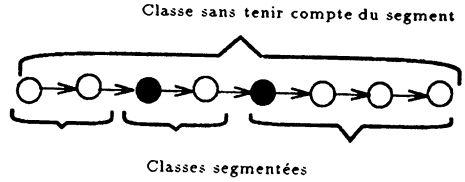


Figure A.69. Segmentation des classes de recombinaison. Les processeurs dont le champ segment est à 1 sont en noir.

2.4.5. Effet des processeurs inactifs.

Les processeurs inactifs n'interviennent pas dans le résultat des communications avec recombinaisons. Leur contenu reste inchangé et n'influe pas sur le résultat des autres processeurs de la même classe. Ils sont donc tout simplement ôtés des

classes de recombinaison.
Lors d'une instruction reduce, le processeur, désigné pour recevoir la combinaison des informations de sa classe, ne la reçoit effectivement que s'il est actif.

2.5. L'instruction Scan : variantes et exemple.

Le scan est en fait une opération plus connues sous le nom : Parallel Prefix [11]. La Connection Machine offre de nombreuses variantes de l'instruction scan. Comme cette instruction est peut être la plus originale par rapport à l'algorithmique séquentielle, nous en donnons un exemple d'utilisation après avoir passé en revue toutes ses variantes.

2.5.1. Les différentes variantes de l'instruction SCAN.

La définition de l'instruction scan, que nous avons utilisée jusqu'à présent peut être résumée de la façon suivante : chaque processeur actif reçoit la combinaison de l'information de tous les processeurs actifs qui le précédent dans sa classe de recombinaison.
Comme nous l'avons vu, une première variante consiste à segmenter les classes de recombinaison définies par la n-grille et un de ses axes. C'est le scan segmenté.
Une deuxième variante consiste à prendre le terme "qui le précède" au sens strict (le contenu d'un processeur n'intervient plus sur le résultat de ce même processeur). On parle alors de scan exclusif que l'on oppose au scan inclusif, usuel.
La troisième variante consiste à effectuer le scan dans l'ordre décroissant des processeurs. On parle de scan croissant et de scan décroissant.

Ces différentes options se combinent entre elles. Voici une récapitulation sur un même exemple (scan avec addition sur 16 processeurs contenant initialement tous, dans leur champ source, la valeur 1) :

Pour la figure, les conventions sont les suivantes :

- Les processeurs inactifs sont en gris.
- Les processeurs dont le champ segment est à 1 sont entourés.
- Les classes de recombinaison sont représentées par des flèches indiquant l'ordre suivant lequel le scan est effectué.
- La première ligne correspond à la configuration initiale du champ source.
- La deuxième ligne correspond à la configuration du champ destination après l'exécution du scan; l'absence de valeur indiquant que ce champ n'a pas été modifié.

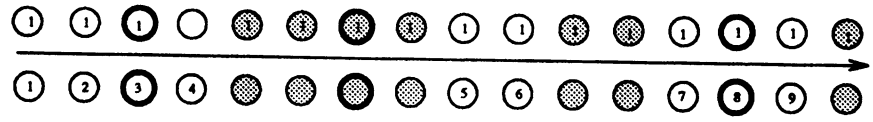


Figure A.70. scan inclusif croissant.

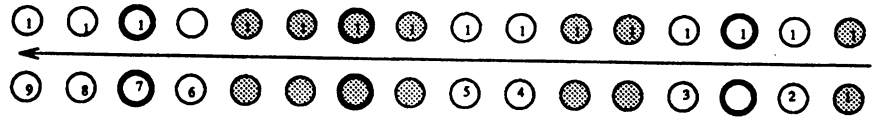


Figure A.71. scan inclusif décroissant.

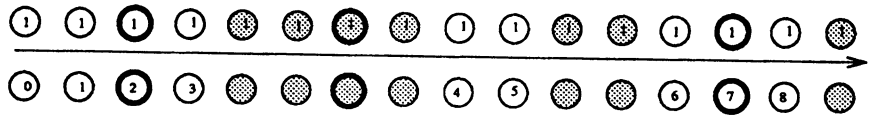


Figure A.72. scan exclusif croissant

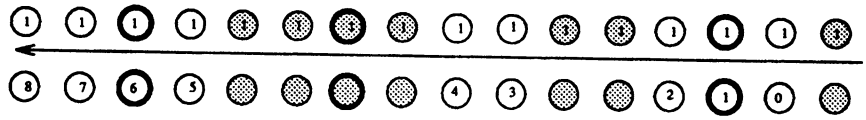


Figure A.73. scan exclusif décroissant.

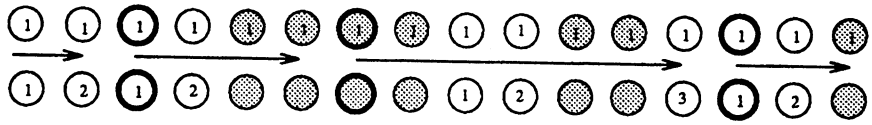


Figure A.74. scan segmenté inclusif croissant.

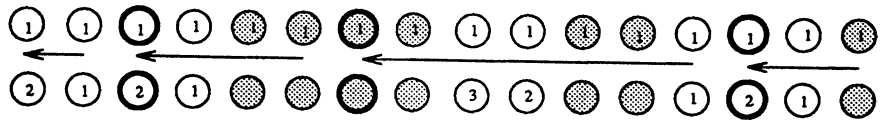


Figure A.75. scan segmenté inclusif décroissant.

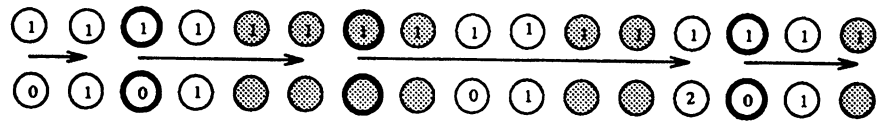


Figure A.76. scan segmenté exclusif croissant.

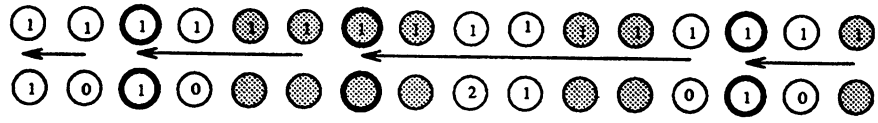


Figure A.77. scan segmenté exclusif décroissant.

Une autre variante est le scan avec bit-de-départ. Le champ segment s'appelle alors bit-de-départ et l'opération se déroule comme si les processeurs étaient parcourus séquentiellement : le scan redémarre en chaque processeur actif dont le bit-de-départ est à 1. Contrairement aux autres variantes de scan, lors d'un scan avec bit-de-départ, les classes de recombinaison dépendent du contexte et du sens de parcours. (Pour une information complète, se référer à la documentation fournie par TMC sur le langage ParIS).

2.5.2. Exemple

Considérons un ensemble ordonné de processeurs, chacun contenant un nombre mémorisé dans un champ A. On veut trier ces nombres, c'est-à-dire les redistribuer de telle façon que pour tout couple (p, q) de processeurs si p est "avant" q alors le contenu de p est plus petit que celui de q.
L'algorithme que nous allons détailler est directement inspiré de l'algorithme Quick-sort séquentiel. Le principe est de choisir une clé (i.e un nombre) et de scinder l'ensemble en trois partie :

- une première comprenant tous les nombres strictement inférieurs à la clé,
- une deuxième comprenant tous les nombres égaux à la clé,
- une troisième comprenant tous les nombres qui lui sont strictement supérieurs.

Il n'y a donc plus de comparaisons à faire entre éléments de parties différentes et il suffit de réitérer ce principe sur chacune d'entre elle.

En prenant comme clé le premier élément de chaque ensemble, on passe donc de :

{ 5 7 3 2 4 1 6 2 } à { 3 2 4 1 2 } { 5 } { 7 6 }

Et avec deux étapes supplémentaires on obtient donc le résultat souhaité:
d'abord: { 2 1 2 } { 3 } { 4 } { 5 } { 6 } { 7 }
et enfin: { 1 } { 2 } { 2 } { 3 } { 4 } { 5 } { 6 } { 7 }

Sur la Connection Machine, l'ordre des processeurs est géré en associant à l'ensemble des processeurs une 1-grille. Dans chaque processeur, un champ position mémorise l'adresse du processeur sur cette 1-grille.
La fragmentation de l'ensemble des processeurs en plusieurs parties est gérée par un champ segment de 1 bit par processeur de telle façon que les classes de recombinaison du scan segmenté correspondent à ces parties. Le fait d'utiliser des scan permet de travailler simultanément sur toutes les classes.
Pour chaque classe, la clé choisie est le contenu du premier processeur de la classe.

La fragmentation de chaque classe en trois parties est faite en plusieurs étapes élémentaires:

i) Le premier élément de chaque classe, A, est diffusé dans le champ clé de tout les processeurs de la classe:
scan avec copie de A dans clé, segmenté, inclusif, croissant

ii) Chaque processeur a trois champs de 1 bit inf, eq et sup destinés à conserver le résultat de la comparaison du contenu du processeur avec la clé. Ceux-ci sont mis à jour:
si $A <$ clé alors inf=1, eq=0 et sup=0
si $A =$ clé alors inf=0, eq=1 et sup=0
si $A >$ clé alors inf=0, eq=0 et sup=1

iii) Chaque processeur calcule sa place dans sa sous-classe. Le résultat est conservé dans les champs inf_n°,eq_n° et sup_n°.
scan avec addition de inf dans , segmenté, inclusif, croissant
scan avec addition de eq dans eq_n°, segmenté, inclusif,croissant
scan avec addition de sup dans sup_n°, segmenté, inclusif, croissant

iv) Le nombre d'éléments dans chaque sous-classe, connu par le dernier élément de chaque classe est diffusé à tous les processeurs de la classe dans les deux champs nb_inf, nb_eq (le nombre d'éléments supérieurs à la clé n'est pas utile).
scan avec copie de inf_n° dans nb_inf, segmenté, inclusif, décroissant
scan avec copie de eq_n° dans nb_eq, segmenté, inclusif, décroissant

v) La position dans la 1-grille du premier processeur de chaque classe est diffusée dans le champ base des processeurs de sa classe.
scan avec copie de position dans base, segmenté, inclusif, croissant

vi) Chaque processeur calcul la nouvelle position de son contenu.
si inf=1 alors nouvelle_position=base-1+inf_n°
si eq=1 alors nouvelle_position=base-1+nb_inf+eq_n°
si sup=1 alors nouvelle_position=base-1+nb_inf+nb_eq+sup_n°

vii) Dans chaque processeur, nouvelles_position (qui est une adresse sur la 1-grille) est transformée en une Send-address, nouvelle_adresse.
nouvelle_adresse = Send-address(nouvelle_position)

viii) Chaque processeur envoie son contenu A dans le champ A du processeur indiqué par nouvelle-adresse.
send de A dans A à nouvelle-adresse

ix) Le champ segment est mis à jour pour tenir compte de la segmentation de chaque classe en trois classes.
si position = base + nb_inf alors segment=1
si position = base + nb_inf + nb_eq alors segment=1

x) Les processeurs dont le contenu est égal à la clé sont inactivés.
si base + nb_i $\leq$ position $<$ base + nb_inf + nb_eq alors context-flag=0

On recommence ces 10 étapes jusqu'à ce que tous les processeurs soient inactifs. Le tri est alors fini. (Pour l'expression de ce test de terminaison, se référer à l'instruction global du paragraphe 6.3.)

2.6. Communications avec l'hôte

2.6.1. Echange d'informations entre l'hôte et un processeur.

Grâce aux deux instructions read-from et write-to, l'ordinateur hôte peut échanger des informations avec un processeur virtuel de la Connection Machine. Une variable de l'hôte peut donc prendre pour valeur celle d'un champ d'un processeur et de même un champ d'un processeur peut prendre pour valeur celle d'une variable de l'hôte. Notons que dans les deux cas l'échange d'information a lieu même si le processeur virtuel est inactif.

2.6.2. Echange d'informations entre l'hôte et un ensemble de processeurs.

Une sous-grille de dimension k de la n-grille peut échanger des informations avec l'hôte grâce aux deux instructions read-from-news-array et write-to-news-array. La valeur d'un champ sur une sous-grille peut être ainsi transférée de ou vers un tableau de l'hôte ayant même dimensions que la sous-grille. Notons que dans les deux cas l'échange d'information a lieu même si comme précédemment les processeurs virtuels sont inactifs.

Une valeur de l'hôte peut être aussi diffusée à tous les processeurs actifs grâce à l'instruction move-constant.

2.6.3. Réduction de l'information de tous les processeurs.

L'instruction globale, associée à une opération de combinaison associative, permet de récupérer, dans une variable de l'hôte, la combinaison des valeurs d'un champ de tous les processeurs actifs de la Connection Machine. Bien que cette instruction soit importante, elle n'est pas très efficace à cause du délai de communication entre le séquenceur et l'hôte.
Grâce à une instruction globale, associée au "ou logique" et opérant sur le context-flag, il est ainsi possible de savoir s'il existe un processeur actif. Nous avons donc le test de terminaison requis par le programme précédent de tri.

2.7. Le Communication Compiler

Beaucoup d'applications réelles ont leur efficacité qui diminuent sensiblement à cause du coût prohibitif des communications. En effet lors de l'appel d'une communication de type général (*send* et *get*), la machine utilise un schéma de communication qui travaille au coup par coup, sans stratégie globale, et donc qui conduit à des temps trop importants.
Il semble donc intéressant d'avoir un outil qui optimise les communication pour diminuer le coût global. C'est pourquoi TMC et Dahl [33] ont développé un interface : le Communication Compiler. Il permet de créer un graphe statique associé à une communication donnée. Mais développons plus en détails les différentes étapes qui permettent d'obtenir ce graphe final.

Il faut tout d'abord résoudre un problème de scheduling, puis un problème de routage.

2.7.1. Le scheduling

Sur un réseau fixé de processeurs, il existe deux types de problèmes différents :

1. Soit on connaît le nombre de message à envoyer, leur taille, leur source et leur destination, et alors on se ramène à des schémas de communication de type régulier tels que le broadcast, le All-To-All ; schémas que nous décrivons dans le chapitre II. Dans ce cas, il existe des algorithmes qui permettent de résoudre séparément chaque schéma [102, 66, 52, 94].

2. Soit le schéma de communication est un variable du problème à résoudre, et alors il faut déterminer pour chaque message un chemin, et trouver une solution qui permette d'effectuer toutes les communications sur l'ensembles des chemins de façon optimale. C'est ce problème que nous allons étudier.

Un chemin est défini comme une suite non ordonnée de sauts élémentaires d'un processeur à un autre. Le but du Communication Compiler est de décomposer l'ensemble des chemins en sauts élémentaires, puis selon certains critères, de choisir à chaque étape dans l'ensemble des sauts restants ceux qui vont minimiser le

nombre total d'étapes.

Les contraintes de la fonction à minimiser sont les suivantes :

- La localisation des messages restants à faire parvenir à destination (dépend de la communication initiale et des étapes précédentes).
- Le nombre maximum de messages que peut envoyer un processeur lors d'une étape (dépend du réseau d'interconnexion de la machine).

2.7.2. Le routage

Il s'agit de trouver un algorithme de routage qui en fonction du modèle de communication et des paramètres choisis, peut à chaque étape trouver une solution à la proposition faite par l'étape de scheduling. En fait, à chaque étape, le Communication Compiler va assigner les messages aux liens qu'ils vont devoir suivre en fonction des deux contraintes suivantes :

- Un message ne peut être déplacé deux fois lors de la même étape.
- Le nombre de liens utilisés lors d'un étape ne peut dépasser le nombre physique (dépend de l'architecture).

Le principe d'assignation est basé sur la théorie des graphes. Le problème à résoudre se modélise sous la forme d'un graphe biparti où le premier ensemble correspond aux messages et le deuxième aux liens. Un sommet message m_i est relié à un sommet lien $lien_j$ si et seulement si le message m_i utilise le lien $lien_j$ lors du cycle de communication.

2.7.3. La solution

A chaque étape, on a un nouveau problème à résoudre, car à chaque étape on se retrouve dans une nouvelle configuration qui demande une nouvelle recherche, c'est-à-dire un nouveau problème de scheduling et un nouveau problème de routage. La figure A.78 décrit le caractère itératif du Communication Compiler.

Il existe plusieurs algorithmes qui permettent de résoudre de façon optimale ce type de problème, mais malheureusement leurs implantations, même en parallèle, ont un coût prohibitif. Dahl propose dans [33] une heuristique qui ne garantit pas une solution optimale, mais une solution très proche en un temps raisonnable. Cette heuristique cherche à maximiser le nombre de message à envoyer lors de chaque étape. Au niveau du graphe, cela revient à diminuer au maximum le nombre d'arêtes sortant des différents sommets, message et lien à la fois. Le critère d'arrêt est toujours le même : ne plus avoir de communication à effectuer. Dans la pratique, l'algorithme est basé sur la méthode du recuit simulé.

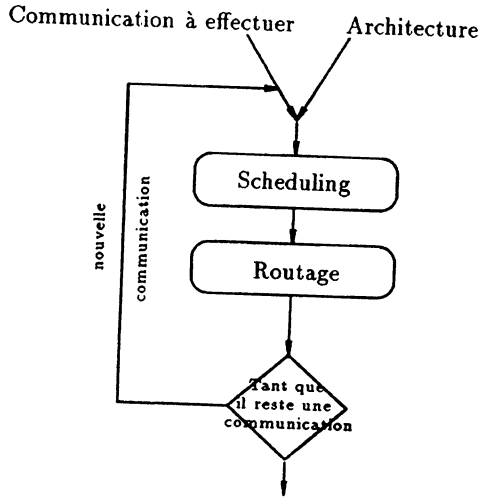


Figure A.78. Schéma itératif du Communication Compiler

Le principe d'utilisation du Communication Compiler est le suivant: On doit tout d'abord exécuter une première fois la communication générale avec la bonne taille de messages, c'est-à-dire effectuer une première fois le programme. A la fin de cette première passe, on obtient un fichier contenant le graphe optimisé. On peut alors exécuter une deuxième fois le programme avec en paramètre le graphe optimisé, il suffit d'une simple lecture de ce fichier lors de l'appel de la communication pour effectuer cette dernière.

Avant de montrer le gain obtenu, il est nécessaire de préciser trois critiques qui limitent sensiblement les possibilités d'utiliser le Communication Compiler en toute circonstance:

- Le temps nécessaire pour la création du fichier contenant le graphe optimisé, bien que diminué avec le nouvel algorithme, reste encore très important. Par exemple pour réaliser le transposition d'une matrice de taille 1024×1024, il faut compter plus d'une heure pour obtenir le fichier. Il n'est donc pas rentable d'effectuer ce travail si on utilise pas souvent le programme.
- Le temps d'accès au fichier lors du premier appel est suffisamment grand pour faire chuter les performances. Il faut donc utiliser plusieurs fois la même communication pour être efficace, car alors le temps d'ouverture du fichier devient négligeable. [109] est un exemple typique d'utilisation justifiée du Communication Compiler.
- La taille du fichier est très imposante, elle peut dépasser 3Mo selon la taille du problème.

2.7.4. Les performances

Ce travail expérimental a été réalisé en collaboration avec C. Calvin [16]. Les tests ont été effectués sur des matrices carrées de taille $N = 128$ et 256 avec cinq types de communications différentes :

1. Communications *Send* de taille N régulières, c'est-à-dire le long des colonnes de la matrice.
2. Communications *Get* de taille N régulières, c'est-à-dire le long des colonnes de la matrice.
3. Communications *Send* de taille N^2 structurés avec la transposition de la matrice.
4. Communications *Get* de taille N^2 structurés avec la transposition de la matrice.
5. Communication *Send* de taille N^2 mais aléatoires.

Les résultats sont les suivants :

	Sans Comm Comp		Avec Comm Comp	
	N=128	N=256	N=128	N=256
Send Régulier	0.3	2.7	0.25	0.7
Get Régulier	1.6	4	0.25	0.7
Transposition (Send)	2	15	1.3	5
Transposition (Get)	10	33	1.3	5
Send Aléatoire	16	355	30	5

Tableau A.13. Temps en seconde des différentes communications.

La première chose qui apparaît sur ce tableau, c'est la régularité de la dernière colonne. En effet, pour un problème de taille N, on obtient toujours 0.7, et pour N^2 toujours 5. Cela signifie :

1. Un Get deux fois plus coûteux normalement est transformé en un Send, ou plus exactement le fichier obtenu ne fait pas la différence entre les deux types de communication.
2. Pour l'ensemble des exemples, on a atteint la borne inférieure, et que toutes les communications ont été réduites au même nombre minimum d'étapes élémentaires.

La deuxième remarque un peu une conséquence de la première, est que le gain dû au Communication Compiler est plus important pour un Get que pour un Send. En fait c'est surtout le gain du Send qui est très faible. Un exception

semble se dégager avec la communication aléatoire qui elle obtient un gain de plus de 700%.
Pour finir, il est évident que la taille influence les performances du Communication Compiler. Plus la taille augmente, plus les communications sont nombreuses mais plus aussi les communications seront internes au même processeur physique diminuant ainsi le nombre global de communications à effectuer entre processeurs distants. La communication de type Send aléatoire en est un exemple parfait.

3. Langages

3.1. Programmation parallèle

Les années qui viennent de s'écouler ont été marquées par l'apparition de réalisations matérielles de machines parallèles. Des langages spécifiques ont été développés pour coordonner l'action des processeurs et contrôler les échanges d'informations.
Dans le cas d'un parallélisme synchrone S.I.M.D., ces langages sont finalement peu différents des langages séquentiels car ils nécessitent un contrôle unique des instructions. Les nouveautés consistent essentiellement en la possibilité de sélectionner les processeurs qui exécuteront les instructions et d'échanger des informations entre processeurs par l'adjonction de primitives de communication.

3.1.1. Langages

Sur la Connection Machine, l'utilisateur dispose de plusieurs langages parallèles dérivés sur les langages usuels: *CM-Fortran, C* et *Lisp.* Historiquement, la Connection Machine était dédiée à des applications en intelligence artificielle, ce qui explique le développement initial de *Lisp. Le CM-Fortran, langage préféré pour les applications numériques, est assez efficace. C* était encore en cours de développement lors de la rédaction initiale de ce document, et surtout n'était pas très efficaces lors de nos premières expérimentations. Au niveau des performances, le Fortran est un peu moins rapide que le Lisp car plus évolué.
Il est possible de programmer à un niveau inférieur: ParIS (Parallel Instruction Set). Ce langage peut être comparé à un assembleur très évolué et donc demande à l'utilisateur une attention plus précise pour contrôler le déroulement des instructions. Cependant, il est bien plus efficace que les autres langages et finalement assez pratique à l'usage. De plus en plus, les utilisateurs recommandent l'emploi de C/ParIS, qui est un C séquentiel contenant les primitives parallèles supplémentaires en ParIS.
On dispose de plus d'un simulateur de *Lisp qui permet une mise au point à distance de programmes sur des géométries de petite dimension. Cela dit, il faut disposer d'une machine assez grosse (d'un Sun4 avec 24 MegaOctets de mémoire centrale ou d'un Vax) pour développer confortablement. Notons que ces simulateurs risquent de disparaître peu à peu.

Avant de présenter tous ces langages individuellement, il convient de parler des données et de leur organisation à l'intérieur de la Connection Machine. La dernière section présente plus longuement *CMIS* l'assembleur de la Connection Machine.

3.1.2. Notion de variables parallèles, Opérateurs séquentiels et parallèles

Le but des langages parallèles de type S.I.M.D. est de permettre la réalisation simultanée d'une même opération sur les éléments que l'on a sélectionnés. Un algorithme sur la Connection Machine est composé de deux classes d'instructions : la première contient les instructions destinées à l'ordinateur séquentiel (hôte), la deuxième comprend les instructions parallèles destinées aux processeurs de la Connection Machine.

- Dans cette première classe, on retrouve toutes les instructions de l'hôte. Elles opèrent sur des variables dites scalaires, gérées et mémorisées sur l'hôte.
- Pour la deuxième classe, celle des instructions parallèles, tous les processeurs élémentaires ont un élément dans leur mémoire locale, à la même place, et toutes les opérations portent sur chacun individuellement au même instant. Les opérateurs parallèles sont identiques à ceux du cadre séquentiel, seul le fait de préciser que l'opération est parallèle, déclenche l'exécution sur la Connection Machine.

En plus de ces opérateurs "usuels", des opérations de communication ainsi que de sélection d'un sous-groupe de processeurs ont été ajoutées. Les communications sont nécessaires dès que des échanges d'informations sont requis entre processeurs. Pour la sélection, tout se passe pour l'utilisateur comme si seuls les processeurs sélectionnés travaillaient, alors que dans la pratique tous les processeurs exécutent cette instruction, la mémoire n'étant modifiée que pour les processeurs actifs. Cette opération conditionnelle est réalisée en créant un masque à l'aide d'une variable parallèle booléenne (flag-context) qui est à vrai sur les processeurs où la condition est vérifiée.

Notons que des opérations communes entre variables scalaires et parallèles peuvent être réalisées grâce à la duplication de la variable scalaire sur différents processeurs à l'aide d'un schéma de communication interne à la machine.

3.2. Le Fortran

3.2.1. Généralités

Depuis un an environ, le CM-Fortran est devenu la base du nouveau Fortran 90. Il est construit sur les bases du Fortran 77, et complété d'une partie du Fortran 8× (Norme d'un Fortran amélioré qui permet de traiter les tableaux comme variable à part entière, ce qui facilite le traitement vectoriel ou parallèle). Par exemple, l'addition de deux tableaux A et B de taille identique s'écrit sous la forme d'une instruction simple :

$$C = A + B.$$

Sur la Connection Machine, chaque élément d'un tableau est associé à un processeur virtuel pour réaliser l'addition d'un seul coup. CM-Fortran utilise tous les avantages du Fortran 8× lors de l'implémentation d'algorithmes parallèles sur Connection Machine. Il est à remarquer que tous ces programmes peuvent être transférés facilement sur toute autre machine supportant le standard 8×, et s'exécuter sans aucun problème.
Cependant, le CM-Fortran contient plus d'opérateurs sur les tableaux que le standard 8×, comme des opérateurs mathématiques (Diagonal, Replicate, Rank), ou encore des opérateurs moins connus nécessitant éventuellement des communications (Spread, Eoshift, Merge). Ce sont surtout ces nouveaux opérateurs qui seront explicités ci-dessous [29].

3.2.2. Le traitement de tableaux en CM-Fortran

Comme nous l'avons dit précédemment, le Fortran 8× permet de traiter les tableaux comme un objet unique, équivalent à une variable scalaire. Sur la Connection Machine, ces tableaux sont stockés avec un élément par processeur virtuel, et s'adaptent parfaitement à la grille multidimensionnelle associée (par exemple une 2-grille pour une matrice).
On retrouve la plupart des fonctions habituelles de traitement des tableaux rencontrées en Fortran 77. Une des seules restrictions importantes est que les tableaux doivent être de même dimension pour tous les opérateurs binaires comme l'addition ou la multiplication. En effet, dans ce cas la Connection Machine alloue le même processeur virtuel aux deux éléments correspondants. Ainsi lors de l'addition des deux matrices A et B, les éléments $A(1,1)$ et $B(1,1)$ seront sur le même processeur virtuel. Cela élimine des déplacements inutiles de données.

En plus de ces opérateurs élémentaires, le CM-Fortran contient des fonctions supplémentaires portant soit sur les caractéristiques des tableaux, soit sur leurs transformations ou encore sur des opérations plus complexes.

- Les fonctions caractéristiques renvoient exclusivement des informations sur leurs arguments, basées sur les propriétés des tableaux (comme le rang, le nombre d'éléments par sous-dimensions, etc..).
- Les fonctions complexes travaillent sur des tableaux dont les éléments sont des petits vecteurs ou matrices. On dispose alors localement des opérations classiques (produit-scalaire, multiplication de matrices, etc..).
- Les fonctions de transformation, appelées ainsi car les résultats obtenus sont de "forme" différente par rapport aux entrées, sont elles-mêmes divisées en trois sous-groupes :
  - Les fonction de réduction,

- les fonctions de construction,
- les fonctions de manipulation.

Examinons plus en détail ces fonctions.

Les fonctions de réduction sont les plus intéressantes. Elles permettent de manipuler des opérations selon une dimension unique du tableau. Par exemple, si A une matrice 45×30, alors Sum(A, dim=1) retourne comme résultat un vecteur de taille 30 contenant la somme de chaque colonne. Un masque peut être utilisé afin de limiter la portée de l'opérateur de réduction comme dans le cas de l'algorithme de Gauss pour la recherche du pivot dans une colonne:

maxval (A, Mask= ((i.GT.Numcol) and (Numcol.EQ.j)), dim=2)

où i et j sont respectivement les numéros de lignes et de colonnes, et Numcol le numéro de la colonne où l'on recherche le pivot.

Les secondes, fonctions de construction, construisent à partir de différents types d'éléments (tableaux, vecteurs, scalaires) et selon la fonction utilisée, de nouveaux tableaux qui peuvent être de taille quelconque. Les plus utiles sont le spread qui recopie un tableau selon une nouvelle dimension, merge qui incorpore un tableau dans un autre selon un masque booléen.
En Fortran, l'instruction spread n'est pas tout à fait identique à celle qui a été présentée dans la section précédente. En effet, on crée une matrice de dimension supérieure, ce qui implique la modification de la topologie logique employée.

Le troisième groupe, fonctions de manipulation, n'altère pas la forme du tableau initial. Les fonctions qui le composent, sont : transpose qui comme son nom l'indique, transpose une matrice, cshift qui décale d'un nombre fini de fois le tableau de façon circulaire suivant une dimension donnée, et eoshift qui fait de même sans boucler sur lui même.

Remarque 23 : Beaucoup de ces fonctions utilisent les communications sur les dimensions de l'hypercube sans que l'utilisateur ne s'en préoccupe. Pratiquement, il raisonne directement sur la structure logique des tableaux (grilles).

3.2.3. La condition

Comme nous venons de voir, l'utilisation d'un masque dans les opérations de réduction est un exemple de sélection d'un groupe de processeurs afin d'y réaliser une instruction unique. Le même principe est employé de façon plus générale par l'instruction *Where*. Le masque qu'elle utilise a une portée plus globale qui ne se limite pas uniquement aux tableaux, et surtout qui permet d'effectuer une opération sur les deux sous-ensembles opposés, créés par le masque. L'exemple

suivant montre plus clairement les choses :

```
Where (A .GE. 0.0)
      A = Sqrt (A)
elsewhere
      A = Sqrt ((-A))
end where
```

Le résultat de cette opération est un tableau contenant la racine carrée de la valeur absolue de A. Attention, l'opération nécessite une décomposition en deux sous-opérations qui sont exécutées l'une après l'autre.

Lorsque l'utilisateur travaille sur des tableaux, il est possible de remplacer where par l'instruction forall. Ces deux instructions différent entre elles uniquement dans la définition du masque. En effet, au lieu de faire un test général sur une condition, forall utilise les indices comme le for usuel. Par exemple l'initialisation de la matrice de Hilbert est la suivante :

$$\textbf{forall}\ (i = 1 \cdots n\ ,\ j = 1 \cdots n)$$
$$H(i,j) = \frac{1.0}{Real(i+j-1)}$$

Remarque : l'utilisation d'un masque supplémentaire permet de subdiviser une nouvelle fois la forme géométrique délimitée par l'intervalle des indices. Sur l'exemple suivant, on annule la partie inférieure de la matrice précédente H :

$$\textbf{forall}\ (i = 1 \cdots n\ ,\ j = 1 \cdots n,\ i.\mathrm{GT}.j)$$
$$H(i,j) = 0.0$$

CM-Fortran étant un langage évolué, les différents ordres de condition peuvent s'imbriquer les uns dans les autres avec l'emploi d'une pile de sauvegarde du contexte précédent.

3.2.4. Les communications en CM-Fortran

En CM-Fortran, il n'existe pas de fonction de communication proprement dite, elles sont toutes incluses dans les fonctions présentées ci-dessus. Il est possible, par exemple, de transférer une partie d'un tableau dans une autre et même dans un autre tableau, et cela grâce au concept de section de tableau du standard 8×. L'exemple suivant, où l'on transfère le bloc supérieur gauche 5×5 de la matrice A de dimension 10×10 dans le bloc inférieur droit, explique le principe de ce nouveau concept :

$$A(1:5, 1:5) = A(5:10, 5:10)$$

Une extension de cette opération permet de créer un nouveau vecteur B, à partir des valeurs d'un vecteur V à l'aide d'un autre vecteur S contenant les indices des éléments de V à recopier dans B. Les deux vecteurs B et S sont forcément de même taille. L'opération est la suivante:

$$V = [20, 5, 78, 24, 50]$$
$$S = [1, 4, 5, 2, 3, 1, 1, 3, 5]$$
$$B = V(S) = [20, 24, 50, 5, 78, 20, 20, 78, 50]$$

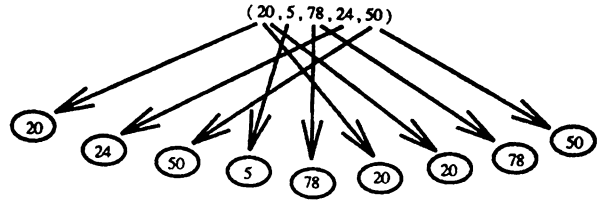


Figure A.79

Remarque 24 : Si de plus, les longueurs de V et S sont identiques et que S n'a que des nombres distincts, on obtient une permutation des éléments.

3.2.5. Conclusion

La caractéristique majeure du CM-Fortran réside dans le fait que l'utilisateur n'a pas directement accès aux procédures de communication, et que celles-ci sont implicites dans des manipulations de tableaux. Malgré tout, le code généré peut ne pas être optimal. Par exemple, la fonction cshift qui permet une circulation des éléments selon une dimension, utilise les primitives NEWS pour les communications. Mais dans le cas où la dimension choisie n'est pas une puissance de 2, le tore physique ne peut être utilisé en totalité, et un send plus coûteux est employé par le système pour compléter la puissance de 2, ce qui devient beaucoup plus long.

3.3. Le langage StarLisp

3.3.1. Introduction

Le langage *Lisp est une extension de Common Lisp pour la Connection Machine. Au Lisp conventionnel s'ajoute toutes les procédures de communications et autres primitives qui correspondent aux opérations de base de la Connection Machine [26]. Du fait que les ordres *Lisp sont proches de la machine, les programmes s'exécutent plus rapidement que les deux autres langages de haut niveau

(CM-Fortran et C*) avec un facteur d'environ 10%. *Lisp reste un langage fonctionnel, c'est-à-dire que toute expression entre parenthèses est évaluée et donc retourne un résultat.
L'élément parallèle primordial en *Lisp est la pvar (parallel variable) qui ressemble à un grand vecteur Lisp contenant un élément par processeur virtuel. Son contenu peut être soit des nombres entiers, soit des flottants, soit des booléens ou encore tout autre objet Lisp. C'est sur ces objets que s'exécutent les opérations parallèles du langage. Dans le cas d'opérateurs parallèles, le résultat est soit une pvar soit nil.

Sur le même principe que pour les autres langages, la majorité des fonctions Common Lisp séquentielles peut être réalisée en parallèle sur tous les processeurs, exceptées les opérations de traitement de listes, abandonnant ainsi une partie de ses avantages. Les communications ne sont plus intégrées à des procédures comme pour CM-Fortran, et comprennent les deux "schémas" habituels :

- Les communications régulières entre processeurs voisins,
- et quelconque dont le principe est détaillé en 2.

Depuis la version 5.0, les communications sur des grilles de dimension quelconque sont disponibles.

L'environnement *Lisp comprend un debugger et un système de gestion d'erreurs d'exécution, permettant à l'utilisateur de vérifier son application et d'en suivre l'évolution. *Lisp peut être aussi bien compilé qu'interprété. Le compilateur et l'interpréteur étant écrits en Common lisp, ils sont transportables sur toutes machines possédant ce langage (Sun4, station Symbolics, Vax..).

3.3.2. Structure des Pvars

Une pvar est un objet Lisp qui réfère à un champ de la mémoire locale des processeurs de la Connection Machine. Une pvar contient, comme nous l'avons dit dans l'introduction, n'importe quel type d'élément Lisp. En effet, *Lisp comme Common Lisp, n'exige pas de définition explicite du type de la variable. Afin de diminuer le temps d'exécution, il est conseillé de fixer le format des pvars par la fonction : (**pvar** type).

Il est possible de définir des structures à l'aide de la fonction *defstruct. Par exemple, la définition d'une particule, caractérisée par sa vitesse et ses coordonnées dans l'espace, s'écrit de la façon suivante :

```
(*defstruct particule
    (vitesse 1.0 : type (single-float) )
    (coordonnées : type (array (single-float (3) ) ) )
```

Leur allocation mémoire est réalisée selon l'opération à effectuer. Si le résultat de l'appel d'une fonction est une pvar, l'emplacement mémoire réservé est alors dans la pile et pourra être écrasé ultérieurement. On peut dire que le résultat est alors considéré comme temporaire. Sinon, pour conserver le résultat, une affectation dans le tas devra être faite par l'appel à l'extension parallèle d'un set classique du Lisp (*set), et de même pour une affectation dans la pile par l'extension d'un let (*let).

La manipulation des pvars demande à l'utilisateur une certaine attention pour ne pas perdre une information qu'il désire conserver. C'est pourquoi il faut toujours faire attention à ce que retourne la fonction qui est utilisée. Par la suite dans tous les exemples, nous utiliserons la convention de T.M.C. suivante:
si la fonction *Lisp commence par une étoile (*), alors le résultat retourné est à *nil*, ou si l'opérateur est suivi du symbole !!, alors le résultat est une pvar.

3.3.3. La sélection

Si l'on désire ne travailler que sur une partie des données, il faut sélectionner une partie des processeurs virtuels. Certaines fonctions *Lisp permettent de le faire.
Cette sélection est basée sur le résultat de l'évaluation d'une condition sur tous les processeurs, c'est-à-dire d'une pvar booléenne. Si le résultat est nil alors le processeur est éliminé de la sélection courante des processeurs actifs. Les fonctions *when, *if, *cond ne retournent aucune valeur. Elles sont donc utilisées pour leurs effets de bord. Par exemple, pour travailler sur une colonne k d'une matrice associée à une 2-grille, la sélection s'écrit:

(***when** (=!! (**self-address-grid!!** (!! 0)) (!! k))
(corps de l'opération à effectuer))

Comme pour le CM-Fortran, il est possible d'imbriquer différents ordres de sélection les uns à la suite des autres grâce à la sauvegarde du contexte précédent. De même, il est possible de resélectionner la totalité des processeurs lorsque cela est nécessaire. Ainsi, la fonction *all permet d'appliquer un bloc d'instructions à l'ensemble des données sans perdre le contexte initial qui est retrouvé à la fin de l'exécution du bloc.

Remarque 25 : Lors de l'utilisation d'une condition *when, *if ou *cond (et les opérateurs !! correspondants), l'ordre d'exécution de tous les blocs conditionnels est envoyé à tous les processeurs même si aucun d'eux n'est actif. Il faut donc faire attention pour l'écriture de fonctions récursives !
Voir le programme suivant qui calcule la fonction factorielle du contenu de chaque processeur, et qui ne s'arrête jamais:

```
(*defun fact (n)
   (if!! (zerop!! n)
         (!! 1)
         (*!! n (fact (-!! n (!! 1))))))
```

3.3.4. Communications

Les trois types de communications définies dans le chapitre précédent (NEWS, avec combinaisons et générales), peuvent être utilisées en *Lisp. De façon générale, comme pour les opérations arithmétiques sur les pvars, le résultat d'une fonction de communication dépend de l'opérateur utilisé et du caractère (*, !!) qui lui est attaché. Il faut donc toujours vérifier le résultat renvoyé par la procédure de la communication effectuée.

Dans le cas de communication générale (send), il est nécessaire de préciser par un paramètre le comportement en cas de collision de message. On a ainsi la possibilité de combiner les différentes arrivées dans un même processeur. Si l'utilisateur connaît les lieux d'arrivée de chaque message, il peut optimiser le code en indiquant s'il y a ou non collision entre les messages. Par exemple:

```
(*pset add value-pvar dest-pvar cube-address-pvar
   notify-pvar collision-mode)
```

permet de transférer une pvar dans une autre, d'un processeur à un autre (spécifié par sa « cube-address » stockée dans une pvar) en additionnant les entrées dans le même processeur.

Les communications NEWS sur une grille ayant un schéma régulier, ne réclament pas ce genre de paramètres. Seuls les déplacements relatifs le long des axes sont à spécifier. Par exemple:

```
(*news source-pvar dest-pvar (0 1))
```

transfère une pvar dans une autre pour tous les processeurs actifs selon un schéma régulier de communication donné par les coordonnées relatives. Dans l'exemple ci-dessus, la communication se fait avec le processeur 'a droite sur la deuxième dimension d'une 2-grille, qui correspond au 1 sur l'exemple. Si l'on avait pris le couple $(-2\ 0)$, cela correspondrait à un déplacement de 2 processeurs virtuels vers la gauche suivant la première dimension. Notons que c'est l'utilisateur qui fixe ses propres conventions sur les directions (sens et axes), le tout est de rester cohérent avec ce que l'on choisi.

En ce qui concerne les scans, chaque processeur contient l'opération partielle

inclusive ou exclusive sur les éléments précédents, avec une liste de processeurs, segmentée ou non. Ce segment représenté par une pvar de booléens, permet de découper une dimension en sous-séquences. L'exemple ci-après précise les différents paramètres utiles à ces opérations.

(**scan!!** (self-address!!) '+!! :direction :forward
:segment-pvar segment-pvar
:include-self t)

Avec Self-address → 0 1 2 3 4 5 6 7 ...
Segment-pvar → nil nil nil t t nil nil t ...
Résultat sans segment → 0 1 3 6 10 15 21 28
Résultat avec segment → 0 1 3 3 4 9 15 7

En ce qui concerne les communications avec l'hôte, on peut charger ou décharger une pvar dans un tableau de l'hôte. On peut ainsi récupérer des informations sur l'hôte ou envoyer un tableau de données à traiter sur la Connection Machine, ce qui est utile pour traiter des données "réelles". Par exemple pour décharger une pvar sur l'hôte, l'ordre est le suivant :

(**pvar-to-array** source-pvar dest-array)

où dest-array est un tableau de l'hôte de même configuration que celle de la Connection Machine.

3.3.5. Aspect général et conclusion

Avant d'exécuter un programme en *Lisp, il convient d'initialiser la Connection Machine en vidant les mémoires locales des processeurs, et surtout de lui affecter une nouvelle topologie (logique) et un vp-set (ensemble de processeurs virtuels). Cette initialisation est effectuée par :

(***cold-boot** :initial-dimensions géométrie).

Comme nous l'avons mentionné, *Lisp peut être soit compilé soit interprété. Le compilateur étant assez restrictif sur les ordres disponibles, il est conseillé d'utiliser l'interpréteur pour faire tourner le programme. Les temps mesurés sont identiques pour l'utilisation de la Connection Machine proprement dite, seuls les temps d'utilisation de l'hôte diffèrent en raison de la perte de temps lors de l'interprétation de chaque ordre.

Remarque 26 : Pour les prises de temps, il est nécessaire d'inclure des ordres ParIS.

Un simulateur *Lisp est disponible sur différentes machines hôtes. Il permet

de mettre au point les programmes sur de petites tailles, sans s'attacher physiquement à la machine. Il évite ainsi de gêner quelqu'un qui travaille dessus, pendant le travail long et fastidieux de débuggage et de mise au point. Attention, une mémoire importante est nécessaire pour simuler dans un temps convenable l'exécution du programme. Le simulateur entraîne des contraintes majeures sur la programmation : les ordres ParIS sont exclus, ainsi que certaines fonctions *Lisp (et bien sûr, le nombre de processeurs simulés est restreint).

3.4. ParIS

3.4.1. Introduction à ParIS

ParIS est le langage de programmation de plus bas niveau accessible normalement par un utilisateur sur la Connection machine. Il permet de décrire uniquement la partie parallèle du code; la partie séquentielle est écrite dans un langage courant comme le C, ou Lisp [27]. De la même façon, il peut être inclus dans les langages parallèles présentés précédemment (CM-Fortran et *Lisp). Dans les deux cas, la bibliothèque ParIS doit être incluse en tête de programme.
La syntaxe des commandes diffère légèrement selon les langages dans lequel elle est plongée, il est donc nécessaire de se référer à la documentation ParIS pour toute implémentation. Par la suite, nous utiliserons toujours la syntaxe C/ParIS pour les exemples [28].

ParIS ne peut être considéré à la fois ni comme un langage évolué, ni comme un langage machine. Le fait que le contrôle de type et le découpage des expressions mathématiques soient inexistants (c'est-à-dire à gérer par l'utilisateur), ParIS devient un langage non évolué contrairement aux langages parallèles précédents. Puisque d'autre part des opérations comme la gestion de l'occupation mémoire ou celle des processeurs virtuels ne sont pas à faire, il est impossible de comparer ParIS à un assembleur parallèle. C'est plutôt un langage propre à la Connection machine et à sa structure.

Les instructions ParIS sont constituées :

- d'un ensemble de fonctions qui permettent de gérer les VP-set, leurs géométries, ainsi que la mémoire des processeurs virtuels.
- d'un ensemble d'opérations modifiant un champ dest à partir de différents champs source comme par exemple :

Add (dest, source1, source2, len)

où *len* est le nombre de bits des champs source et destination.
Ces opérations sont soit des opérations arithmétiques, soit des communica-

tions, aussi bien sur des entiers (signés ou non) que des flottants.

- d'un petit nombre de constantes du système qui permettent d'écrire des programmes indépendants de la taille de la machine.

3.4.2. Les fonctions ParIS

Le principe de base de la Connection Machine est d'avoir un processeur virtuel par donnée à traiter. Les fonctions ParIS sont là aussi bien pour définir le problème que pour réserver les emplacements mémoire nécessaires pour les variables. Aussi, lorsque la taille réelle du problème est supérieure au nombre physique de processeurs, il faut créer la nouvelle géométrie et indirectement le VP-ratio associé (rapport du nombre de processeurs virtuels sur celui des processeurs physiques). Trois opérations permettent de définir la topologie et la taille du problème sur lequel l'utilisateur va travailler :

geo=**CM_create_geometry** (taille des dimensions, nombre de dimensions);
vp_set=**CM_allocate_vp_set**(geo);
CM_set_vp_set(vp_set);

Il est à remarquer que l'on peut à tout instant changer de topologie ou de taille en invoquant ces fonctions.

Les fonctions ParIS servent également lors de la gestion de la mémoire de chaque processeur virtuel, à créer des champs en leur réservant un emplacement mémoire. Ces champs sont une suite de bits consécutifs, alloués dans la mémoire de chaque processeur. Ils sont caractérisés par un identificateur "var" qui sert à les reconnaître. Toute opération ParIS fait en général appel à ces champs comme paramètres. Ils contiennent les variables parallèles sur lesquelles portent les opérateurs. Il est possible de créer de telles variables sur la pile ou le tas. Leur initialisation est effectuée par une fonction d'allocation (de même, on peut désallouer un champ).

var=**CM_allocate_heap_field**(dim)
CM_deallocate_heap_field(var)

La première de ces deux fonctions réserve un champ de taille "dim" (en bits) dans le tas, alors que la seconde récupère la place de la variable "var" dans le tas. Des fonctions similaires existent aussi pour la pile :

var=**CM_allocate_stack_field**(dim)
CM_deallocate_stack_through(var)

3.4.3. Les opérations en ParIS

ParIS permet un large éventail d'opérations arithmétiques comme pour les langages séquentiels; mais en plus il comprend les opérations de communication (qu'elles soient internes à la Connection Machine ou externes avec l'hôte). Les opérations ParIS sont très lisibles grâce à des codes mnémoniques. Les paramètres d'une opération ParIS sont des champs (qui contiennent des variables parallèles sur lesquelles portent les opérateurs) et éventuellement des variables de l'hôte. Par exemple, on a:

CM_s_add_3_1L(res, source1, source2, len)

qui réalise l'addition de la variable contenue dans le champ source1 et de celle contenue dans source2, puis met le résultat dans le champ résultat res. Le chiffre 3 indique qu'il y a trois champs et 1L qu'ils ne nécessitent qu'un paramètre de l'hôte (la longueur). La lettre "s" spécifie que l'on travaille sur des entiers signés (on dispose aussi d'entiers non signés "u" et de nombre flottant "f". Dans ce cas, il faut, à la place du paramètre len, deux nouveaux paramètres pour fixer la taille de la mantisse et de l'exposant). Pour utiliser l'unité flottante associée à 32 processeurs, il faut fixer les valeurs des mantisses et exposants respectivement à 23 et 8 (format IEEE classique).

Remarque 27 : Il est possible d'effectuer des opérations sur deux champs de longueurs différentes mais de type identique, il suffit de préciser alors les deux longueurs distinctes.

Remarque 28 : Pour les communications de la Connection Machine vers la machine hôte, on ne parle plus de champ résultat. Le paramètre résultat est un tableau de l'hôte qui est rempli avec un élément de chaque processeur par case, ou comme nous l'avons signalé précédemment une variable simple.

Un certain nombre de constantes permettent d'écrire les programmes de façon indépendante de la taille réelle de la Connection Machine. Ces constantes donnent par exemple le nombre maximum de processeurs physiques, ou encore la taille maximum des entiers. En C/ParIs, leur syntaxe est la suivante:

CM_physical_processors_limit,
CM_maximum_integer_length etc..

Ces constantes sont fixées une fois pour toute lors de l'initialisation de la Connection Machine (**Cold_boot** et création de la géométrie).

3.4.4. Gestion explicite du contexte

Contrairement aux deux langages précédemment présentés, le contexte doit être géré explicitement par le programme : il n'existe pas de pile de contexte. Ainsi, si l'on veut travailler momentanément sur un contexte spécifique et retrouver plus tard le contexte initial; il faut au préalable avoir mémorisé ce contexte initial. Ainsi l'instruction :

CM_store_context(dest)

mémorise le contexte actuel dans un champ "dest"; et l'instruction :

CM_load_context(source)

permet de charger le contenu du champ "source" comme contexte.

Pour ne garder comme processeurs actifs que les processeurs vérifiant une certaine condition, il faut dans un premier temps mettre à jour le bit de test en fonction de la condition à vérifier, puis dans un second temps corriger le bit de contexte à partir du bit de test.
Par exemple, l'instruction :

CM_u_eq_1L(a,b,longueur_de_a_et_de_b)

suivie de l'instruction :

CM_logand_context_with_test()

permet d'exécuter la suite du code uniquement sur les processeurs dont les contenus des champs a et b sont égaux et qui étaient déjà actifs au moment du test.

Remarque 29 : Certaines instructions en ParIS sont indépendantes du choix d'un contexte et s'applique à tous les processeurs sans distinction. L'utilisateur doit donc faire attention lors de la programmation de ces instructions.
De même, la plupart des instructions ont une option (always) qui permet de ne pas tenir compte du contexte.

3.4.5. Conclusion

ParIS est un langage qui permet de distinguer parfaitement la partie séquentielle et la partie parallèle du code. Il demande une attention un peu plus soutenue pour l'utilisateur mais en contre partie permet une meilleure programmation, plus proche de l'architecture S.I.M.D. de la Connection Machine. Nous conseillons à tout nouvel utilisateur de se familiariser avec ce langage en utilisant le C/ParIS (qui possède une documentation spéciale et très bien faite). En général, il est possible de programmer en ParIS sur n'importe quel type de frontal et sous tous les langages.

4. CMIS

Les performances obtenues avec les différents langages de haut niveau varient sensiblement de l'un à l'autre. Ces variations dépendent de l'application, mais surtout de la façon avec laquelle elle s'adapte au mieux aux spécificités de la Connection Machine. Pour aider les programmeurs, Thinking Machine Co a développé une bibliothèque de procédures comprenant les éléments de base d'algèbre linéaire, du traitement du signal, etc.: CMSSL[3]. Ces procédures sont écrites à partir d'un nouveau langage, très proche de l'architecture: CMIS (CM Instruction Set). Il permet pour un groupe de 32 PE d'atteindre la puissance de crête en calcul. Mais qu'en est-il pour les communications ? quels schémas peut-on implanter ? quelles performances globales peut-on espérer ? voilà plusieurs questions auxquelles nous avons voulu répondre.
Certains chercheurs utilisent déjà ce langage, en particulier pour des problèmes de communication. Par exemple, Edelman [46] l'emploie lors de la réalisation d'un all-to-all personnalisé sans stockage intermédiaire utilisant des chemins hamiltoniens. Ou Dahl [33] qui l'utilise pour la création d'un outils de placement automatique et d'optimisation des communications: le communication compiler dont nous avons parlé précédemment.

Après une présentation assez large de la programmation en CMIS, nous parlerons plus longuement dans les parties 2 et 3 des avantages qu'apporte CMIS pour la programmation de la Connection Machine. Ensuite, nous discuterons de la programmation spécifique de l'unité flottante avec comme exemple la programmation d'un produit matrice vecteur local à chacune des unités. Les performances obtenues sont proches de la puissance crête. Cette procédure sera utilisée dans le chapitre II lors de la résolution du problème du produit matrice vecteur optimal sur une Connection Machine.

4.1. Présentation générale

CMIS est l'assembleur de la Connection Machine et comme tout assembleur, il demande à l'utilisateur beaucoup d'attention lors de son utilisation. Sa caractéristique première est qu'il "colle" au plus près l'architecture de la machine. Il permet de programmer une à une les différentes composantes d'un module: mémoire, liens, unité flottante, registres, etc.. Une première approche nous permet d'observer deux grandes oppositions entre CMIS et les langages de haut niveau sur la Connection Machine:

- La première concerne la gestion des mémoires locales. La notion de mot mémoire est habituellement associée à l'emplacement physique sur un seul processeur élémentaire (i.e. un mot de 32 bits est géré par un processeur élé-

[3]Connection Machine Scientific Software Library

mentaire). En CMIS, un mot peut également appartenir à un module (i.e. un ensemble de 32 PE). De cette nouvelle vision du module, on peut déduire une nouvelle façon de stocker un mot de 32-bit, et ainsi accélérer les transferts. Le paragraphe 4.3 est entièrement consacrée à ce sujet.

- La deuxième que nous détaillons plus loin, est un des avantages primordiaux de la programmation en CMIS. Les liens de l'hypercube peuvent être utilisés directement, contrairement aux langages de haut niveau. Les communications ne sont plus vues comme des échanges d'informations entre processeurs élémentaires mais comme des échanges ("cube-swap") de mots de 32 bits entre les modules et à travers les liens de l'hypercube.

Une unité flottante (FPU) a été ajoutée à chaque module de 32 PE dans le but d'accélérer les calculs flottants simple précision, ou double précision depuis les dernières versions. Pour l'obtention de performances élevées, la programmation et la gestion des entrées/sorties en CMIS doivent être effectuées de manière minutieuse. Une plus large présentation du FPU et de sa programmation est faîte dans ce chapitre.

Une dernière remarque importante s'impose:

Remarque 30 : Jusqu'à présent un programme parallèle était destiné à l'ordinateur hôte et comportait des instructions pour la Connection Machine. Un code CMIS lui est exécuté directement par le séquenceur. Il communique avec l'ordinateur hôte et pilote les processeurs élémentaires ainsi que les unités flottantes. Un code CMIS doit donc être synchronisé avec le programme principal s'exécutant sur l'ordinateur hôte.

4.2. Le langage CMIS

La programmation en CMIS demande une attention toute particulière en raison de la multitude de choses à gérer: le pipeline de l'unité flottante, l'instruction à effectuer, etc. Une machine séquentielle est constituée d'une unité de contrôle et d'une unité arithmétique et logique (U.A.L); sur la Connection Machine, on a également une unité de contrôle: le séquenceur et des milliers d'U.A.L: les processeurs 1-bit, les transposeurs et les unités flottantes qui font ici office de co-processeurs. Ainsi comme en séquentiel, on va avoir une partie du code pour gérer l'unité de contrôle: IMP, et l'autre pour l'ensembles des unités de calculs: CMIS.

- IMP génère des macro-instructions propres au séquenceur qui permettent de le contrôler ainsi que les registres (pointeurs sur la mémoire, incréments associés à ces pointeurs, gestion des processeurs virtuels, etc..). De plus, il s'occupe des communications avec l'ordinateur hôte (suivant un modèle FIFO), comme par exemple le transfert des données.

- CMIS proprement dit se compose de macro-instructions qui contrôlent les unités arithmétiques et logiques de la Connection Machine (processeurs 1-bit, transposeurs, FPU),

4.2.1. Les macro-instructions *IMP*

Elles servent principalement à décoder l'instruction à effectuer en contrôlant les différents registres, pointeurs et adresses des données en mémoire qui seront les paramètres de l'instruction. Pour cela, on dispose de quelques opérations simples sur les entiers qui permettent l'incrément de pointeurs. Par exemple, c'est grâce aux IMP que l'on peut réaliser la gestion des processeurs virtuel, par incrément sur les adresses mémoire.
La programmation matérielle de l'unité flottante dépend également de ces macro-instructions. Pour réaliser une opération flottante, il est nécessaire de préciser les registres utilisés ainsi que les bus d'accès à l'additionneur et au multiplieur, mais nous y reviendrons dans la section 4.5.
De plus, elles servent d'interface entre les langages de haut niveau et CMIS afin d'insérer une procédure en CMIS dans un code plus général. Une instruction *IMP* transforme un mot défini comme un champs ParIS en une adresse *CMIS* indiquant l'emplacement en mémoire de ce mot.

Pour finir, elles permettent de charger et décharger les différents registres du séquenceur et de l'unité flottante, registres qui sont les variables des commandes *CMIS* . Elles fournissent au programmeur un outil pour suivre l'évolution de son programme en affichant à l'écran le contenu des registres du séquenceur, des transposeurs, ainsi que les registres internes de l'unité flottante. Elles permettent de visualiser l'état physique des processeurs à un instant précis, et ainsi de déterminer les erreurs.

De plus, il est possible de créer des « fonctions » *IMP* dont les paramètres sont des registres. Ces derniers sont, soit préalablement chargés par d'autres instructions *IMP* , soit remplis lors de l'appel de la fonction exécutant une suite d'instructions pour la Connection Machine. Il y a donc possibilité d'écrire un programme « structuré » en CMIS.

4.2.2. Les macro-instructions *CMIS*

Les macro-instructions *CMIS* sont dépendantes des *IMP* . Les registres, chargés par les instructions IMP, sont les variables des instructions *CMIS* . Elles permettent de contrôler les actions physiques devant être exécutées par chaque processeur (PE ou module) et de gérer tous les transferts de mots mémoire qu'ils aillent vers les transposeurs ou le FPU, ou reviennent en mémoire. La gestion des PE est réalisée à partir d'un minimum d'instructions, l'équivalent d'un assembleur séquentiel.
Pour les unités flottantes, *CMIS* déclenche une opération qui place en entrée de

celle-ci les données; le choix de l'opération à effectuer est quant à lui passé en paramètre d'une fonction IMP. Il en est de même pour récupérer un résultat en sortie.
Les communications sont également prises en charge par des instructions par l'intermédiaire des transposeurs *CMIS* (cf 4.4).

Pour finir, les instructions *CMIS* ont un avantage majeur sur les langages de haut niveau : l'adressage indirect. Il s'exécute à l'aide des transposeurs, et permet ainsi de ne plus déplacer les mots de façon identique sur tous les processeurs élémentaires.

4.3. Les différents modes de stockage

Dans tous les autre langages, un mot mémoire est stocké dans un processeur élémentaire (i.e. un mot pour chaque PE). Ce type de stockage, habituel sur toutes les machines, est appelé field-wise par référence au champ d'une variable parallèle où un mot de 32-bit est stocké dans un champ d'un processeur élémentaire.

En CMIS, un mot peut aussi appartenir à un module (comportant une mémoire de mots de 32 bits, soit 32 PE et une unité flottante 32-bits). Dorénavant on considérera un module comme étant un seul processeur en faisant abstraction des processeurs élémentaires. De cette nouvelle vision, on peut déduire une nouvelle méthode pour stocker un mot de 32-bits. Les 32 bits de l'opérande sont distribués aux 32 processeurs élémentaires du module, et non plus à un seul. C'est ce que l'on appelle l'accès slice-wise. La figure A.80 montre clairement les deux différents modes de stockage.

Des instructions permettent de travailler dans les deux modes. Or seul le premier mode est compatible avec les premiers langages de haut niveau. On doit donc l'utiliser si on désire insérer des instructions CMIS dans un code ParIS. Le stockage slice-wise est plus rapide pour effectuer les calculs flottants avec l'utilisation des FPU. En effet, il évite les passages par les registres « transposeurs », et permet de charger directement un mot de 32-bits.

Depuis un an environ, Thinking Machine Co propose aux utilisateurs un nouveau compilateur Fortran90 (ou CM-Fortran) [96]. Il permet grâce à un seul paramètre lors de la compilation de choisir entre le stockage slice-wise et field-wise. Il est évident qu'on n'a pas toujours intérêt à changer de mode de stockage, mais en général, ce nouveau compilateur permet un gain important :

- en vitesse d'exécution comme on pouvait l'espérer en diminuant des accès mémoire remplacés par des accès aux registres.

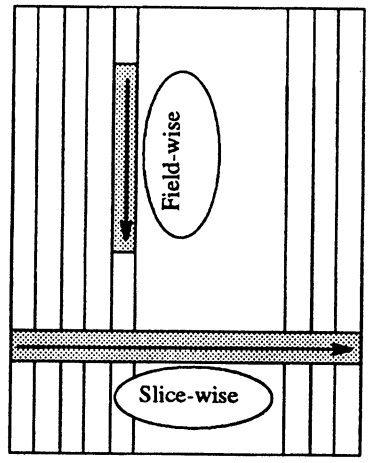


Figure A.80. Les deux modes de stockage dans la mémoire.

- et plus surprenant, en place mémoire par une meilleure utilisation des registres comme variables temporaires.

Prenons deux exemples pour mieux expliquer les choses.
Soit le premier programme :

Programme A.1 : Gain de vitesse

```
Real, array(131072)::a,b,c,d
    a = b + c
    b = b * d

stop
end
```

Les deux figures suivantes présentent le code obtenu pour chacun des stockages. On s'aperçoit que pour le stockage field-wise (cf figure A.81) on fait appel à quatre chargements à partir de la mémoire, deux opérations, et deux déchargements vers la mémoire. Dans le cas du stockage slice-wise (cf figure A.82), l'utilisation des registres associés à l'unité flottante permet de diminuer le nombre de chargements à un seul (on ne compte pas un chargement lorsqu'une opération a un registre en entrée). Cette diminution sensible du nombre total d'instructions permet d'accélérer les performances.

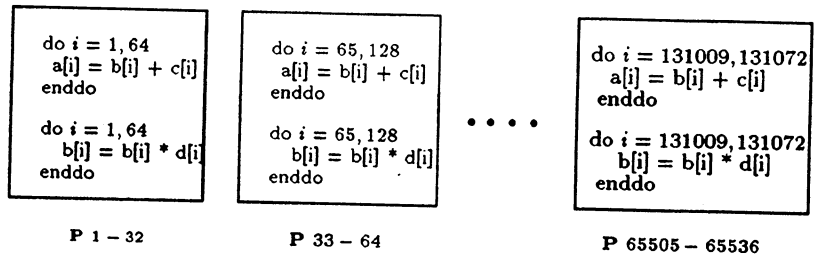


Figure A.81. Version field-wise

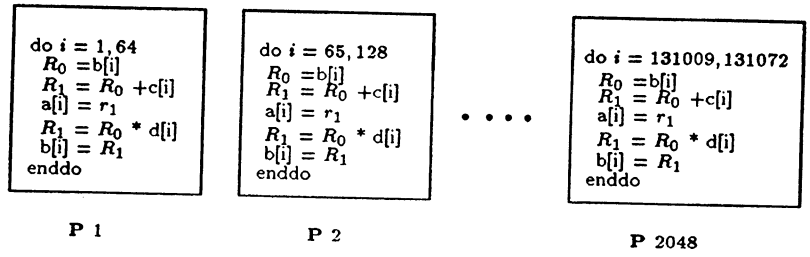


Figure A.82. Version slice-wise

Soit le deuxième programme. Montrons comment gagner de la place mémoire en utilisant encore une fois les registres.

Programme A.2 : Gain de mémoire

```
Real, array(131072)::a,b,c,d
    a = b + c + d

stop
end
```

Le stockage field-wise (cf figure A.83) nécessite la création d'une variable temporaire t pour stocker la première partie du calcul. Un nouveau tableau T de taille 131072 doit donc être créé dans la pile lors de l'exécution.

Pour le stockage slice-wise (cf figure A.84), il suffit de faire appel aux registres temporaires pour éviter d'utiliser de la place mémoire. Le tableau temporaire est stocké dans notre exemple dans le registre R_1.

Remarque 31 : Ici aussi, le nombre de chargements diminue entre les deux stockages. Le deuxième exemple gagne à la fois en vitesse et en place mémoire.

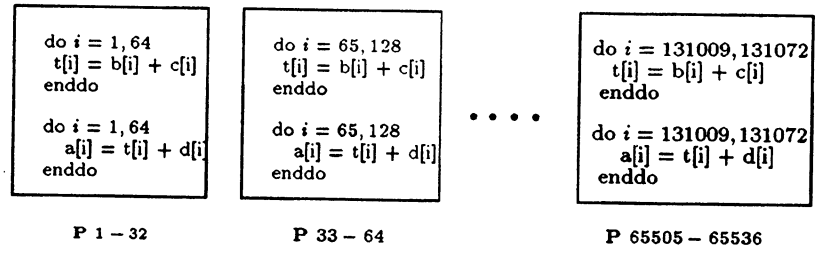


Figure A.83. Version field-wise

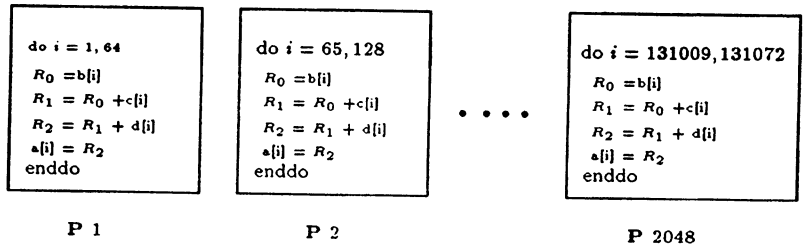


Figure A.84. Version slice-wise

Un des derniers avantages du stockage slice-wise est la possibilité offerte aux utilisateurs de travailler sur des tableaux de petites tailles. En effet lors de la définition de la topologie, il est indispensable avec les langages de haut niveau de travailler avec la totalité des processeurs élémentaires. Par exemple, pour une Connection Machine de taille maximale, le plus petit tableau stocké en field-wise est de taille 256 × 256, alors qu'en slice-wise on peut descendre jusqu'à 64 × 32 car on travaille uniquement avec les processeurs flottants.

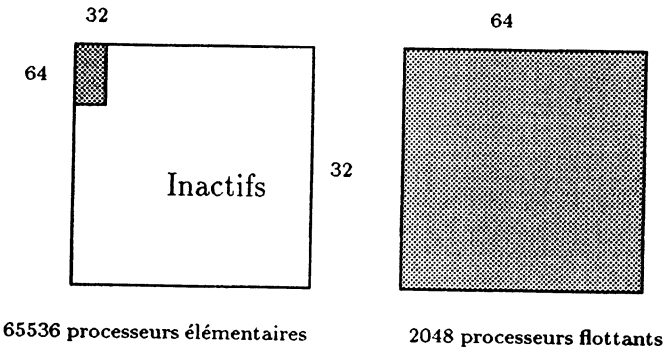


Figure A.85. Différence d'utilisation des processeurs pour un tableau de taille 64 × 32

Dans le cas où le tableau est de taille 64 × 32, la figure A.85 montre que la

surface de processeurs inactifs est très importante.

4.4. Les communications.

Contrairement aux langages de haut niveau qui permettent l'utilisation de différents types de communication (cf A.3), il n'existe qu'une seule communication possible en CMIS : le "cube-swap". Elle consiste en un échange élémentaire de mots de 32 bits entre deux modules voisins. Un module se compose de deux groupes de 16 processeurs plus une unité flottante (cf figure A.52), et possède deux liens bidirectionnels avec chacun de ses voisins. Il peut donc en une seule fois transférer deux mots sur chaque dimension de l'hypercube. Dans la configuration maximale, un module a onze modules voisins et deux liens bidirectionnels le relient à chacun de ses onze voisins. La figure A.86 explique la nouvelle vision de la Connection Machine avec les liens bidirectionnels entre chaque processeur flottant

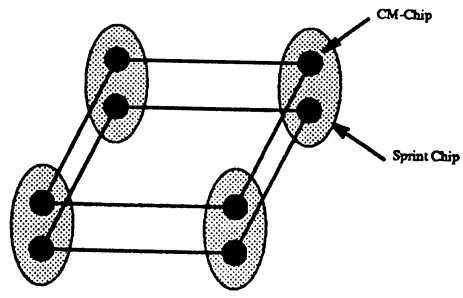


Figure A.86. Nouvelle vision de la CM

Dans la pratique pour réaliser un "cube-swap" en CMIS, il faut utiliser les transposeurs comme buffers de messages. L'instruction de base émet les messages du transposeur A et les reçoit dans le B. La position du message à l'intérieur de celui-ci indique la dimension de l'hypercube suivie par le message, il est donc nécessaire de faire attention lorsque le message est placé ou lu dans le transposeur. Le "cube-swap" utilise la vision slice-wise, il faudra donc utiliser le stockage associé pour être compatible entre les calculs et les communications.

Le temps nécessaire à la réalisation de cette communication est de l'ordre de 26 μsecondes quel que soit le nombre de dimensions utilisées.

Remarque 32 : Il existe une instruction qui utilise le stockage field-wise, mais son utilisation met des fausses informations dans 16 des 32 PE d'un module.

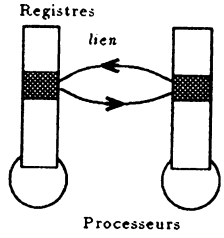


Figure A.87. schéma de base d'un cube-swap.

Remarque 33 : Il est possible de mélanger des programmes écrits en langage de haut-niveau et ceux en CMIS. Il suffit de faire attention aux stockages utilisés et surtout au type de communications choisi.

4.5. L'unité flottante.

L'unité flottante est un Weitek wtl3132 travaillant en simple précision au format IEEE. Il a été associé à un module (i.e. 32 PE) afin d'accélérer les calculs flottants qui prenaient trop de temps en série sur un processeur élémentaire, et sa performance de crête est de 16 Mflops. Il est composé d'un multiplieur et d'un additionneur pipeliné, d'un groupe de 32 registres et d'un autre groupe de 3 registres (cf figure A.88). Sa programmation en CMIS est complexe, l'utilisateur doit d'une part gérer les entrées/sorties des unités arithmétiques, et d'autre part le pipeline des opérateurs. Chaque opération arithmétique est décomposée en trois étapes : le chargement du premier opérande, le chargement du deuxième qui s'enchaîne sur la réalisation de l'opération, et pour finir le traitement du résultat (stockage en mémoire ou dans les registres temporaires).
Chacune d'entre elles se décomposent elles-mêmes en deux instructions : statique et dynamique.

- L'instruction statique fixe quelle opération parmi les trois précédentes va être exécutée,
- tandis que l'instruction dynamique fixe les paramètres d'accès aux opérateurs.

Détaillons ces trois opérations.

- Le premier chargement utilise le groupe des 32 registres pour stocker temporairement le premier opérande qui a une entrée directe sur les unités arithmétiques. On remarque que la taille de 32 registres correspond aux 32 PE ce qui permet de charger le contenu d'un transposeur dans le FPU.
- Le deuxième chargement se fait directement par un bus en déclenchant l'opération arithmétique qui est en paramètre de l'instruction CMIS correspondante.

- La troisième opération utilise soit un bus direct pour écrire le résultat dans un transposeur (field-wise) ou en mémoire (slice-wise), soit dans les 3 registres temporaires ou le groupe de 32 registres pour enchaîner les calculs comme on le démontre ci-dessous.

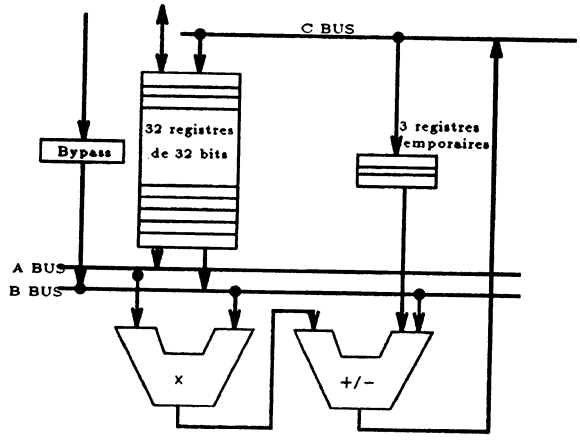


Figure A.88. Schéma d'un FPU

Une des particularités majeures du FPU est de permettre la réalisation d'une opération du type z=a×x+y (AXPY). En effet, la sortie du multiplieur peut être placée en entrée de l'additionneur ce qui permet d'enchaîner les calculs et ainsi d'effectuer des opérations de type mise à jour sur des vecteurs de façon optimale. La taille des registres d'entrée étant limitée à 32, il est évident que si la taille du vecteur à traiter est supérieure, on doit effectuer des chargements et des déchargements supplémentaires qui entraînent un surcoût au niveau des calculs.

En itérant cette opération à l'aide du groupe des 3 registres (appelés registres temporaires) pour stocker le résultat intermédiaire, on peut également écrire un produit scalaire $z=\sum_{i=1}^{j} a[i] \times x[i]$ ou DOT. Le seul problème à résoudre est l'utilisation du pipeline: en effet l'écriture ou la lecture d'un registre demande un top horloge, il faut donc au moins deux tops pour enchaîner les calculs avec l'addition : un top pour écrire le résultat dans le temporaire et un pour le remettre en entrée de l'additionneur.
Or le multiplieur alimente l'additionneur tous les tops (le passage dans une unité, additionneur ou multiplieur, ne demande qu'un top). Il faut donc, soit retarder l'alimentation du pipeline pour avoir une entrée tous les deux tops, soit utiliser un deuxième registre temporaire. Il est évident que la deuxième solution est la plus intéressante car l'alimentation du pipeline est optimale. En fonction du nombre de produits scalaires à effectuer, on se trouve dans l'un de deux cas suivants :

- Dans le cas où on a qu'un seul produit scalaire à effectuer, la meilleure solution consiste à employer deux registres et au lieu d'un. En effet, on peut alors alterner entre les deux registres pour avoir toujours un temporaire en entrée à chaque top. Une dernière addition entre les deux registres permet d'obtenir le résultat final. Comme pour la mise à jour, la taille optimale pour effectuer un produit scalaire est de 32 afin d'éviter les chargements et les déchargements des registres.
- Dans le cas où on a plusieurs produits scalaires à effectuer on a intérêt à essayer de les regrouper deux par deux. En effet, on utilise toujours deux registres temporaires. Chaque registre s'occupe d'un seul produit scalaire, il n'est plus nécessaire d'effectuer la dernière addition entre les deux registres. Le gain ainsi obtenu est de 3 tops horloges. L'inconvénient est que dans ce cas la taille optimale pour un seul produit scalaire est seulement de 16, soit deux fois moins que dans le premier cas.

Lors de la programmation en CMIS du produit matrice vecteur du chapitre II, nous avons dû écrire un produit matrice vecteur local à chaque unité flottante. Nous allons utiliser ce produit comme exemple de programmation en CMIS.

Tout d'abord au niveau de chaque processeur flottant, nous avons choisi d'implanter le produit matrice vecteur ligne, alors que, comme nous le verrons dans le chapitre II une autre méthode plus performante peut être utilisée. Ce choix de la version ligne réside en trois observations.

- Ce produit étant local, il n'est pas nécessaire de faire une méthode hybride mais plutôt d'effectuer les calculs le plus rapidement possible.
- Une relative simplicité de programmation par rapport à la version colonne, même s'il semble d'un premier abord que la taille du problème pouvant être traité de façon optimale paraît inférieure, est un atout important.
- Pour finir il est possible d'atteindre la puissance crête en calcul du processeur flottant. En effet comme nous allons le voir, de la manière dont nous avons programmé le FPU, on a réussi à obtenir le puissance maximale pour un problème de taille 16×16.

Nous allons maintenant détailler la programmation de ce petit produit matrice vecteur $Ax = b$ local de taille optimale. L'algorithme du programme est le suivant :

La programmation du DOT est le seul élément qui demande des explications supplémentaires. Nous allons la découper de façon à comprendre précisément les différentes étapes en utilisant le principe décrit ci dessus qui place deux lignes en entrée pour effectuer deux DOT.

Programme A.3 : version algorithmique

Placer le vecteur x dans les 16 premiers registres de 32 bits.
Pour chaque groupe de deux lignes de la matrice A faire
    DOT entre les registres et les lignes
    Stocker les résultats dans les 16 derniers registres deux par deux

Programme A.4 : DOT

% *On suppose avoir les deux lignes en entrée*
Pour i=1 jusqu'à 16
    % *On s'occupe de la ligne* k
    Multiplier l'élément A_{ki} avec x_i
    Si i=1 alors
        Accumuler avec 0 et le mettre dans le temporaire 1 et dans un des 16 registres restants

    Sinon
        Accumuler avec le temporaire 1 et remettre le résultat dans le temporaire 1 et dans un des 16 registres restants

    % *On passe à la ligne* $k+1$
    Muliplier l'élément A_{k+1i} avec x_i
    Si i=1 alors
        Accumuler avec 0 et le mettre dans le temporaire 2 et dans un des 16 registres restants

    Sinon
        Accumuler avec le temporaire 2 et remettre le résultat dans le temporaire 1 et dans un des 16 registres restants

Le code final ainsi obtenu est très long car il faut décrire les opérations à effectuer pour tous les éléments de la matrice. Heureusement la programmation en CMIS via le Lisp permet l'utilisation de macros qui permettent d'éviter d'écrire chaque ligne en construisant de façon automatique la suite d'instructions qui est très répétitive.

L'utilisation du pipeline est optimale : on sort un résultat tous les tops de l'ad-

ditionneur une fois que le processus est amorcé. Comme on change d'adresse de stockage toutes les 32 instructions, on aura bien au bout des 256 instructions dans les 16 derniers registres les 16 résultats des différents produits scalaires. Les performances obtenues sont de 150 μsecondes pour un produit de taille 16×16, soit 3.3 Mflops (millions d'opérations par seconde) pour chaque processeur flottant. La performance globale sur une 64Kprocesseurs est de 6,77 Gigaflops (milliards d'opérations par seconde). Les résultats sont très proches de la puissance crête de la machine bien que légèrement inférieures car ils tiennent compte du chargement et déchargement du vecteur dans les registres pour effectuer le produit et du vecteur résultat : les 16 produits scalaires.
Pour un problème de taille 32×32, on est obligé de décomposer la matrice en 4 blocs de taille 16×16, puis de faire l'addition des premiers résultats deux par deux pour obtenir la solution finale. Le temps obtenu est de 810 μsecondes, soit plus de 4 fois le précédent. En conclusion, plus la taille augmente plus le fait de regrouper les résultats va faire baisser les performances du produit local.

Pour donner un exemple de programmation en CMIS, voilà le code à générer pour obtenir un produit matrice vecteur local de taille 16×16 sur une unité flottante :

Exemples de programmation en CMIS

```
% Cette fonction utilise la pile d'instructions effectuant le produit 16x16.
% On remarque que l'instruction CMIS dynamique va utiliser l'unite
% flottante (fpu) avec une des entrees qui provient de la memoire
% (mwb qui signifie memory write bypass).
(defun gen-stride2x8-256-instructions ()
  '(cmis-fpu-mwb-stride2x8-frb-dyn-%c4
    256
      ,@(gen-dynamic-instructions)
   )
   )

% Generation d'une pile qui va contenir les 16x16=256 instructions, une pour
% chaque element.
% Une des entrees du multiplieurs est la memoire, l'autre est le vecteur place
% prealablement dans les 32 registres.
% Pour l'additionneur, l'une est la sortie de l'additionneur, l'autre est soit 0
% soit un des temporaires.
(defun gen-dynamic-instructions ()
  (let (result)
    (dotimes (i 8)
      (dotimes (j 16)
```

```
        (push '(imp:wtl3132-dynamic-instruction :io-direction :load
                                                :mult-b-input :bbus
                                                :b-addr ,j          ; vecteur[j]
                                                :alu-b-input ,(if (zerop j)
                                                                  :zero
                                                                  :temp1)
                                                :c-addr ,(+ 16 (* 2 i))
                                                :a-addr ,(+ 16 (* 2 i))
                                                :alu-destination :temp1-and-cbus)
              result)
        (push '(imp:wtl3132-dynamic-instruction :io-direction :load
                                                :mult-b-input :bbus
                                                :b-addr ,j          ; vecteur[j]
                                                :alu-b-input ,(if (zerop j)
                                                                  :zero
                                                                  :temp2)
                                                :c-addr ,(+ 16 1 (* i 2))
                                                :a-addr ,(+ 16 1 (* i 2))
                                                :alu-destination :temp2-and-cbus)
              result)
       ))
    (nreverse result)
  ))

%Creation de l'imp realisant le produit local.
% On remarque:
%       premierement l'affectation des registres
%       deuxiemement l'instruction statique qui nous permet de dire
%       qu'on va effectuer des operations enchainant
%       multiplications et addition.
%       troisiemement l'appel de la fonction dynamique qui declenche
%       le produit.
(defmacro gen-matmult ()
  '(imp:defimp matmult ((source %P1))
      (imp-move-%P1-to-%P2)
      (imp-add-const-to-%P2 16)
      (imp-load-%I1 1)
      (imp-load-%I2 1)
      (imp-load-%A1 17)
      (imp-load-%A2 17)
      (imp-load-%c4 16)

      (cmis-fpu-static
```

```
        (imp:wtl3132-static-instruction :c-port :enable
                                        :wtl3132-function :float-mult-and-add))
        ,(gen-stride2x8-256-instructions)
    (imp-exit)
    )
)
```

4.6. Conclusion

Pour conclure cette présentation générale, CMIS suit de très près l'architecture de la Connection Machine, et permet de l'utiliser de deux façons différentes : slice-wise et field-wise. Son emploi, bien que demandant un gros investissement, est facilité par la possibilité d'écrire des procédures de bases, et ainsi de structurer ces programmes. Il permet une accélération sensible des performances grâce au pipeline du FPU et à l'accès direct aux communications. Mais il reste un défaut majeur : la notion de processeurs virtuels (avantage essentiel des langages de haut niveau) est inexistante au niveau de CMIS. C'est à l'utilisateur lui-même de gérer cette notion pour le placement initial des données et l'incrémentation des pointeurs sur les adresses de la mémoire.

Bibliographie

[1] L. Adams. An M-Step Preconditioned Conjugate Gradient Method for Parallel Computation. Research Report NASA-CR-172150, N83-28940, NASA's Langley Research Center, June 1983.

[2] J.C. Bermond & al. *Ecole Rumeur: Communications dans les réseaux de processeurs.* Ecole d'été Rumeur, August 1992. To transform in book.

[3] E. Anderson. Parallel Implementation of Preconditioned Conjugate Gradient for Solving Sparse Systems of Linear Equations. Technical Report 805, CSRD, University of Illinois, August 1988.

[4] J. M. Arnaudiès and H. Fraysse. *Algèbre, Cours de mathématiques* -1. Bordas, 1987.

[5] S.F. Ashby, T. A. Manteuffel, and P.E. Saylor. A Taxonomy for Conjugate Gradient Methods. *Siam Journal in Numerical Analysis*, 27, 1990.

[6] O Axelsson and Barker. *Finite Element Solution of Boundary Value Problems.* Academic Press, 1984.

[7] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan. Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes. *IEEE Transactions on Computers*, 37:pp 1554–1567, 1988.

[8] C. Berge. *Graphes.* Gautier-Villars, 1983. 3th edition.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation (numerical methods).* Prentice Hall, 1989.

[10] J.Y. Blanc. *Contribution du parallélisme à la résolution d'un problème de répartition de charge dans les réseaux électriques.* Thèse, LMC - Institut Polytechnique de Grenoble, June 1991.

[11] G.E. Blelloch. Scan as Primitive Parallel Operation. *IEEE Transaction on Computers*, 38(11):1526–1538, 1989.

[12] B. M. Boghosian. *Deterministic Cellular Automata with Diffusive Behavior*, volume Cellular Automata and Modeling in Complex Physical Systems, pages 118–129. Springer

Verlag, 1989.

[13] J. P. Brunet, A. Edelman, and J. P. Mesirov. An Optimal Hypercube Direct N-Body Solver on the Connection Machine. In *Supercomputing'90*, 1990.

[14] J.P. Burg. *Maximum Entropy Spectral Analysis.* PhD thesis, Stanford University (CA), 1975.

[15] Calliope. *La parole et son traitement automatique.* 1989.

[16] C. Calvin. Etude et implantations d'algorithmes de produit matrice-vecteur creux sur la Connection Machine, June 1991. Rapport de DEA.

[17] A. T. Chronopoulos and C. W. Gear. Implementation of Preconditionned *s*-Step Conjugate Gradient Methods on a Multiprocessors System with Memory Hierarchy. Research Report UIUCDCS-R-87-1347, University of Illinois, August 1987.

[18] A. T. Chronopoulos and C. W. Gear. Implementation of *s*-Step Methods on Parallel Vector Architectures. Research Report UIUCDCS-R-87-1346, University of Illinois, June 1987.

[19] A. T. Chronopoulos and C. W. Gear. *s*-Step Iterative Methods for Symmetric Linear Systems. Research Report UIUCDCS-R-87-1345, University of Illinois, June 1987.

[20] L. Colombet, L. Desbat, L. Gauthier, F. Menard, Y. Tremolet, and D. Trystram. Real Experiments Using PVM. Submitted to Second International Symposium on High Performance Distributed Computing, July 1993.

[21] L. Colombet, L. Desbat, and F. Menard. Star Modeling on IBM RS6000 Networks Using PVM. Submitted to Parallel CFD'93 Implementations and Results Using Parallel Computers, 1993.

[22] P. Comon and E. Kazamarande. Personnal communication. 1991.

[23] L. Comtet. *Analyse Combinatoire, Volume* 2. Presses Universitaires de France, 1970.

[24] P. Concus, G. Golub, and D.P. O'leary. A Generalized Conjugate Gradient for the Numerical Solution of Elliptic Partial Differential Equations. *Siam Journal in Numerical Analysis*, 1984.

[25] Thinking Machine Corporation. Connection Machine CM-200 Serie. Technical Summary, June 1991.

[26] Thinking Machines Corporation. **Lisp Dictionary*, 1989.

[27] Thinking Machines Corporation. *Parallel Instruction Set*, 1989.

[28] Thinking Machines Corporation. *Programming in C/Paris*, 1989.

[29] Thinking Machines Corporation. *Programming in Fortran*, 1989.

[30] Thinking Machines Corporation. *Connection Machine Scientific Software Library*, 1991.

[31] Thinking Machines Corporation. *Prism User's Guide*, December 1991.

[32] R. Dias da Cunha and T. Hopkins. The Parallel Solution of Triangular Systems of Linear Equations. Technical Report 86, University of Kent at Canterbury, June 1991. Presented

at the 2nd Symposium in High Performance Computing (Montpellier).

[33] E. D. Dahl. Mapping and Compiled Communication on the Connection Machine System. In *The Fifth Distributed Memory Computing Conference (Charleston).* IEEE Computer Society Press, May 1990.

[34] D. Delesalle, L. Desbat, and D. Trystram. Résolution de grands systèmes creux par méthodes itératives parallèles. *Mathematical Modelling and Numerical Analysis*, 1993. To appear.

[35] D. Delesalle, D. Trystram, and D. Wenzek. Tout ce que vous voulez savoir sur la Connection Machine. Technical Report, LMC - IMAG, 1990.

[36] D. Delesalle, D. Trystram, and D. Wenzek. Communications on the Connection Machine. Research Report RR 848-M-, LMC - IMAG, April 1991.

[37] D. Delesalle, D. Trystram, and D. Wenzek. Optimal Total Exchange on a SIMD Distributed-Memory Hypercube. In Q. Stout and M. Wolfe, editors, *The Sixth Distributed Memory Computing Conference Proceedings (Portland)*, pages pp 279–282. IEEE Computer Society Press, April 1991.

[38] O. Delmas. Permutations parallèles "Off-line" de données sur grilles de processeurs. Diplôme DESS ISI informatique., Université de Nice et Sophia-Antipolis, September 1992.

[39] P. Delsarte and Y. Genin. On the Splitting of Classical Algorithm in Linear Prediction Theory. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(5):645–653, May 1987.

[40] J. Demmel, J.J. Dongarra, J. DuCroz, A. Greenbaum, S.J. Hammarling, and D.C. Sorensen. A Project for Developing a Linear Algebra Library for High-Performance Computer. *Aspect of Computation on Asynchronous Parallel Processors*, 1988.

[41] L. Desbat. *Critère de choix des paramètres de régularisation : Application à la déconvolution.* Annexe b : Gradient conjugué et parallélisme, Université Joseph Fourier (Grenoble), 1990.

[42] J. Dongarra et al. A Users' Guide to PVM. Oak Ridge National Laboratory, July 1991.

[43] J.J. Dongarra, I.S. Duff, D.C Sorensen, and H.A. Van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. *Siam*, 1991.

[44] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. LINPACK User's Guide. Siam Philadelphia, 1979.

[45] A. Eberhard, F. Robert, E.H. Snoussi, and M. Tchuente. Factorisation de permutations sur le cube. Research Report 194, IMAG, 1980.

[46] A. Edelman. Optimal Matrix Transposition and Bit-reversal on Hypercube: All-To-All Personalized Communication. *Parallel Distributed Comput.*, pages 328–331, February 1991.

[47] A. Edelman. Large Dense Numerical Linear Algebra in 1993, the Parallel Computing Influence. To appear, following a survey in NA-Digest, December 1992.

[48] G. Fox et al. *Solving Problems on Concurrent Processors: General Techniques and Re-*

gular Problems, volume I. Prentice Hall, 1988.

[49] G.L. Hennigan et al. A Proposed Domain Decomposition Technique for Finite Element on FPS T-Serie. In *Proceedings of* 4^{th} *Conference on Hypercube*, 1989.

[50] G.E Forsythe. On the Asymptotic Directions of the s-Dimensional Optimum Gradient Method. *Numerische Mathematik*, 11:pp 57–76, 1968.

[51] P. Fraigniaud. *Noyaux de communication sur machines à mémoire distribuée.* Thèse, LIP - ENS Lyon, 1990.

[52] P. Fraigniaud, S. Miguet, and Y. Robert. Scattering on a Ring of Processors. *Parallel Computing*, 13:pp 377–383, 1989.

[53] H. Frydlender. *Implantation de réseaux de neurones artificiels sur multi-processeurs à mémoire distribuée.* Thèse, Institut National Polytechnique de Grenoble - I.M.A.G., November 1992.

[54] R. Giles. A Parallel Scalable Approach to Short-Range Molecular Dynamics on the CM5. Technical Report TR-234, Thinking Machine Corporation, 1992.

[55] G.H. Golub and C.F. Van Loan. *Matrix Computation.* North Oxford Academic Publishing, second edition, 1989.

[56] G.H. Golub and G. Meurant. *Résolution numérique des grands systèmes linéaires.* Eyrolles Paris, 1983.

[57] A. Gupta, V. Kumar, and A. Sameh. Performance and Scalability of Conjugate Gradient Methods on Parallel Computer. Technical Report TR 92-64, University of Minnesota, Minneapolis , MN 55455, November 1992.

[58] J. Gustafsson and G. Lindskog. A Preconditioning Technique Based on Element Matrix Factorisations. *Comp. Meth. Appl. Mech. Engng*, 55, 1986.

[59] A.L. Hagemen and D.M. Young. *Applied Iterative Methods.* Academic Press, 1981.

[60] S.W. Hammond and R. Schreiber. Efficient ICCG on a Shared Memory Multiprocessor. *International Journal of High Speed Computing*, 4(1):pp 1–22, March 1992.

[61] M.T. Heath and C.T. Romine. Parallel Solution of Triangular Systems on Distributed Memory Multiprocessors. Technical Report ORNL/TM-10384, Oak Ridge National Laboratory, March 1987.

[62] M. Hestenes and E. Stiefel. Methods of Conjugate Gradient for Solving Linear Systems. *Journal Res. Nat. Bur. Stan.*, 49, 1952.

[63] C.T. Ho, S. L. Johnsson, and A. Edelman. Matrix Multiplication on Hypercubes Using Full Bandwith and Constant Storage. In Q. Stout and M. Wolfe, editors, *The Sixth Distributed Memory Computig Conference Proceeding (Portland)*, pages pp 447–451. IEEE Computer Society Press, April 1991.

[64] O.G. Johnson, C.A. Micchelli, and G. Paul. Polynomial Preconditionnings for Conjugate Gradient Calculations. *SIAM Journal of Numerical Analysis*, 20:pp. 362–376, 1983.

[65] S. L. Johnsson. A Radix-2 FFT on the Connection Machine. Technical Report TR-106 NA89-2, Thinking Machine Corporation, 1989.

[66] S. L. Johnsson and C. T. Ho. Optimum Broadcasting and Personalized Communications in Hypercubes. *IEEE Transactions on Computers*, 38(9):pp 1249–1268, September 1989.

[67] S.L. Johnsson and C.T. Ho. Optimal Communication Channel Utilization for Matrix Transposition and Related Permutations on Boolean Cubes. Research Report RJ 72936, IBM, January 1991.

[68] R. Jones. Protein Sequence Comparison on the Connection Machine CM2, a Collection of Results. Technical Report TR-29 CB90-3, Thinking Machine Corporation, 1990.

[69] R. M. Karp and V. Ramachandran. *Parallel Algorithms for Shared-Memory Machines*, volume A of Handbook of Theoretical Computer Science, chapter 17, pages pp 869–941. Elsevier, 1990.

[70] S.K. Kim and A.T. Chronopoulos. A Class of Lancoz-Like Algorithms Implemented on Parallel Computers. *Parallel Computing*, 17:pp 763–777, 1991.

[71] P. M. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corporation, 1981.

[72] D. Krizanc. A Note on Off-line Permutation Routing on Mesh-Connected Processor Array. *Parallel Processing Letters*, 1(1):pp 67–70, September 1991.

[73] S.Y. Kung and Y.H. Hu. A Highly Concurrent Algorithm and Pipelined Architecture for Solving Tœplitz Systems. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-31(1):66–75, 1983.

[74] E. Landers. Protein Sequence Comparison on a Data Parallel Computer. In *1988 International Conference on Parallel Processing*, pages pp 255–269. Penn State Press, 1988.

[75] P. Lascaux and R. Theodor. *Calcul matriciel appliqué à l'art de l'ingénieur.* 1897.

[76] P. Laurent-Gengoux and D. Trystram. Parallel Conjugate Gradient Algorithm with Local Decomposition. Research report, TIM3 - IMAG, 1988.

[77] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[78] G. Li and T.F. Coleman. A Parallel Triangular Solver for Hypercube Multiprocessor. Technical Report TR 86-787, Department of Computer Science, Cornell University, Ithaca New York, October 1986. Appear in *SIAM Journal of Scientific and Statistical Computing* 9:485:502 1988.

[79] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 10:pp 382–396, 1989.

[80] Z. Liu and J.H. You. An Implementation of Non-Linear Skewing Scheme. *Information Processing Letters*, 1991.

[81] J. Makhoul. Stable and Efficient Lattice Methods for Linear Prediction. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25(5):423–428, October 1977.

[82] O.A. McBryan. Connection Machine Application Performance. Research Report, University of Boulder (Colorado), 1989.

[83] R.B. Mueller-Thuns, D. McFarland, and P. Banerjee. Algorithm-Based Fault Tolerance for Adaptive Least Squares Lattice Filtering on Hypercube Multiprocessor. In *International Conference on Parallel Processing*, 1989.

[84] D. Nassimi and S. Sahni. A Self-Routing Benes Network and Parallel Algorithms. *IEEE Transactions on Computers*, 31(4), 1982.

[85] R. Natarajan and P. Pattnaik. Dense Matrix LU Factorization and Triangular Solvers on a Distributed-Memory Multiprocessor. Research Report RC 17087 (#75817), IBM Research Division, June 1991.

[86] P. Olsson. A Study of Dissipation Operators for the Euler Equations and a Three-Dimensional Channel Flow. Technical Report TR-50 CS89-3, Thinking Machine Corporation, 1989.

[87] B. Plateau and D. Trystram. Optimal Total Exchange on a 3D-Grid. *Information Processing Letters*, 45(2):pp 95 -102, May 1992.

[88] L. Prechelt. Measurements of Maspar MP-1216A Communication Operations. Technical Report, Universität Karlsruhe, November 1992. Draft.

[89] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques*. Masson, 1989.

[90] C.P. Rialan and L.L. Scharf. Fixed-Point Error Analysis of the Lattice and the Schur Algorithms for the Autocorrelation Method of Linear Prediction. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(12):1950–1957, December 1989.

[91] C.H. Romine and J.M. Ortega. Parallel Solution of Triangular Systems of Equations. Technical Report RM-86-05, Department of Applied Mathematics, University of Virginia, 1986.

[92] D. Rover, V. Tsai, Y. Chow, and J. Gustafson. Signal Processing Algorithm on Parallel Architectures : a Performance Update. *Journal of Parallel and Distributed Memory*, 13:237–245, 1991.

[93] Y. Saad. Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method. Research Report YALEU/DC/RR-282, Yale University, 1983.

[94] Y. Saad and M. Schultz. Data Communication in Parallel Architectures. *Parallel Computing*, 11:pp 131–150, 1989.

[95] Y. Saad and M. H. Schultz. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, 37:pp 867–872, 1988.

[96] G. Sabot. Efficiency Hints for Slicewise CM Fortran 1.0. Short paper in Journées du SEH (ETCA), November 1990.

[97] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance Effects of Irregular Communication Patterns on Massively Parallel Multiprocessors. Contractor Report 187514, NASA, 1991.

[98] N.M. Sammur and M.T. Hagan. Mapping Signal Processing Algorithms on Parallel Architectures. *Journal of Parallel and Distributed Memory*, 8:180–185, 1990.

[99] J. Sethian. Interactive Scientific Visualisation and Parallel Display Techniques. Technical Report TR-183 VZ88-1, Thinking Machines Corporation, May 1988.

[100] A. K. Somani and S. B. Choi. On Embedding Permutations in Hypercube. In Q. Stout and M. Wolfe, editors, *The Sixth Distributed Memory Computing Conference (Portland)*, pages 622–629. IEEE Computer Society Press, April 1991.

[101] H. S. Stone. Parallel Processing with Perfect Shuffle. *IEEE Transactions on Computers*, 20(2):pp 153–161, February 1972.

[102] Q. F. Stout and B. Wagar. Intensive Hypercube Communication. Prearrange Communication in Link-Bound Machines. *Journal of Parallel and Distributed Computing*, 10(2):pp 167–181, 1990.

[103] T. Szymanski. On the Permutation Capability of a Circuit-Switched Hypercube. In Silver Spring, editor, *Proc. International Conference on Parallel Processing*, volume 1, pages 103–110. IEEE Computer Society Press, August 1989.

[104] W. Taylor. Renormalization of Lattice Gas Transport Coefficients. Technical Report TR-208, Thinking Machine Corporation, 1991.

[105] J. Tobochnik. Quantum Monte Carlo on a Lattice. Techinal Report TR-246, Thinking Machine Corporation, 1992.

[106] C. Tong. The Preconditioned Conjugate Gradient Method on the Connection Machine. *International Journal of Hight Speed Computers*, 1, 1989.

[107] D. Trystram. Supercalculateur. Polycopié du cours UJF-Ensimag, 1990.

[108] L. W. Tucker. Object Recognition Using the Connection Machine. In *IEEE Conference on Computer Vision and Pattern Recognition*, 1988.

[109] C. Weill-Duflos. Méthode du gradient conjugué avec préconditionnement polynomial pour la résolution de systèmes creux sur la Connection Machine. Rapport de DEA, ETCA, June 1990.

Table des matières

Résumé :

Cette thèse présente les limites du mode S.I.M.D. dans le cadre de la programmation parallèle d'algorithmes d'algèbre linéaire. Plus précisément, celles de la règle d'or du parallélisme massif: « *un élément de la matrice par processeur* », sont développées. Des expérimentations sont effectuées sur une Connection Machine 2.
Néanmoins, la première partie montre comment la création de procédures de communication écrites à partir d'un nouvel algorithme de construction d'arbres équilibrés, et un placement de données judicieux permettent d'atteindre des performances proches de la puissance crête. Mais ce type de travail ne peut pas être effectué sur n'importe quel algorithme, et tout ne s'adapte pas aussi bien.
Dans la deuxième partie, nous présentons les avantages de la décomposition en BLAS pour la construction d'algorithmes massivement parallèles. Elle met, dans le chapitre 4, en évidence la barrière de synchronisation pour la méthode du Gradient Conjugué. Nous proposons dans ce cas particulier comme solution, une ancienne méthode qui bien qu'elle soit, en séquentiel, de convergence plus lente, est plus rapide en parallèle. De plus, la structure des matrices est un facteur important. Elle permet d'accélérer les calculs et d'augmenter la dimension des problèmes à résoudre. L'architecture des machines actuelles en limite encore trop l'utilisation.
La dernière partie est entièrement consacrée aux permutations, et aux communications qu'elles entraînent. Dans la cadre de l'algorithme de Burg, nous proposons une solution qui calcule en même temps à la fois les coefficients de réflexion et ceux d'autorégression sans coût supplémentaire.

Mots Clefs :

Algorithmique parallèle, Algèbre linéaire, S.I.M.D., Connection Machine, Communications structurées

Abstract :

This thesis presents the limits of the S.I.M.D. mode within the frame of linear algebra algorithms using a Connection Machine 2. In particular, those of Data parallelism (« one matrix element per processor »), are developed.
Nevertheless the first part shows how, with new communication routines and a judicious mapping, we reach close to the peak rate performance. But this method does not work for each algorithm.
The second part deals with the BLAS decomposition for massively parallel algorithms. They display the bottleneck in the Conjugate Gradient method. For this case, we propose a solution which consists in an well-kown old method. Indeed, althought its convergence is slower on sequential, its exectution time is better on parallel. Moreover, the matrix structure is an important property. It leads to speed up the computation time and increases the dimension of problem. But the architecture is not yet taking into account of it.
Finally, the last part is devoted to permutations and communications that bring about. In the frame of Burg's algorithm, we describe a solution which computes in the same time, the reflection coefficient and those of autocorrelation.