Contribution du parallélisme à la résolution d'un problème de répartition de charge dans les réseaux électriques

Jean-Yves Blanc

THESE

présentée par

Jean-Yves BLANC

pour obtenir le grade de DOCTEUR

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 23 novembre 1988)

(Spécialité : **Mathématiques Appliquées**)

==========

CONTRIBUTION DU PARALLELISME A LA RESOLUTION D'UN PROBLEME DE REPARTITION DE CHARGE DANS LES RESEAUX ELECTRIQUES

==========

Date de soutenance : 21 juin 1991

Composition du jury :

J. DELLA DORA (président)
B. PHILIPPE (rapporteur)
P. SPITERI (rapporteur)
A. FERREIRA
J. RYCKBOSCH
D. TRYSTRAM

Thèse préparée au sein du Laboratoire de Modélisation et Calcul

Je tiens à exprimer ici mes remerciements aux membres du jury :

A Jean Della Dora pour l'honneur qu'il me fait en présidant ce jury et aussi pour sa grande gentillesse.
A Bernard Philippe et Pierre Spiteri pour avoir accepté de juger ce travail malgré les contraintes temporelles. Qu'ils trouvent ici la marque de ma gratitude pour leurs précieux conseils.
Et surtout à Denis Trystram, mon directeur de recherche favori, pour sa gentillesse, sa compétence et son goût de la recherche. Les trois années passées sous sa direction ô combien éclairée m'ont entre autre permis de percevoir les innombrables difficultés dont est pavée la recherche en algorithmique parallèle : les congrès à l'étranger, la rigidité des horaires, les relations tendues entre chercheurs, le profond respect de la hiérarchie ...
A Afonso Ferreira de m'avoir choisi pour son premier jury.
A Jérome Ryckbosch pour la confiance qu'il a montrée dans le domaine du parallélisme et sans qui ce travail n'aurait sans doute pas existé.

Je tiens à remercier tous les membres du LMC, permanents, thésards et étudiants pour leurs conseils, leurs remarques et tout ce qui fait la remarquable ambiance de ce laboratoire.

Je voudrais tout particulièrement remercier les thésards de Denis pour les trois années que nous avons passées ensemble : François (pour son accent et toutes les choses que nous avons faites tous les deux), Bertrand (pour son extraordinaire gentillesse et nous avoir supportés François et moi), Denis (pour ??), Didier (pour nos discussions sans fin et nos parties de go), Philippe (pour avoir fait semblant de s'intéresser au hockey) et Laurent (pour sa grande méconnaissance de l'Islande et du Mig 29).

Un grand merci également à Yvan (pour son abondante documentation multimédia et ses blagues hilarantes), à Alphonse (le boat-people de mon cœur) et à son sbire, et enfin à Jean-Laurent (pour son indéfectible amitié et tous les goûts que nous avons en commun).

Je voudrais également remercier tous ceux qui ont collaboré à la réalisation matérielle de cette thèse.

Sommaire

Présentation 9

1. Calculs de répartion dans les réseaux électriques à topologie variable 13

1.1. Description du problème de répartition de charge 14
1.1.1. Définition d'un réseau électrique 15
1.1.2. L'approximation du courant continu 16
1.2. Méthodologie de résolution 17
1.2.1. Description de la méthode 17
1.2.2. Exemples de transformations élémentaires 18
1.2.2.1. Déconnection de lignes 19
1.2.2.2. Basculement de lignes 20
1.2.2.3. Basculement d'injections 21
1.2.2.4. Modélisation des couplages 21
1.2.3. Systèmes linéaires successifs obtenus 22
1.3. Développement sur environnement parallèle 23
1.3.1. Introduction à la notion de parallélisme 23
1.3.2. Présentation des différentes formes de parallélisme 24
1.3.3. Puissance des unités de traitement 27
1.3.4. Principe des communications 27
1.3.5. Complexité et performances 29

2. Résolution séquentielle 31

2.1. Méthode de Woodbury 32
2.2. Factorisation de Cholesky modifiée 34
2.2.1. Modification de rang 1 35
2.2.2. Modifications de rang m (1<m<<n) 37
2.3. Mise à jour de la factorisation QR 40
2.4. Gradient conjugué pour la résolution de systèmes successifs 42
2.4.1. Présentation du problème 42
2.4.2. Résolution itérative 43
2.4.3. Description de l'algorithme du gradient conjugué 43
2.4.4. L'algorithme du gradient conjugué pour les systèmes consécutifs 47
2.4.5. Complexité de la méthode 49

2.4.6. Améliorations supplémentaires 49
2.4.7. Expérimentations numériques 50
2.4.7.1. Réinitialisation à une direction 50
2.4.7.2. Réinitialisation à deux directions 52
2.4.7.3. Comparaison avec les méthodes usuelles 54

3. Présentation du FPS T40 57

3.1. Architecture du FPS T40 58
3.1.1. Généralités 58
3.1.2. Structure du FPS T40 59
3.1.3. Environnement logiciel 60
3.1.4. Description d'un nœud 60
3.2. Le Transputer 62
3.2.1. Introduction 62
3.2.2. Architecture d'ensemble 62
3.2.3. Le scheduler 65
3.2.4. Registres et organisation de la mémoire 65
3.2.5. Jeu d'instructions et programmation 66
3.2.6. Performances 66
3.3. Les unités de calcul vectoriel du T40 67
3.3.1. Présentation générale des unités vectorielles du T40 67
3.3.2. Les différents composants du VPU 69
3.3.2.1. La mémoire locale sur chaque nœud 69
3.3.2.2. Les registres vectoriels à décalage 70
3.3.2.3. L'unité de calcul 71
3.3.3. Exemple de programmation du VPU 72
3.3.4. Contraintes de programmation et performances 75
3.3.5. Algorithme de division vectorielle 76
3.4. Les communications 78
3.4.1. Principes généraux 78
3.4.2. Communications entre processeurs 79
3.4.2.1. Description du fonctionnement 79
3.4.2.2. Restrictions d'utilisation 81
3.4.2.3. Exemple de communications 82
3.4.3. Communications externes 83
3.4.3.1. Avec l'ordinateur hôte 83
3.4.3.2. Entre les nœuds systèmes et les processeurs 83

4. Architecture à mémoire distribuée : analyse informatique 85

4.1. Méthodologie 86

4.1.1. Les calculs vectoriels 86
4.1.1.1. Coûts d'une opération élémentaire 86
4.1.1.2. Performances du saxpy 87
4.1.1.3. Performances du produit-scalaire 89
4.1.2. Les communications 92
4.1.2.1. Communications élémentaires 92
4.1.2.2. Communications plus élaborées 94
4.2. Exemple : calcul du produit matrice-vecteur 103
4.2.1. Présentation générale 103
4.2.2. Version produit scalaire 104
4.2.3. Version saxpy 105
4.2.4. Version par blocs 105
4.2.5. Implantation des diverses solutions 106
4.3. Algorithmes de diffusion asynchrone 107
4.3.1. Etat de l'art 107
4.3.1.1. Définitions fondamentales et outils 107
4.3.1.2. Les différents algorithmes de diffusion synchrone 108
4.3.2. La diffusion asynchrone 110
4.3.2.1. La stratégie à 1-paquet 110
4.3.2.2. Les stratégies à q-paquet 114
4.3.2.3. Quelques expérimentations 118
4.3.3. Avantages et inconvénients 119

5. Expérimentations sur le FPS T40 121

5.1. Implantation du gradient conjugué 122
5.1.1. Exposé du problème 122
5.1.2. Implantation locale sur un processeur vectoriel 122
5.1.3. Implantation distribuée sur tous les processeurs 124
5.2. Implantation du gradient conjugué préconditionné pour systèmes consécutifs 127
5.2.1. Résolution vectorielle par la méthode Cholesky 127
5.2.1.1. Vectorisation de la factorisation de Cholesky 127
5.2.1.2. Vectorisation des résolutions triangulaires 128
5.2.1.3. Résultats expérimentaux 129
5.2.2. Le gradient conjugué pour les systèmes successifs 130
5.2.2.1. Les problèmes d'entrées / sorties 131
5.2.2.2. Implantation asynchrone simple 132
5.2.2.3. Implantation synchrone multi-arbres 137

6. Architecture SIMD massivement parallèle : la CM2 141

6.1. Présentation de la Connection Machine 142
6.1.1. Principes de fonctionnement 142

6.1.2. Organisation globale de la CM2 142
6.1.3. Les processeurs élémentaires 144
6.1.4. Les routeurs 145
6.1.5. Les unités flottantes et les transposeurs 146
6.1.6. Les périphériques dédiés 146
6.2. Les communications 147
6.2.1. Présentation des différents types de communications 147
6.2.2. Communications générales 148
6.2.3. Communications de type news 149
6.2.4. Communications avec recombinaison 149
6.2.5. Communications avec l'ordinateur hôte 150
6.2.6. Le communication compiler 151
6.3. Programmation de la CM2 152
6.3.1. Les langages de programmation 152
6.3.1.1. Concepts parallèles 152
6.3.1.2. Le *Lisp 153
6.3.1.3. Le CM-Fortran 155
6.3.1.4. Le C* 156
6.3.1.5. ParIS 157
6.3.1.6. CMIS 158
6.3.2. Exemple de programmation : le produit matrice-vecteur 158
6.3.2.1. Remarques préliminaires 158
6.3.2.2. Description de l'implantation 159
6.3.2.3. Le programme *Lisp 160

7. Expérimentations sur la CM2 163

7.1. Préambule 164
7.2. L'algorithme du gradient conjugué 164
7.2.1 Implantation SIMD classique du gradient conjugué 164
7.2.2. Le gradient conjugué : résultats expérimentaux 168
7.3. La factorisation de Cholesky 170
7.3.1. Implantation de la factorisation de Cholesky 170
7.3.2. Factorisation de Cholesky : résultats expérimentaux 172
7.4. Résolutions de systèmes triangulaires 173
7.4.1. Une première implantation 174
7.4.2. Une version plus sophistiquée 176
7.5. Résolution de systèmes successifs sur la CM2 179
7.5.1. Remarques préliminaires 179
7.5.2. Résultats expérimentaux : la méthode de Woodbury 182

8. Architecture MIMD à topologie reconfigurable : le MégaNode 187

8.1. Présentation du MégaNode 188
8.1.1. Les machines SuperNodes 188
8.1.2. Architecture du MégaNode/128 189
8.1.3. Les différents composants du MégaNode 190
8.2. Environnement logiciel 191
8.2.1. Les environnements de programmation usuels 191
8.2.2. Le langage C de 3L 193
8.2.3. Les contraintes de programmation 193
8.3. Expérimentations sur le MégaNode 195
8.3.1. Choix de la topologie 195
8.3.1.1. Choix d'une stratégie de parallélisation 195
8.3.1.2. Choix de l'arbre ternaire 196
8.3.1.3. Diffusion et regroupement personnalisés dans un arbre ternaire 196
8.3.2. Détermination de la granularité du MégaNode 199
8.3.2.1. Influence du parallélisme sur les communications 199
8.3.2.2. Le produit scalaire 200
8.3.2.3. Le saxpy 201
8.3.2.4. Le produit matrice-vecteur 202
8.3.3. Implantation du gradient conjugué pour systèmes successifs 203
8.3.3.1. Implantation distribuée de la factorisation de Cholesky 203
8.3.3.2. Les résolutions de systèmes triangulaires 205
8.3.3.3. Le gradient conjugué distribué 207
8.3.3.4. Remarques 208

Conclusion 209

Références bibliographiques 213

Présentation

Tout système complexe quel qu'il soit (voiture, avion ou même centrale nucléaire) est susceptible de subir un jour ou l'autre des incidents de fonctionnement. Cela est d'autant plus vrai que le système est complexe et subit des influences extérieures. C'est le cas en particulier des réseaux électriques qui vont nous préoccuper ici. Ils sont de nos jours d'une grande complexité (diverses méthodes de production d'électricité cohabitent, les fils électriques qui portent le courant forment des topologies trés compliquées, les tensions sont différentes suivant les portions du réseau, la consommation est hétérogène suivant les lieux et les heures ...) et sous l'influence de multiples facteurs extérieurs source d'incidents potentiels.
Pourtant, trouver des parades aux incidents qui peuvent se produire est essentiel, aussi bien pour notre sécurité que pour des impératifs économiques. La complexité croissante des réseaux électriques tout autant que l'affinement des techniques mathématiques qui permettent leur modélisation obligent à recourir à des volumes de calcul de plus en plus importants. C'est dans cette optique que se situe ce travail : quelle peut être la contribution du parallélisme à la résolution de problèmes de répartion de charge dans les réseaux électriques ?

L'idée qui a présidée à la rédaction de cette thèse est de montrer comment différents types d'architectures parallèles peuvent modifier l'approche des méthodes numériques spécifiques à la résolution de problèmes de répartion de charge dans les réseaux électriques (méthodes qui dérivent souvent d'algorithmes plus généraux qui sont en général préalablement expérimentés).

Le chapitre 1 présente tout d'abord la modélisation qui permet de formaliser les problèmes de charge dans les réseaux électriques. Puis les systèmes linéaires successifs qui en sont issus sont ensuite décrits : les matrices sont symétriques, pleines et sans structure particulière, les modifications qui permettent de passer d'un système à l'autre sont à priori quelconques, de petite norme mais pas forcément de rang donné. Les matrices qui vont servir aux expérimentations sont aussi décrites. Nous introduisons dans un second temps la notion de parallélisme en informatique, en fournissant les notions spécifiques qui seront discutées lors de la présentation et des expérimentations sur les différentes machines parallèles : classification des architectures parallèles, explications sur la puissances des unités de traitement, principes des communications entre processeurs et définitions portant sur la complexité et les mesures de performances des algorithmes parallèles.

Le chapitre 2 présente tout d'abord différentes méthodes classiques de résolution de systèmes linéaires successifs : méthode de Woodbury, factorisation de Cholesky modifiée (algorithmes de Fletcher & Powell et Bennett). Une méthode itérative originale basée sur un préconditionnement spécifique de l'algorithme du gradient conjugué préconditionné est ensuite présentée, cette méthode est souvent plus rapide en séquentiel que les autres méthodes classiques étudiées précédément et possède en outre des caractéristiques prometteuses quant à sa parallélisation.
Ces méthodes sont ensuite expérimentées sur les différentes machines parallèles, selon leur adaptabilité à telle ou telle forme de parallélisme.
La première machine parallèle étudiée (le FPS T40, un multiprocesseur MIMD vectoriel) est présentée dans le chapitre 3 ; en insistant sur les spécificités de son architecture (unité de calcul vectoriel sur chaque processeur et relative lenteur des communications), et surtout sur les particularités et les contraintes de programmation qui en découlent pour l'utilisateur : difficulté de programmation très efficace des VPU et schémas de communications rigides ou peu performants.
Le chapitre 4 est une analyse informatique des ordinateurs MIMD à mémoire distribuée, présentée dans le cas particulier du FPS T40. On y montre quelles sont les questions à se poser avant d'écrire des programmes sur ce type de machine, et le gain qui peut en découler pour l'utilisateur. Une modélisation fine permet de mettre en avant les coûts respectifs des calculs et des communications. Cette approche est illustrée avec l'exemple du produit matrice-vecteur où les trois formes de cet algorithme sont étudiées (dans le cas particulier du FPS T40, la forme à base de produits scalaires doit être privilégiée). Enfin, un algorithme original de diffusion est présenté. Il s'agit d'une méthode partiellement asynchrone qui se montre très efficace pour des messages de taille courante et qui peut être vue comme une généralisation de la diffusion rotative due à Johnnsson et Ho.
Nous rendons compte dans le chapitre 5 des expérimentations qui ont été effectuées sur le FPS T40. Il apparaît qu'une implantation distribuée du gradient conjugué n'est pas efficace : le grain en n^2 (n étant la taille du problème) de la machine conduisant à une exécution par trop séquentielle de l'algorithme. Par conséquent l'algorithme du gradient conjugué préconditionné pour les systèmes linéaires successifs a été parallélisé avec une stratégie centralisée (chaque processeur se charge de la résolution d'un système). Diverses versions de cette stratégie ont été expérimentées : asynchrones ou multi-arbres, et les plus sophistiquées (où les préconditionnements sont distribués simultanément via plusieurs arbres de recouvrement disjoints) permettent d'obtenir des performances élevées pour ce type de machine.
Le chapitre 6 présente un ordinateur SIMD massivement parallèle : la Connection Machine (CM2). L'architecture originale ainsi que la programmation très spécifique de ce type de machine sont analysées en détail dans l'optique de fournir une base aux expérimentations qui ont été effectuées sur la CM2. Les différents concepts mis en œuvre sur la CM2 sont illustrés par un exemple : le produit matrice-vecteur écrit en langage de haut niveau (*Lisp).
Les expérimentations réalisées sur la CM2 sont décrites dans le chapitre 7. Tous les algorithmes implantés sur la CM2 utilisent un placement des données simimlaire : la CM2

est configurée en 2-grille et les données sont distribuées géographiquement. Il apparaît que la Connection Machine n'est bien adaptée qu'à certains types d'algorithmes qui permettent une utilisation massive de ses ressources comme le gradient conjugué par exemple. La factorisation de Cholesky qui nécessite un masquage des processeurs donne déjà des performances plus faibles. Deux versions de la résolution de systèmes triangulaires ont été implantées (l'une classique où l'on diffuse l'élément qui vient d'être calculée et l'autre plus sophistiquée, inspirée des algorithmes systoliques) et ne donnent pas de bons résultats : le nombre de proceseurs actifs est trop faible sur ce type d'algorithme. Une réponse au problème de calcul de répartition de charge dans les réseaux életriques a été apportée sur la Connection Machine dans le cas particulier où les matrices sont modifiées par des transformations de rang1 : une implantation très efficace de l'algorithme de Woodbury est proposée, la dernière phase de calcul (la plus coûteuse) est effectuée localement, sans communications.

Un second type d'ordinateur MIMD à mémoire distribuée, plus équilibré et à topologie reconfigurable, a ensuite été étudié : le chapitre 8 décrit le MégaNode ainsi que les diverses expérimentations qui ont été réalisées sur cet ordinateur. La présentation du MégaNode insiste sur la reconfigurabilité de cette machine et présente les environnements de programmation disponibles. Une topologie en arbre ternaire a été choisie pour réaliser les expérimentations : afin d'éviter de coûteuses communications liées à une stratégie complétement distribuée, une stratégie "maître-esclave" a été adoptée (un réseau en forme d'arbre convient alors parfaitement), d'autre part les Transputers ont trois canaux physiques de communications.

Un algorithme de diffusion et de regroupement personnalisés spécialement adapté à un arbre ternaire est présenté. La granularité du MégaNode a été déterminée expérimentalement : elle est de n^2 (si le problème est de taille n). Les expérimentations portant sur des matrices de grande taille, tous les noyaux de calculs de l'algorithme de Cholesky (version ligne à base de produits scalaires) ont été distribués (même ceux de grain n), par conséquent l'implantation de l'algorithme de Cholesky ne permet d'obtenir que de très faibles performances. L'implantation de la résolution de systèmes triangulaires est plus adaptée au MégaNode et à la topologie en arbre ternaire. Cependant le gradient conjugué est la méthode de résolution dont l'implantation a donnée sur le MégaNode les meilleurs résultats : les noyaux de calculs en n peuvent être exécutés sur le processeur maître et les noyaux de calculs en n^2 sont distribués à tous les processeurs.

Chapitre 1

Calculs de répartition dans les réseaux électriques à topologie variable

Où la modélisation du problème est introduite et conduit à la présentation des systèmes linéaires successifs, ainsi que des matrices ayant été utilisées pour les tests. Une présentation du parallélisme définit le vocabulaire et les principaux outils qui seront utilisés par la suite. La présentation du problème et la méthodologie de résolution décrites ici sont basées sur des rapports internes à EDF, réalisés au sein de la Direction des Etudes et Recherches, et plus particulièrement par J. Ryckbosch.

Commençons par décrire les problèmes rencontrés par les exploitants qui motivent les calculs de répartition dans les réseaux électriques "à topologie variable".
Tout d'abord, un réseau électrique est composé d'un ensemble complexe et fragile de composants divers : câbles électriques, pylônes, lignes souterraines, disjoncteurs, centres de productions (thermiques, nucléaires ou hydro-électriques) ... Tous ces éléments permettent de produire et transporter du courant électrique depuis les lieux de productions jusqu'aux lieux de consommations (qui en sont souvent fort éloignés).
Pour des raisons économiques évidentes, l'exploitant doit pouvoir répartir au mieux la charge sur les divers éléments du réseau et faire face aussi rapidement que possible aux variations aussi bien de production que de consommation. De plus la complexité et la fragilité des réseaux électriques peuvent amener l'exploitant à envisager pannes et incidents divers. Ainsi, pour réagir à un incident avec rapidité, il peut lui être nécessaire de modifier la topologie du réseau électrique : couper une ligne endommagée, changer l'intensité du courant dans certaines lignes, augmenter la production des centres de production, etc...

1.1. Description du problème de répartition de charge

Des avaries telles qu'une coupure accidentelle de ligne ou une baisse de régime d'une centrale de production peuvent générer des conséquences graves pour les utilisateurs. La conception d'un système informatique permettant à l'exploitant de choisir la topologie adéquate de son réseau face à de tels incidents (la topologie du réseau physique étant bien entendue fixée, mais la prise en compte d'incidents sur certains tronçons amènent à modifier la topologie du réseau en déconnectant une ligne, en basculant une injection ...) est par conséquent un problème de première importance.
EDF a d'ailleurs consenti de gros efforts pour bâtir un tel système (voir par exemple [DuRB] qui décrit un logiciel d'aide à la recherche de parades topologiques). C'est dans ce cadre que se place cette étude.

J.Ryckbosch propose quelques problèmes [Ryc1] qui peuvent se poser lors de la réalisation d'un système de ce type et fournit une approche en trois parties pour trouver la solution.
Les problèmes soulevés par la réalisation d'un tel système sont assez larges car il devra fournir les solutions d'une famille entière de problèmes et répondre à diverses questions telles que :

- à quelle échéance de temps agir ?
- quels devront être les moyens d'actions employés (modifications de la topologie du réseau uniquement ou modulation conjointe de la production) ?
- quels types de manœuvres sont admissibles ?

• quel devra être le niveau de sécurité assuré ?

On peut déjà envisager ces outils informatiques comme agencés en trois unités :

i - une heuristique générant un grand nombre de topologies susceptibles d'être des solutions approchées
ii - un algorithme vérifiant la cohérences des solutions obtenues en i
iii - une série de critères permettant de ne retenir parmi les topologies solutions que les plus proches des pratiques d'exploitation (celles-ci ne pouvant en général pas toutes prises en compte dans la modélisation de l'étape ii)

Une approximation naturelle consiste à adopter pour le point ii un calcul de répartition dans l'hypothèse de l'approximation classique du courant continu (qui sera seule abordée ici), affiné par un examen des transits destiné à détecter d'éventuelles valeurs inadmissibles [Ryc1].

Explicitons maintenant les détails physiques qui vont permettre de comprendre le modèle mathématique qui conduira à une solution numérique du problème.

1.1.1. Définition d'un réseau électrique

Précisons tout d'abord ce que l'on entend par "réseau électrique", en effet bien que semblant quelque peu naïve, la question dépend essentiellement de l'interlocuteur. Et il peut être intéressant de préciser ce qui est derrière la modélisation mathématique du problème, plutôt abstraite.
Pour l'exploitant, un réseau électrique est constitué de lignes, de transformateurs, de disjoncteurs, etc ... Ainsi la définition précise d'un réseau électrique est difficile à formuler, elle est de plus trés variée et même évolutive : elle dépend du type de matériel, de son âge, de son entretien, de son degré de vieillissement ou encore de son emplacement. Les connaissances qui s'y rapportent sont empiriques : tel équipement attire la foudre, tel autre est sensible au gel, etc ...
Par contre, pour un mathématicien, la réponse est bien moins pragmatique, et même très différente : pour pouvoir effectuer des calculs (ce qui semble à priori être la raison d'être d'un mathématicien), il est nécessaire de faire abstraction de la réalité jusqu'à un certain point.

Les définitions qui suivent et le mode de résolution choisi sont extraits de [Ryc1] ; le lecteur y trouvera les détails techniques et les démonstrations des méthodes utilisées dont seuls les résultats sont donnés ici. On peut aussi consulter en complément sur le même sujet [Anon] et [Blan].

Ainsi donc, un réseau électrique peut se définir mathématiquement par la donnée de trois éléments :

- un graphe G = (A,U), où A est un sous-ensemble de IN et U un graphe cartésien sur A, les éléments de A sont les indices des nœuds du réseau électrique et ceux de U les lignes électriques
- X un vecteur de réels strictement positifs, indicés sur l'ensemble U, où x_u est la réactance (c'est à dire l'inverse de la résistance) de la ligne u
- I un vecteur, indicé sur l'ensemble A, dont la somme des composantes i_a (à valeur réelle) est nulle, où i_a est l'injection de courant (quantité de courant émise ou reçue) au sommet a

1.1.2. L'approximation du courant continu

Nous nous plaçons ici dans l'approximation classique dite du courant continu [Ea]. On cherche à calculer le vecteur t des transits de puissance, solution du système linéaire suivant, constitué des deux lois classiques de l'électricité (dites loi des nœuds et loi des mailles) :

$$(1)\begin{cases} S_G\, t = I \\ C_G^t\, X\, t = 0 \end{cases}$$

où l'on définit :

- S_G comme la matrice d'incidence nœud-arc de G (de dimension |A|x|A|)
- C_G comme une matrice cyclomatique de G (elle décrit une base des cycles de G)
- X est abusivement assimilé à une matrice diagonale dont la diagonale est égale au vecteur X lui-même

On peut montrer que l'existence et l'unicité de t sont assurées si le graphe G est connexe (et si $\sum_{a \in A} i_a = 0$). On obtient le même résultat, plus généralement, si chaque composante connexe du graphe G est équilibrée (c'est à dire à bilan production-consommation nul).

On va terminer la modélisation du réseau électrique en précisant la notion de modification du réseau électrique : il faut se donner une transformation T permettant de passer d'un réseau R_1 à un autre réseau R_2.
Prenons comme exemple la déconnexion d'une ligne du réseau. Soit le réseau R1 = $(G^{(1)},X^{(1)},I^{(1)})$, on veut exprimer la déconnexion de la ligne j = (a,b) entre les points a et b. On définit alors T comme suit.
Soit R2 = T(R1). On a alors les relations suivantes :

- $I_k^{(2)} = I_k^{(1)}$ si $k \neq j$

- $X_k^{(2)} = X_k^{(1)}$ si $k \neq j$
- $G^{(2)} = (A^{(1)}, U^{(1)} \setminus \{(a,b)\})$

Par la suite, nous considérerons qu'un changement de topologie quelconque peut se ramener à la combinaison de plusieurs actions élémentaires, ce qui revient à dire que T peut être considéré comme le produit de transformations élémentaires.
Afin de pouvoir simuler des manœuvres effectivement réalisées par les exploitants des réseaux électriques, la liste d'actions élémentaires suivante a été retenue [Ryc1] (elles sont explicitées en 1.2.2) :

- déconnexion d'une ligne (en cas d'accident par exemple)
- basculement d'une ou deux extrémités d'une ligne entre plusieurs sommets (délestage aprés une coupure de ligne par exemple)
- transformation d'un sommet en plusieurs
- basculement d'une production ou d'une consommation entre plusieurs sommets (augmentation de puissance par exemple)

1.2. Méthodologie de résolution

Le but recherché est de pouvoir effectuer les calculs de répartition dans un réseau dont la topologie est susceptible de varier. Cela revient à enchaîner les trois étapes suivantes :

i - définir un ensemble E de manœuvres élémentaires, choisies dans la liste donnée en 1.1.2.
ii - définir un sous-ensemble E' de E (l'ordre n'intervient pas et on suppose que les manœuvres sont effectivement réalisables)
iii - calculer le transit de puissance dans le réseau transformé par E'

1.2.1. Description de la méthode

Une première idée consisterait à résoudre pour chaque topologie le système S(G) défini précédemment. Cette méthode présente cependant un inconvénient important : à chaque nouvelle topologie, le système serait profondément modifié, de façon non triviale, même si G ne subit qu'une modification minime. Les calculs ne seraient pas alors réutilisables d'une fois sur l'autre. Ainsi, pour calculer les transits selon N topologies, il faudrait résoudre N systèmes linéaires de grande taille. Ceci ne serait pas réaliste car en général N est de l'ordre de plusieurs milliers (dans le cas de correction de topologie sur incident par exemple pour un réseau électrique moyenne-tension comme celui de l'Ile de France) [Blan].

Nous allons donc remplacer le système (1) par une autre relation, que l'on notera R'(G') :

$$t = Mu + m_0$$

où
- G' est un graphe qui ne dépend pas de la topologie considérée
- M est une matrice constante qui ne dépend que de G'
- m_0 est un vecteur constant qui ne dépend que de G'
- u est un vecteur de dimension card(E), soit le nombre de manœuvres élémentaires effectuées sur le graphe G

Pour calculer u associé à une série de manœuvres élémentaires, il faut résoudre le système linéaire suivant :

$$Bu = b_0$$

où B et b_0 dépendent de la topologie du réseau et B est de dimension assez petite.

Il reste à expliciter :
- la construction de G' à partir de G et de E, ainsi que des vecteurs X' et I' associés
- la construction des matrices B

Le principe de cette démarche est résumée ci-dessous.

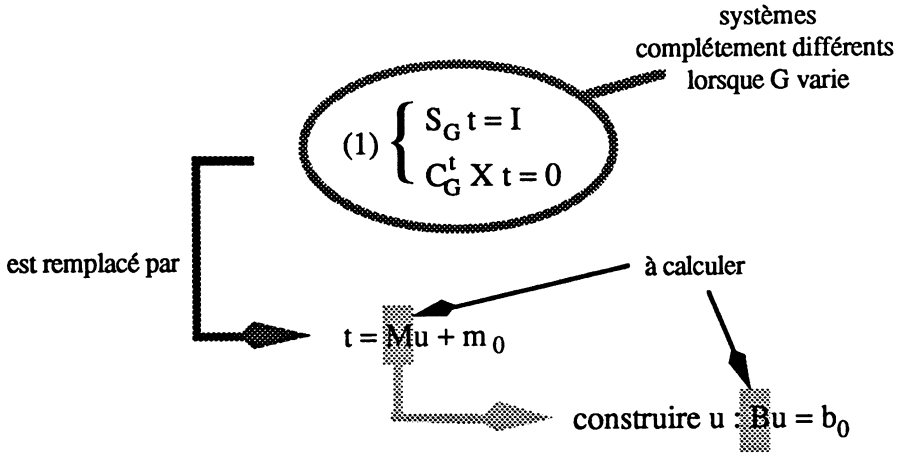


Figure 1 : principe de la méthode de résolution

1.2.2. Exemples de transformations élémentaires

Nous donnons ci-après quelques exemples de manœuvres élémentaires qui interviennent en correction de topologie sur incident. Dans chacun des cas sont fournis les résultats qui permettent d'obtenir les équations qui définnissent les systèmes linéaires modélisant le

problème, les démonstrations peuvent être trouvées dans [Ryck1].

1.2.2.1. Déconnection de lignes

On veut donc calculer les transits sur un réseau R = (G,X,I) alors que certaines lignes sont déconnectées. On pose pour ce faire G = (A,U), avec ici $U = U_1 \cup U_2$ où U_1 est l'ensemble des lignes non modifiées (quelle que soit la topologie envisagée) et U_2 l'ensemble des lignes susceptibles d'être déconnectées.
On peut montrer qu'il existe un vecteur u (indicé sur U_2) tel que :

$$(i)\begin{cases} S\,t = I \\ u_i t_i = 0 \quad \forall i \in U_2 \\ C^t X t = (C^t)^{U_2} u \end{cases}$$

où t est un vecteur indicé sur u, $(C^t)^{U_2}$ désigne la sous matrice de C^t constituée des colonnes d'indices dans U_2.
Posons enfin $U_2 = \hat{U} \cup \tilde{U}$ où $\hat{U}$ représente le sous-ensemble E' (en fait l'ensemble des déconnections effectivement réalisées).
On a donc numériquement : $i \in \hat{U} \Rightarrow t_i = 0$ et $i \in \tilde{U} \Rightarrow u_i = 0$.
On obtient alors :

$t_{U_1 \cup \tilde{u}}$ représente les transits de puissance dans le réseau $\tilde{R}$ obtenu par la déconnection dans R des lignes de $\hat{U}$.

La démonstration utilise les deux techniques classiques en électricité des injections équivalentes et du principe de superposition. Par exemple, la suppression de la branche k du réseau est équivalente à l'injection à ses extrémités (addition à l'une, soustraction à l'autre) d'un courant δ_k tel que le nouveau transit circulant dans la branche k soit justement δ_k, comme indiqué sur la figure 2. La démonstration de ces résultats se trouve dans [Ryc1].

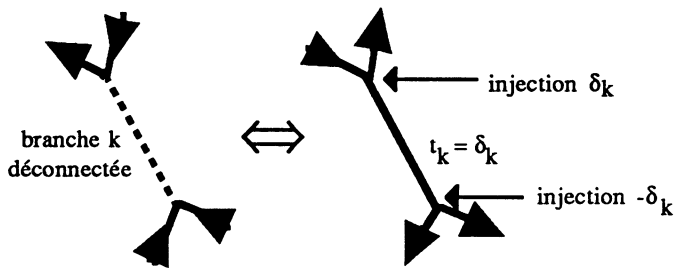


Figure 2 : injection équivalente et superposition

1.2.2.2. Basculement de lignes

Un basculement de lignes peut se définir comme suit : les extrémités de certaines lignes sont susceptibles de basculer d'un sommet à un autre.
Les notations précédentes doivent ici être adaptées : on a encore G=(A,U), mais U se décompose en $U = U_1 \cup U_2 \cup U_3$, où U_1 est l'ensemble des lignes fixes, U_2 l'ensemble des lignes dont une extrémité est basculée et U_3 l'ensemble des lignes dont les deux extrémités sont basculées.
On définit également le réseau enveloppe G' : soit d'une part $u_2 \in U_2$, où u_2=(a,c), mais la première extrémité (par exemple) est susceptible de se trouver en b. On associe donc à u_2 : v_2=(a,c), w_2=(b,c) et $x_{v2}=x_{u2}=x_{w2}$. On pose alors : U'_2 est l'ensemble des lignes v_2 et w_2 obtenues lorsque u_2 parcourt tout U_2 et X'_2 l'ensemble des x_{v2} et x_{w2} obtenus de la même façon.
Soit d'autre part $u_3 \in U_3$, tel que sa première extrémité se trouve dans l'ensemble {a,b} et sa seconde dans {c,d}. On introduit dans G' un nouveau sommet s, et l'on substitue à u_3 quatre nouvelles arêtes : $v_3^{(1)}$=(a,s), $v_3^{(2)}$=(b,s), $v_3^{(3)}$=(s,c) et $v_3^{(4)}$=(s,d). On définit comme précédemment les ensembles : $U_3^{(1)}$, $U_3^{(2)}$, $U_3^{(3)}$, $U_3^{(4)}$ et $X_3^{(1)}$, $X_3^{(2)}$, $X_3^{(3)}$,$X_3^{(4)}$.
La figure ci-dessous illustre ces définitions (les lignes pointillées indiquent les basculements) :

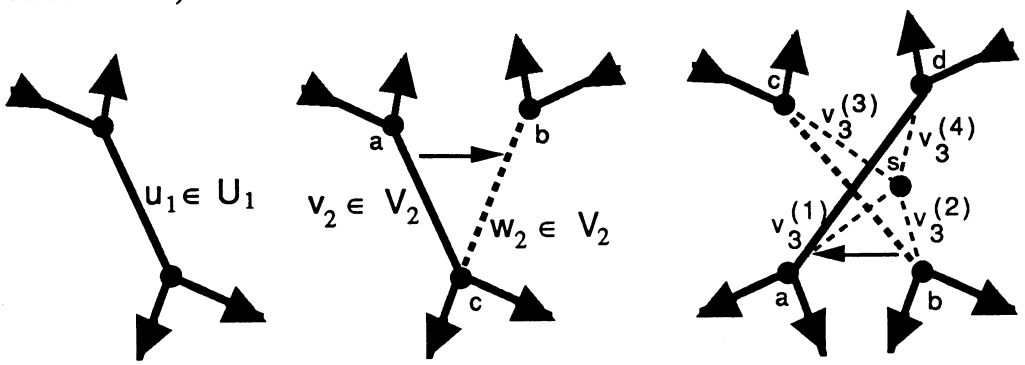


Figure 3 : exemples des 3 types de basculement de lignes

Soit maintenant un vecteur t indicé sur U', il existe un vecteur u (indicé sur $U_2^{(1)} \cup U_3^{(1)}$) qui vérifie les équations classiques) :

$$\begin{cases} S't = I \\ C'^{t}Xt = (C'^{t})^{U'_2 \cup U'_3} u \\ \text{si } u \in U_2 : (t_{v_2}=0 \text{ et } u_{w_2}=0) \text{ ou } (t_{w_2}=0 \text{ et } u_{v_2}=0) \\ \text{si } u \in U_3 : (t_{v_3^{(1)}}=0 \text{ et } u_{v_3^{(2)}}=0) \text{ ou } (t_{v_3^{(2)}}=0 \text{ et } u_{v_3^{(1)}}=0) \text{ ou } (t_{v_3^{(3)}}=0 \text{ et } u_{w_3^{(4)}}=0) \text{ ou } (t_{v_3^{(4)}}=0 \text{ et } u_{v_3^{(3)}}=0) \end{cases}$$

Comme auparavant, il est possible d'exprimer t sous forme linéaire en u (avec les deux premières équations du système précédent) :

$$t=\begin{bmatrix} S' \\ C'^t \end{bmatrix}^{-1}\begin{bmatrix} I \\ 0 \\ \cdots \\ 0 \end{bmatrix}+\begin{bmatrix} S \\ C'^t \end{bmatrix}\begin{bmatrix} 0 \\ C'^t_{U'_2 \cup U'_3} \end{bmatrix} u$$

Montrons maintenant que u est déterminé par un système linéaire : il existe, d'une part pour chaque ligne de U_2, deux variables associées (u_{w_2} et u_{v_2}), ainsi que deux relations linéaires correspondant à la seconde équation du système précédent ; et d'autre part pour chaque ligne de U_3 il existe quatre variables associées : $u_{v_3}^{(1)}$, $u_{v_3}^{(2)}$,$u_{v_3}^{(3)}$,$u_{v_3}^{(4)}$, chaque façon de réaliser les expressions de la quatrième équation du système précédent conduit à 4 équations linéaires. Le nombre d'équations est donc bien égal au nombre de variables, et on obtient finalement un système linéaire : $Bu=t_0$.

1.2.2.3. Basculement d'injections

Cette opération est trés similaire au basculement de lignes et seule la construction du réseau enveloppe diffère légèrement.
L'injection i peut être attachée soit au sommet a, soit au sommet b. On modifie le réseau en lui rajoutant un nouveau sommet s portant l'injection i : s est relié au réseau par les deux nouvelles lignes (a,s) et (b,s). La figure ci-dessous illustre ces notations.

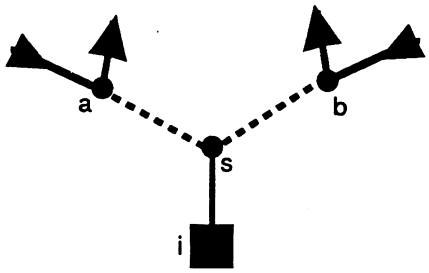


Figure 4 : basculement d'injection

On écrit pour finir une équation similaire à la seconde équation du premier système de la section précédente spécifiant que seule l'une des deux lignes (a,s) ou (b,s) à un transit t nul.

1.2.2.4. Modélisation des couplages

Un réseau électrique comportant des couplages est un réseau dont certains sommets peuvent être divisés en plusieurs nœuds par ouverture ou fermeture de couplages entre barres. Le réseau représenté dans l'exemple ci-dessous présente deux couplages.

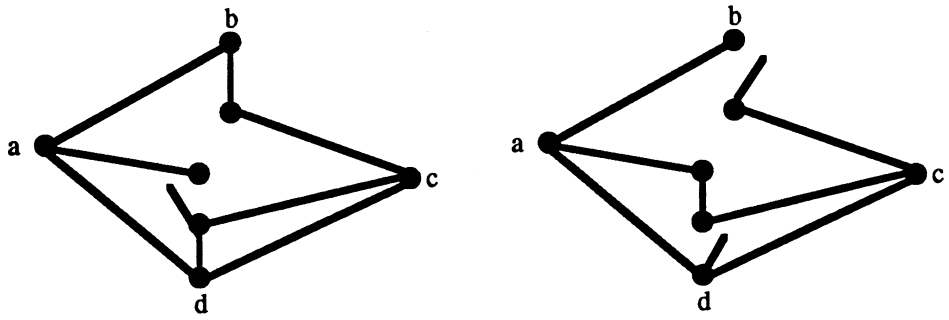


Figure 5 : exemple de réseau comportant des couplages

Dans l'état de gauche, le réseau comporte un seul nœud en b et deux nœuds en d, tandis que dans l'état de droite, le réseau comporte deux nœuds en b en toujours deux nœuds en d, mais avec des connexions différentes.

Le réseau enveloppe est modélisé à l'aide du graphe G'=(A',U'), avec $A'=A_1 \cup A_2$ et $U'=U_1 \cup U_2$, où A_1 est l'ensemble des sommets n'ayant pas de couplages et n'étant exploités qu'en un seul nœud électrique, A_2 est l'ensemble des nœuds électriques associés à des sommets n'appartenant pas à A (cet ensemble est décrit lorsque tous les couplages sont ouverts), U_1 est l'ensemble des lignes du réseau et U_2 est l'ensemble d'arètes (de lignes fictives dont la réactance est quelconque) qui modèlisent les couplages.
Comme précédemment, on suppose qu'il existe un vecteur u tel que (t étant le vecteur des transits de puissance dans toutes les lignes) :

$$\begin{cases} S't = I \\ C'^{t}Xt = (C'^{t})^{U_2}u \end{cases} \qquad \begin{cases} (u_l - x_l t_l)t_l = 0 \\ \forall l \in U_2 \end{cases}$$

Posons $\widehat{U} = \{ l / t_l=0 \}$ et $\widetilde{U} = \{ l / (u_l - x_l t_l = 0) \text{ et } (t_l \neq 0) \}$. On a alors : t_{u_l} représente les transists de puissance dans le réseau lorsque les couplages associés à $\widehat{U}$ sont ouverts et ceux associés à $\widetilde{U}$ sont fermés.

1.2.3. Systèmes linéaires successifs obtenus

En pratique les systèmes linéaires obtenus avec un problème classique de répartition de charge (sur un réseau comme celui de l'Ile de France) ont une taille de l'ordre de 100x100. Cependant à terme, une augmentation importante de la taille de ces matrices est prévisible afin de traiter des réseaux de plus grande taille.
Les matrices générées ne sont que modérément creuses. Cependant aucune conclusion générique ne peut être faite sur le type des matrices qui dépend étroitement du réseau (indirectement certes) et des manœuvres élémentaires appliquées.

La génération des différentes parades topologiques donne un grand nombre de systèmes linéaires successifs : pour des matrices de taille 100x100, on génère de l'ordre de 10000 matrices. Les modifications générées sont à priori quelconques mais de petite norme, pas forcément de rang donné.

Les matrices utilisées pour les expérimentations sont quelconques. Le conditionnement n'influe pas sur les méthodes directes, mais des matrices avec un conditionnement en n ou $\sqrt{n}$ (où n est la taille de la matrice) ont été choisie pour les méthodes à base de gradient conjugué.
Les modifications générées (sur place) sont aléatoires, mais de petite norme ; sauf dans le cas de la méthode de Woodbury expérimentée sur la CM2 (en 7.5.2) où elles sont de rang 1.

1.3. Développement sur environnement parallèle

Nous nous proposons ici de définir les termes de base et les outils qui seront employés par la suite lors de la présentation et des expérimentations sur les diverses machines parallèles.

1.3.1. Introduction à la notion de parallélisme

Les progrès technologiques récents ont permis l'essor considérable des ordinateurs scientifiques. La perspective d'utiliser des méthodes numériques de plus en plus efficaces et rapides s'est transformée en un besoin. Une meilleure adéquation des modèles et de la réalité est recherchée. L'accroissement de la taille des problèmes traités correspond à des exigences de rapidité de calcul de plus en plus grandes.
Cependant, cette tendance se heurte aux limites de la technologie actuelle, et met à l'épreuve les capacités financières de l'utilisateur. L'emploi du parallélisme au sens large est déjà un élément de réponse, tant du point de vue des performances que financier : de nombreux ordinateurs scientifiques parallèles sont commercialisés. De larges perspectives sont ouvertes à la recherche dans le domaine du calcul parallèle.

L'idée du calcul parallèle est simple : dans l'absolu, différentes parties indépendantes d'un algorithme peuvent être résolues simultanément pourvu que l'on dispose de composants de traitement distincts, ce qui permet de gagner du temps. Ce concept de parallélisme est indépendant des contraintes technologiques, et les améliorations apportées dans ce domaine sont compatibles avec les progrès réalisés dans la conception des circuits. Un axe fondamental de la recherche en parallélisme est donc constitué par l'étude d'algorithmes offrant de fortes potentialités d'exécution parallèle.

L'idée du calcul parallèle intervient à divers niveaux d'organisation et n'est pas neuve. En particulier, les architectures utilisant plusieurs unités fonctionnelles de haut niveau, en parallèle, sont une réalité, et ceci depuis déjà une dizaine d'années.
Le premier ordinateur parallèle disponible fut l'ILLIAC IV avec 64 processeurs identiques [Bar], qui était opérationnel au début des années 70 à la NASA. En 1974, CDC livre un STAR-100 (co-processeur pour le traitement des tableaux) au laboratoire Lawrence Livermore. Le premier CRAY apparait en 1976 à Los Alamos. Puis l'université de Carnegie Melon crée le Cmpp. A cette époque, les instructions étaient identiques sur chacun des processeurs, seules les données différaient (un principe qui a été repris depuis notamment sur la Connection Machine, cf chapitre 6). Un peu plus tard, la firme Denelcor commercialise le HEP, premier ordinateur dont les processeurs, au nombre de 16, pouvaient exécuter des instructions différentes (ce qui est le cas des deux autres machines présentées ici : le FPS T40 et le MégaNode).
Le parallélisme s'est considérablement développé ces dernières années et, selon une enquête récente, sera amené à se développer encore. Citons tout d'abord deux microprocesseurs récents qui intéressent de prêt le parallèlisme : le i860 d'Intel intégrant pour la première fois 1 million de transistors sur le chip (qui sert de base à une nouvelle génération d'iPSc) et le T9000, superscalaire, d'Inmos [Inm2]. Plusieurs projets très ambitieux ont pour but d'atteindre d'ici 5 ans des performances de l'ordre du teraflops : par exemple Cray étudie une machine hétérogène YMP-MPP qui combinerait 16 Cray Y-MP et une machine SIMD à parallèlisme massif de construction locale (le projet MPP), Intel développe toute une série de machines (dont la plus ancienne est le iPSC/860) dont la dernière devrait être une machine hétérogène de la classe terraflops incorporant simultanément une technologie de chez Touchstone et le projet à grain fin d'Intel iWARP, IBM a fourni un important soutient financier à Supercomputer Systems (l'entreprise de Steve Chen, le créateur du Cray X-MP) en vue de produire une machine terraflops d'ici 1995, en Europe le projet Genesis a un but similaire [MiSa]. De nombreuses retombées industrielles devraient suivre ces projets ambitieux.
De nombreux articles et travaux montrent le considérable intérêt du parallélisme pour les applications industrielles [Agra], [Fly], [HoJe], [HwBr], [OrVo] ...

1.3.2. Présentation des différentes formes de parallélisme

L'expérience acquise en matière de parallélisme permet de distinguer plusieurs types de parallélisme, suivant le niveau d'abstraction auquel on se place, depuis les opérations sur les bits, jusqu'aux tâches beaucoup plus complexes.
Outre ces différents niveaux de parallélisme, d'autres facteurs contribuent à la diversité des architectures parallèles. Etant donnée la variété des concepts relevant du parallélisme, plusieurs classifications peuvent être utilisées pour distinguer les architectures entre elles. Des analyses tant à propos de la taxonomie des machines, que des distinctions des concepts abstraits de parallélisme sont proposées dans [Agra], [HoJe] et [Alma].

Une première classification des différents types de parallélisme, et certainement la plus connue est la classification de Flynn. Elle est fondée sur le lien entre les instructions et les données. Flynn introduit la notion de flux d'instructions et de données, le premier donnant lieu à une exécution et le second à un traitement. On obtient ainsi trois classes :

- la classe SISD (Single Instruction/Single Data) regroupe les calculateurs séquentiels décrits par le modèle séquentiel classique de Von Neumann.
- la classe SIMD (Single Instruction/Multiple Data) regroupe les calculateurs vectoriels et certains calculateurs systoliques, les array-processors.
- la classe MIMD (Multiple Instruction/Multiple Data) regroupe la plus grande part des calculateurs parallèles.

Cette classification ne permet pas de distinguer des architectures très différentes comme une unité vectorielle et un array-processor.
De nombreux aspects mal pris en compte par la taxonomie de Flynn sont briévement présentés ci-aprés.

Une constatation simple dans le domaine du calcul numérique montre l'intérêt du principe d'anticipation : on peut accroître la rapidité d'exécution des algorithmes numériques en anticipant le chargement et le rangement des données en mémoire, de sorte que les opérations soient effectuées dès que possible, et sans attendre le transfert des opérandes. L'anticipation est utilisée par les architectures "pipeline" (maintenant bien connues).
Les unités du pipeline travaillent simultanément sur les différentes tâches, et plusieurs tâches sont traitées en même temps par le "pipeline". Un tel pipeline introduit donc un parallélisme réel, bien qu'un seul résultat définitif ne sorte à la fois en fin du pipeline.

De nombreuses opérations sur les vecteurs font intervenir des calculs répétitifs sur leurs composantes, et se prêtent donc bien au calcul "pipeline". Beaucoup d'unités de calcul vectoriel font intervenir non pas des sous-unités travaillant en parallèle sur les composantes, comme les "array processors", mais une unité "pipeline". Le "pipeline" est réinitialisé à chaque fois qu'un vecteur lui est soumi. Ce temps d'initialisation est pénalisant, aussi, plus la taille des vecteurs est grande, mieux le "pipeline" est exploité.
Aujourd'hui, il y a très peu de grands ordinateurs qui n'offrent pas d'unité vectorielle "pipeline". Même les architectures qui ne sont pas créées à l'origine pour le calcul vectoriel, telle la Connection Machine, comprennent, au moins en option, des unités vectorielles "pipeline".
Notons que l'emploi d'une unité vectorielle a un fort impact sur la conception des algorithmes numériques. Si l'unité de calcul vectoriel accède directement aux opérandes depuis la mémoire principale, il est généralement suffisant de reformuler les algorithmes en termes d'opérations vectorielles élémentaires (comme un produit scalaire par exemple). D'autres machines sont caractérisées par la présence de registres vectoriels (dont la gamme des célèbres CRAY). Il est alors nécessaire de réutiliser intensivement les données

qui ont été chargées dans ces registres. Cela conduit à une formulation des algorithmes en termes d'opérations matrice-vecteur (produit matrice-vecteur ...). Ce concept est discuté plus en détail dans le chapitre 4.

Signalons une forme de parallélisme voisine de celle du "pipeline" : les réseaux systoliques (un des algorithmes implantées sur la Connection Machine en 7.4.2 s'inspire d'un algorithme conçu pour ce type de parallélisme). Le principe est le suivant : plusieurs unités travaillent en même temps, et plusieurs résultats sont élaborés en même temps. Cependant, les unités de traitement, ou cellules, sont connectées de manière plus complexe que pour un "pipeline", et les données suivent des chemins élaborés. La fonction du réseau est déterminée par la manière dont les données se rencontrent. Les cellules peuvent donc être identiques sans que la fonction soit naïve, et les réseaux systoliques sont soit MIMD soit SIMD. Le terme "systolique" est dû à la régularité avec laquelle les données se déplacent de cellule en cellule, comme les battements du cœur.
Comme pour les "pipeline", les réseaux systoliques effectivement implantés ne comprennent que des cellules dont les fonctionalités sont élémentaires (arithmétique ou mémorisation). Ces réseaux sont développés comme co-processeurs dans divers domaines, dont l'algèbre linéaire, mais demeurent peu répandus.

Un concept important et qui n'est pas pris en compte par la classification de Flynn est la façon dont la mémoire est allouée aux processeurs d'une machine parallèle.
Les ordinateurs parallèles peuvent présenter différentes organisations de leur mémoire. La mémoire peut être constituée de mémoires locales aux unités de traitement, cette organisation est dite à "mémoire distribuée". Les unités de traitement ne pouvant communiquer par l'intermédiaire de zones de données communes, un réseau de connexion entre unités est nécessaire. Si la mémoire n'est pas locale, l'architecture est dite à "mémoire partagée", et toutes les unités de traitements peuvent accéder aux mêmes données.
En première analyse, le mécanisme qui permet que plusieurs unités accèdent à une même mémoire pénalise l'organisation à mémoire partagée. D'un autre coté, les mécanismes de communication peuvent aussi pénaliser les performances. Le type d'organisation de la mémoire intervient au même titre que le type des unités de traitement dans la méthode de résolution d'un problème, et donc dans les capacités globales de l'ordinateur.
Les architectures à mémoire distribuée ou partagée sont toutes deux représentées parmi les ordinateurs actuels. Aucune machine à mémoire partagée n'a été étudiée ici car elles présentent beaucoup moins de spécificité au niveau de la programmation : elles bénéficient d'un savoir-faire préalable en cette matière, alors que les architectures à mémoire distribuée nécessitent un système de communication qui sont encore l'objet de nombreuses recherches. Notons pour terminer que les architectures à mémoire distribuée permettent d'intégrer beaucoup plus d'unités de traitement, et de résoudre des problèmes de plus grande taille.
Le principe des mémoires virtuelles distribuée est actuellement un sujet de recherche

prometteur, qui devrait permettre de simuler un seul espace mémoire sur l'ensemble de la mémoire distribuée sur les processeurs. L'intérêt principal du concept de mémoire virtuelle distribuée est qu'il permettrait de gérer la mémoire sur des machines MIMD à mémoire distribuée exactement de la même façon que sur les ordinateurs séquentiels classiques ou les ordinateurs MIMD à mémoire partagée.

1.3.3. Puissance des unités de traitement

Lorsqu'on veut caractériser la vitesse d'une unité de contrôle, l'unité de mesure utilisée évalue le nombre d'instructions exécutées par seconde, c'est le Mips (Million of Instructions Per Second). En général, il s'agit d'une moyenne pondérée faite sur l'ensemble du jeu d'instructions de l'unité de traitement (et qui dépend aussi de la fréquence du microprocesseur). Une autre unité, le Mflops (Million of FLoating point Operations Per Second), évalue la vitesse de calcul sur les réels. De la même manière, une unité de calcul n'a pas la même efficacité pour toutes les opérations, aussi se réfère-t-on à la notion de performance maximale (*peak rate*), qui est souvent très supérieure à la puissance obtenue par l'utilisateur (voir par exemple les résultats obtenus pour 1 processeur en 8.3.2.2 dans le cas du Transputer). Les unités Mips et Mflops désignent des comportements différents : le Transputer T414, qui sera présenté ultérieurement, exécute 2 Mips, mais il effectue les opérations sur les réels à l'aide de plusieurs instructions, et donc ne réalise que 0,02 Mflops (le T800 qui possède une FPU intégrée est donné pour 1.5 Mflops).

La qualité de la résolution parallèle d'un problème se mesurera à l'adéquation entre la puissance et les compétences (contrôle ou calcul) des éléments matériels utilisés en parallèle d'une part, et la complexité des objets issus de l'analyse de ce problème d'autre part. On ne résoudra pas naturellement un problème se réduisant à une arithmétique bit à bit avec de grosses unités vectorielles. La puissance globale d'un ordinateur parallèle doit donc être rapportée à la fois au problème à résoudre et au type d'algorithme utilisé..
Un ordinateur parallèle est donc caractérisé par la compétence (puissance et aptitude à travailler sur des bits , entiers ou réels ..., comme pour les machines séquentielles), et le nombre de ses éléments.

1.3.4. Principe des communications

Selon l'architecture de l'ordinateur ou le problème considéré, il peut s'avérer indispensable d'interconnecter les unités de traitement entre elles. La majorité des ordinateurs n'est pas consacrée à un emploi spécifique, et la présence d'un réseau suffisamment souple est nécessaire. Retenons quatre configurations classiques :

- le réseau de connexion complet semble à priori idéal, il est utilisé lorsque le nombre d'unités à relier n'est pas trop grand(voir par exemple les connecteurs du MégaNode en 8.3.1). Cependant, à partir d'un certain seuil, la rentabilité tant financière que technique d'un tel réseau se détériore. Aucun routage n'est nécessaire entre deux points à relier.
- certain réseaux, dits reconfigurables, ne permettent pas d'utiliser simultanément toutes les connexions qu'ils permettent : il peut être possible de faire communiquer un nœud avec plusieurs voisins, mais seulement un à la fois. Pour des raisons d'efficacité, les connexions d'un tel réseau sont figées dans une configuration donnée. C'est le cas du MégaNode présenté au chapitre 8.
- le réseau de connexion hypercubique est très répandu car il offre un bon compromis entre la distance entre deux points à relier et le nombre total de connexions du réseau [HoJe]. Les communication entre deux nœuds non directement voisins passent par des nœuds intermédiaires. Le FPS T40 (présenté au chapitre 3) entre dans cette catégorie.
- la grille (carrée ou hexagonale) est d'un coût faible. Elle constitue souvent un des niveaux intermédiaires de communication dans un réseau mixte, où les nœuds d'un hypercube peuvent être en fait eux-mêmes une grille par exemple. Un routage est nécessaire ([TuRo]).

Dans le cas des architectures à mémoire partagée, le type de connexion entre les unités de traitement et cette mémoire est important dans le choix des algorithmes. Cette influence est identique à celle qu'ont certains types de mémoire sur les ordinateurs séquentiels.

Indépendamment du réseau d'interconnexion, deux caractéristiques particulières du système de communication influent sur la conception des algorithmes que l'on désire implanter.
Tout d'abord le synchronisme ou l'asynchronisme des communications entre processeurs. Les communications synchrones suivent le principe du rendez-vous. Elles sont bien adaptées aux problèmes engendrant des communications régulières. Elles permettent d'effectuer facilement des synchronisations. A l'opposé, les communications asynchrones se révèlent plus utiles pour prendre en compte des événements imprévus intervenant sur une unité particulière. Par l'intermédiaire d'une mémoire partagée, les communications s'établissent naturellement sur le mode asynchrone. Les rendez-vous s'implantent facilement avec des communications asynchrones, avec un faible surcoût. Ce n'est pas le cas de l'inverse. D'une manière générale, les communications asynchrones offrent plus de souplesse et sont plus efficaces.
Le temps de communications entre processeurs est une notion essentielle à prendre en compte lors de la conception d'algorithmes pour des machines parallèles. Sauf pour un réseau complet, deux unités quelconques ne sont pas nécessairement directement voisines. Il s'en suit que les vitesses de communication entre unités peuvent ne pas être uniformes. Certains constructeurs établissent un mécanisme de routage tel que cette

vitesse soit constante quelle que soit la paire d'unités à relier. D'autres ont choisi l'option inverse (c'est le cas de la Connection Machine). C'est là un autre aspect important dans le choix d'un algorithme. Un message d'une unité à une autre peut être transmis en un seul bloc ou en plusieurs paquets. A chaque paquet correspond un temps de préparation. Le temps de transmission d'un message fait donc intervenir un terme proportionnel au nombre de paquets, et un terme proportionnel à la taille du message. C'est le modèle le plus généralement utilisé pour modéliser les temps de communication. Ce modèle est celui choisi pour modéliser les communications lors de l'analyse du FPS T40 au chapitre 4. Il ne prend pas en compte, lorsqu'il y a routage, les problèmes d'engorgement.

1.3.5. Complexité et performances

On mesure les performances d'un algorithme parallèle par son facteur d'accélération S_p (défini comme le rapport du temps d'exécution de l'algorithme séquentiel sur le temps obtenu en utilisant p processeurs), il mesure le gain en temps du au parallélisme :

$$S_p = \frac{T_1}{T_p}$$

Ce facteur a une réelle signification pour les machines dont l'architecture comprend une mémoire partagée. Elles n'autorisent en pratique qu'un nombre relativement faible de processeurs. Dans le cas d'architectures plus massivement parallèles, nous verrons comment modifier le facteur d'accélération pour analyser les performances d'un algorithme.

Tout programme numérique possède une partie intrinsèquement séquentielle qui ne pourra pas être parallélisée (dû à des problèmes de précédence entre les différentes instructions). Notons t_s, le temps correspondant à cette partie séquentielle, et t_p le temps correspondant aux parties parallélisables du programme. Le temps total est alors t_s+t_p. La loi d'Amdahl dit que le facteur d'accélération maximal qui peut être obtenu est :

$$S_p = \frac{t_s+t_p}{t_s+\frac{t_p}{p}}$$

Cette approche est raisonnable pour les machines parallèles classiques, cependant, sur des machines massivement parallèles à mémoire distribuée (dont les communications entre processeurs sont beaucoup plus coûteuses), cette approche est insuffisante. En effet, le facteur d'accélération ainsi défini n'est en général pas très bon, et ne rend pas compte du fait que la taille des problèmes traités peut être beaucoup plus importante.
On peut alors définir un nouveau facteur d'accélération par rapport au plus grand problème possible sur l'ensemble des processeurs [Gust]. Par exemple, on calcule t_p-

t_{comm} qui correspond au temps de calcul total sur p processeurs et pt_1 qui est l'extrapolation du temps séquentiel à une machine permettant de résoudre le plus grand problème. Le facteur d'accélération est alors :

$$S_p = \frac{t_p - t_{comm}}{pt_1}$$

L'efficacité E_p est une notion complémentaire au facteur d'accélération, elle est définie comme le rapport de S_p sur p (et mesure en fait le taux d'occupation moyen des processeurs). E_p est comprise entre 0 et 1, et plus elle est proche de 1, plus le programme sera parallèle.

Chapitre 2

Résolution séquentielle

Où l'on introduit les méthodes classiques de résolutions de systèmes linéaires successifs dans l'optique de leur parallélisation. Une méthode itérative originale basée sur le gradient conjugué avec un préconditionnement spécial est ensuite présentée.

2.1. Méthode de Woodbury

Une méthode peu satisfaisante pour résoudre un système linéaire consisterait à inverser la matrice qui en est issue. Cependant, lorsqu'on veut résoudre une succession de systèmes variant peu les uns par rapport aux autres, il peut être intéressant de connaître explicitement l'inverse de la matrice issue du premier système. En effet, on peut ensuite utiliser la formule de Woodbury pour calculer les inverses des matrices issues des systèmes linéaires suivants.

Rappelons ce qu'est une modification de rang m : soit une matrice A de taille nxn, on dit que A est modifiée par une modification de rang m lorsque l'on a :

$$A = A + ZCZ^t$$

où Z est une matrice nxm et C une matrice mxm. Dans le cas d'une modification de rang 1, A peut s'écrire :

$$A = A + zcz^t$$

où z est un vecteur de taille n et c un scalaire.

La formule de Woodbury permet de calculer l'inverse d'une somme de matrices à partir des matrices et de leurs inverses, en particulier de calculer l'inverse d'une matrice ayant subie des modifications de rang m [GoVL].

Formule de Sherman-Morrison-Woodbury : soit A une matrice régulière de taille nxn, X et Y deux matrices de taille nxm (avec m<n), et C une matrice régulière de dimension m. Si la matrice $C^{-1}+Y^tA^{-1}X$ (d'ordre m) est régulière, alors la matrice $B = A+XCY^t$ est aussi régulière et son inverse est :

$$(A+XCY^t)^{-1} = A^{-1}-A^{-1}X(C^{-1}+Y^tA^{-1}X)^{-1}Y^tA$$

La démonstration est basée sur la relation :

$$(I_n + AB^t)^{-1} = I_n - A(I_m+B^tA)^{-1}B^t$$

Nous obtenons alors en développant :

$$(A+XCY^t)^{-1} = (A(I_n + A^{-1}(XCY^t))^{-1} = (I_n+A^{-1}XCY^t)^{-1}A^{-1}$$

Puis :

$$\begin{aligned}(A+XCY^t)^{-1} &= (I_n - A^{-1}X(I_m +CY^tA^{-1}X)^{-1}CY^t)A^{-1} \\ &= A^{-1} - A^{-1}X(I_m +CY^tA^{-1}X)^{-1}CY^tA^{-1}\end{aligned}$$

Nous pouvons finalement en déduire :

$$(A+XCY^t)^{-1} = A^{-1} - A^{-1}X(C^{-1} +Y^tA^{-1}X)^{-1}Y^tA^{-1}$$

L'intérêt de ce résultat est qu'il permet d'obtenir l'inverse d'une matrice en $O(mn^2)$ opérations arithmétiques. Rappelons que le calcul de l'inverse d'une matrice requiert dans le cas général $O(n^3)$ opérations. Cette méthode représente donc un gain certain, dès que les matrices sont de grande taille (et que m n'est pas trop grand) [GoVL].

L'algorithme de la méthode de Woodbury est donné ci-après dans le cas d'une modification de rang 1 (qui est le cas où la formule est la plus utilisée). L'inverse de la matrice B s'écrit alors :

$$B^{-1} = (A+czz^t)^{-1} = A^{-1} - \frac{(cA^{-1}zz^tA^{-1})}{1+cz^tA^{-1}z}$$

Le calcul de B^{-1} peut alors se décomposer de la façon suivante (ce calcul n'est pas optimal en séquentiel, mais il a été retenu car cette méthode permet de trouver une implantation trés intéressante en parallèle, voir 7.5.2) :

- calcul du vecteur $x = A^{-1}z$
- calcul du scalaire $s = 1+cz^t.x$
- calcul de la matrice $M = x*x^t$
- calcul du résultat final : $B^{-1} = A^{-1}-(c/s)M$

Notons qu'un algorithme plus efficace en séquentiel (c'est à dire nécessitant moins d'opérations arithmétiques) consiste à ne pas calculer explicitement l'inverse, mais à résoudre le système linéaire. On veut donc résoudre $Bu=b_0$, et l'on procéde comme suit :

- $x = A^{-1}z$
- $u = A^{-1}b_0$
- $\alpha = c(z^tu)$
- $u = u - \alpha x$
- $\beta = \frac{1}{1+cz^tx}$
- $u = \beta u$

La complexité de cette méthode est donc en $2n^2$. Elle permet donc d'obtenir un gain de 5/2 par rapport à la première méthode décrite ci-dessus (le coût de cette méthode est analysé ci-après).
De plus lorsque plus de n systèmes linéaires (ce qui est en pratique le cas, voir les chiffres cités en exemple en 1.2.3), le calcul de B^{-1} est plus efficace.

L'algorithme peut ainsi s'écrire (en pseudo-code) :

```
{ calcul du vecteur x }
pour i et j variant de 1 à n
      x[i]=x[i]+Amoinsl[i,j]*z[j]
{ calcul du scalaire s }
pour i variant de 1 à n
```

```
      s=s+z[i]*x[i]
s=1+c*s
{ calcul du résultat final }
s=c/s
pour i variant de 1 à n
       t=x[i]*s
      pour j variant de 1 à n-1
          Bmoins1[i,j]=Amoins1[i,j]-t*x[j]
```

Le coût en nombre d'opérations arithmétiques de la version précédente de l'algorithme de Woodbury est le suivant :

- le calcul du vecteur x nécessite n(2n-1) opérations
- le calcul du scalaire s nécessite 2n+1 opérations
- le calcul de la matrice M nécessite n^2 opérations
- le calcul du résultat final nécessite $2n^2$ opérations

L'algorithme coûte donc au total $5n^2+n+1$ opérations arithmétiques, alors qu'une résolution par inversion basée sur la méthode de Gauss nécessite $\frac{8n^3}{3}+O(n^2)$ opérations. Le gain est donc considérable dès que les tailles des matrices sont élevées.
Notons que cet algorithme nécessite $3n^2+n+1$ opérations triadiques "multiplication suivie d'une addition".

2.2. Factorisation de Cholesky modifiée

Toute matrice A symétrique définie positive peut se décomposer en un produit de trois matrices par l'algorithme de Crout (variante de Cholesky) :

$$A = L D L^t$$

où L est une matrice triangulaire inférieure à diagonale unité et D une matrice diagonale à coefficients positifs.
Supposons que la matrice A soit modifiée par une modification symétrique de rang m (décrite en 2.1) : $A = A + ZCZ^t$.

Il existe plusieurs algorithmes pour obtenir les facteurs correspondants à la décomposition LDL^t de la matrice A modifiée [Kaza]. Distinguons les cas des modifications de rang 1 et de rang m>1.

2.2.1. Modifications de rang 1

Dans le cas d'une modification de rang 1, la matrice A modifiée peut s'écrire :

$$A = A + czz^t$$

où c est un scalaire et z un vecteur de taille n.
La méthode présentée, due à Fletcher & Powell [FlPo], est basée sur une décomposition directe de la matrice modifiée, elle constitue un cas particulier de l'algorithme de Bennett pour les modifications de rang m>1. Une autre méthode (de Gill & Murray [GiMu]) consisterait à transformer la matrice modifiée B^{-1} en une matrice dont la factorisation se fait facilement.

La méthode est basée sur la décomposition de la matrice A sous la forme :

$$A = LDL^t = \sum_{i=1}^{n} d_i l_i l_i^t$$

où l_i est la i^{eme} colonne de la matrice L. Ainsi, la matrice $B = A+czz^t$ peut s'écrire :

$$B = \tilde{L}\tilde{D}\tilde{L}^t = \sum_{i=1}^{n} \tilde{d}_i \tilde{l}_i \tilde{l}_i^t$$

Supposons maintenant que $d_1 l_1 l_1^t + czz^t$ puisse s'écrire :

$$d_1 l_1 l_1^t + czz^t = pxx^t + qyy^t$$

où les vecteurs x et y (de taille n) sont des combinaisons linéaires de l_1 et de z tels que $x[1]=1$ et $y[1]=0$. On peut prendre par exemple $y = z - z[1]l_1$ et $x = l_1 + \beta y$.
En explicitant l'équation précédente, on obtient :

$$p = d_1 + cz[1]^2$$
$$\beta = \frac{cz[1]}{p}$$
$$q = c - p\beta^2$$

Comme y[1]=0, pxx^t peut s'identifier à $\tilde{d}_1\tilde{l}_1\tilde{l}_1^t$. On obtient donc d_1 ainsi que la première colonne du facteur modifié de L. Le problème revient alors à mettre à jour $\sum_{i=2}^{n} d_i l_i l_i^t$ avec une modification qyy^t de rang 1. Ces deux matrices ayant les premières lignes et colonnes nulles, le problème est donc réduit à n-1 variables. On peut ainsi itérer le procédé jusqu'à obtention de la décomposition entière de la matrice modifiée.

Soit $c_j z^{(j)} z^{(j)t}$ le reste de rang 1 quand $\tilde{d}_1\tilde{l}_1\tilde{l}_1^t$ est déterminé avec $c_1=c$, $z^{(1)}=z$ et $z_i^{(j)}=0$ pour i<j. L'itération de base s'exprime alors par la relation :

$$\tilde{d}_j\tilde{l}_j\tilde{l}_j^t + c_j z^{(j)} z^{(j)t} = \tilde{d}_j\tilde{l}_j\tilde{l}_j^t + c_{j+1} z^{(j+1)} z^{(j+1)t}$$

avec :

$$\tilde{d}_j = d_j + c_j z_j^{(j)2}$$
$$\beta_j = \frac{c_j z_j^{(j)}}{\tilde{d}_j}$$
$$z^{(j+1)} = z^{(j)} - z_j^{(j)} l_j$$
$$\tilde{l}_j = l_j + \beta_j z^{(j+1)}$$
$$c_{j+1} = c_j - \tilde{d}_j \beta_j^2$$

où d_j, β_j et c_{j+1} jouent le rôle respectivement de p, β et q dans les équations précédentes à chaque étape j.
En utilisant les expressions de $z^{(j+1)}$ et β_j, $\tilde{l}_j$ peut s'exprimer sous la forme :

$$\tilde{l}_j = l_j + \beta_j (z^{(j)} - z_j^{(j)} l_j)$$
$$\tilde{l}_j = (1 - \beta_j v_j) l_j + \beta_j z^{(j)}$$
$$\tilde{l}_j = \frac{d_j}{\tilde{d}_j} l_j + \beta_j z^{(j)}$$

Notons que $\tilde{l}_j$ peut être calculé simultanément avec $z^{(j+1)}$ et c_{j+1} :

$$c_{j+1} = \frac{c_j d_j}{\tilde{d}_j}$$

On peut donc finalement écrire l'algorithme suivant (propre aux modifications de rang 1) en pseudo-code, en posant $v_j = z_j^{(j)}$:

```
pour i variant de 1 à n
   v=z[i]
   p=c*v
```

```
dt=d[i]+p*v
beta=p/dt
pour j variant de i+1 à n
   z[j]=z[j]-v*l[j,i]
   l[j,i]=l[j,i]+beta*z[j]
si i≠n alors c=c*d[i]/dt
d[i]=dt
```

Intéressons nous maintenant au nombre d'opérations arithmétiques élémentaires requises pour l'algorithme de Fletcher & Powell.
Chaque étape de la boucle sur i nécessite :

- pour le calcul de p_i,d_i, et β_i : 4 opérations arithmétiques
- pour le calcul de la $i^{ème}$ colonne de l : 4(n-i)-1 opérations arithmétiques
- pour le calcul de c_i : 2 opérations arithmétiques

On peut donc en déduire le nombre total d'opérations :

$$\sum_{i=1}^{n} (4 + 2 + \sum_{j=i+1}^{n} 4) = 6(n-1) + 4\frac{n(n-1)}{2} = 2n^2+4n-2$$

Remarquons que la complexité de la décomposition LDL^t est de l'ordre de n^3 opérations arithmétiques. On en déduit donc que l'algorithme de Bennett est préférable dans le cas de modifications de rang 1. Rappelons que la méthode de Woodbury nécessite de l'ordre de n^2 opérations arithmétiques (mais qu'il faut au préalable connaître explicitement l'inverse de la matrice A).

2.2.2. Modifications de rang m (1<m<<n)

L'algorithme présenté est une adaptation de l'algorithme de Bennett qui permet de mettre à jour les facteurs pour une décomposition LDL^t d'une matrice A (symétrique et d'ordre n) ayant subie une modification de rang m (décrite en 2.1) : $B = A + ZCZ^t$.
Cette méthode présente un intérêt plus important lorsque m est trés inférieure à n (la taille de la matrice A), ce qui est pratiquement toujours vrai dans les applications étudiées ici.
L'algorithme de Bennett est basé sur une redécomposition astucieuse de la matrice modifiée.
Rappelons pour cela tout d'abord le principe de la décomposition LDL^t. Définissons deux notations suivantes :

- L_i la matrice unité dont la i^{eme} colonne est remplacée par la i^{eme} colonne de L.
- $K_{.i}$ (respectivement $K_{i.}$) la i^{eme} colonne (resp. la i^{eme} ligne) d'une matrice K

Factorisons la matrice A de façon à obtenir :

$$L_1 = \frac{1}{a_{11}} A_{.1}$$

On peut de plus partitionner A comme suit :

$$A = \begin{bmatrix} a_{11} & F^t \\ F & H \end{bmatrix}$$

Ce qui permet alors d'obtenir :

$$(L_1)^{-1} A \, (L_1)^{-t} = \begin{bmatrix} a_{11} & 0 \\ 0 & H - \frac{1}{a_{11}} FF^t \end{bmatrix}$$

Notons la matrice A ainsi partitionnée $A^{(1)}$. On peut alors écrire :

$$A^{(2)} = H - \frac{1}{a_{11}} FF^t$$

En répétant les opérations précédentes sur $A^{(2)}$, on obtient $L_{2.2}$ et d_2. La décomposition complète de la matrice A s'obtient alors en itérant le procédé.

Appliquons maintenant ce procédé de décomposition à la matrice A modifiée. La première colonne de la matrice modifiée fournit immédiatement $\tilde{L}_{.1}$ et $\tilde{d}_1$. Posons maintenant :

$$B^{(1)} = A^{(1)} + Z^{(1)} C^{(1)} Z^{(1)t}$$

On obtient alors (par analogie avec l'égalité portant sur la matrice A) :

$$(\tilde{L}_1)^{-1} B^{(1)} \, (\tilde{L}_1)^{-t} = \begin{bmatrix} \tilde{d}_1 & 0 \\ 0 & B^{(2)} \end{bmatrix}$$

où $B^{(2)}$ est une matrice carrée de taille n-1.
On suppose maintenant que $B^{(2)}$ s'écrit sous la forme :

$$B^{(2)} = A^{(2)} + Z^{(2)} C^{(2)} Z^{(2)t}$$

avec $Z^{(2)}$ matrice de taille (n-1)xm et $C^{(2)}$ une matrice de taille mxm. Ces deux matrices peuvent être explicitement calculées.
Enfin, $\tilde{L}_2$ et $\tilde{d}_2$ peuvent être calculés avec la première colonne de $A^{(2)}$ (elle même calculée à partir de L_2 et d_2). Ce qui permet de décomposer entièrement la matrice B.

Pour chaque étape i de l'algorithme, on peut ainsi agencer les calculs :

- $\tilde{d}_i = d_i + Z_{i.}^{(i)} C^{(i)} Z_{.i}^{(i)}$
- une boucle sur j allant de i+1 à n :

$$Z_{j.}^{(i+1)} = Z_{j.}^{(i)} - l_{ji} Z_{i.}^{(i)}$$

$$\tilde{l}_{ji} = l_{ji} + \frac{Z_{j.}^{(i+1)} p^{(i)}}{\tilde{d}_i}$$

- $C^{(i+1)} = C^{(i)} - \frac{p^{(i)} p^{(i)t}}{\tilde{d}_i}$

où l'on définit : $p_{(i)} = Z_{i.} C^{(i)}$.

L'algorithme de Bennett pour une transformation de rang m est donné ci-dessous en pseudo-code (on pose $\text{beta}[i] = \frac{p^{(i)}}{\tilde{d}_i}$) :

```
pour i variant de 1 à n
   { calcul du vecteur p }
   pour j et k variant de 1 à m
      p[j]=p[j]+z[i,k]*c[j,k]
   { mise à jour de d[i] }
   pour k variant de 1 à m
      d[i]=d[i]+z[i,k]*p[k]
   pour k variant de 1 à m
      beta[k]=p[k]/d[i]
   { mise à jour de la colonne i de L }
   pour j variant de i+1 à n
      pour k variant de 1 à m
         z[j,k]=z[j,k]-l[j,i]*z[i,k]
         l[j,i]=l[j,i]+z[j,k]*beta[k]
      si i≠n alors
      { mise à jour de la matrice C }
         pour j et k variant de 1 à m
            c[j,k]=c[j,k]-beta[j]*p[k]
```

Remarquons que l'algorithme décrit ci-dessus constitue une généralisation de l'algorithme de Bennett pour le rang 1 : les scalaires v_i,b_i et $z_j^{(i+1)}$ sont remplacés par des vecteurs de taille m et le scalaire c_{i+1} est lui remplacé par la matrice $C^{(i+1)}$ d'ordre m.

L'algorithme de Bennett est numériquement stable lorsque les matrice D et $\tilde{D}$ sont définies positives (on en trouve la démonstration dans [FlPo]pour une modification de rang 1),

soit en fait lorsque A et sa perturbée sont définies positives.

Intéressons nous maintenant au nombre d'opérations nécessaires à l'algorithme de Bennett pour une transformation de rang m. Tout d'abord chaque étape i du calcul requiert :

- calcul du vecteur p : $2m^2$ opérations
- mise à jour de d_i : 2m opérations
- calcul du vecteur beta : m opérations
- calcul de la $i^{ème}$ colonne de L : 4m(n-i) opérations
- mise à jour de la matrice C : $2m^2$ opérations

Par conséquent, le nombre total d'opérations est égal (pour une transformation de rang m) à $2mn^2+4m^2n+mn-2m^2$. On peut donc affirmer que l'algorithme de Bennett permet de réduire le nombre d'opérations nécessaire à une décomposition LDL^t de $\frac{n^3}{3} + O(n^2)$ à $2mn^2 +O(m^2n)$. On peut montrer que la méthode de Bennett est plus intéressante qu'une décomposition LDL^t dès que $\frac{n}{m} \geq 8$.

2.3. Mise à jour de la factorisation QR

De même, on peut effectuer la mise à jour des facteurs d'une matrice décomposée sous sa forme QR.
Nous allons maintenant étudier la mise à jour de la factorisation QR pour une modification de rang 1 ([GoVL]). Les matrices successives des systèmes linéaires à résoudre s'écrivent donc :

$$A_1 = A = QR$$
$$A_{i+1} = A_i + cuw^t$$

où u et w sont des vecteurs.
Décrivons tout d'abord briévement le principe, avant de donner l'algorithme. On cherche à obtenir $A_{i+1} = Q_1R_1$ de la façon suivante : on calcule tout d'abord $y=Q^tu$, puis les matrices de rotations H^2_j qui annulent le $j+1^{ème}$ coefficient d'un vecteur. On obtient donc $H^2_0 \dots H^2_{n-2}y=(r,0 \dots 0)^t$. On montre ensuite que la matrice $Hs=H^2_0 \dots H^2_{n-2}R$ est Hessenberg supérieure (c'est à dire $Hs_{ij}=0$ si $i>j+1$).
Ainsi donc, $H^2_0 \dots H^2_{n-2}(R+ywt)=H_1$ est Hessenberg supérieure, il est alors possible de trouver la matrice de Householder G^2_j annulant le $j+1^{ème}$ élément de la $j^{ème}$ colonne de H_1. On obtient finalement (en remarquant que $A_{i+1}=Q(R+yw^t)$:

$$R_1 = G^2_{n-2} \dots G^2_0H_1$$
$$Q_1{}^t = G^2_{n-2} \dots G^2_0H^2_0 \dots H^2_{n-2}Q^t$$

Avant de donner l'algorithme de la mise à jour de la décomposition QR elle-même, voici quelques définitions algorithmiques préliminaires. L'algorithme de calcul de la matrice de Householder H^2_j annulant le $j+1^{eme}$ élément du vecteur R est donné ci-après en pseudo-code :

```
m=max{|R[j]|, |R[j+1]|}
alpha=0
pour i variant de j à j+1
   v[i]=R[i]/m
   alpha=alpha+R[i]*R[i]
alpha=sqrt(alpha)
beta=alpha*(alpha+|v[j]|)
v[j]=v[j]+signe(v[j])*alpha)
```

L'algorithme ci-dessous (écrit en pseudo-code) permet d'accumuler une matrice de Householder H^2_j dans une matrice M (soit $M=H^2_jM$) :

```
pour p variant de 0 à n-1
   s=v[j]*M[j,p]+v[j+1]*M[j+1,p]
   s=s/beta
   pour i variant de j à j+1
      M[i,p]=M[i,p]-sv[i]
```

Enfin, l'accumulation d'une matrice de Householder H^2_j dans un vecteur R (soit $R=H^2_jR$) peut être effectuée comme suit (l'algorithme est écrit en pseudo-code) :

```
s=v[j]*R[j]+v[j+1]*R[j+1]
s=s/beta
pour i variant de j à j+1
   R[i]=R[i]-s*v[i]
```

L'algorithme de la mise à jour de la factorisation QR pour une transformation de rang 1 est donné ci-dessous (en pseudo-code) :

```
calcul de y=Q^t u
pour j variant de n-2 à 0
   calcul de H2j pour le vecteur y
   accumulation de H2j dans y
   accumulation de H2j dans A
   accumulation de H2j dans Q^t
```

La décomposition QR pour une matrice Hessenberg supérieure nécessite $16n^2-4n-12$ opérations arithmétiques. Le coût de l'algorithme complet de mise à jour est donc :

- pour le calcul de y : n(n-1) opérations
- pour la boucle : (n-1)(12+8+16n)
- pour le calcul de A : 2n
- pour la factorisation QR : $16n^2-4n-12$

Le nombre total d'opérations est donc de $34n^2+n-32$. Rappelons que le nombre d'opérations nécessaires à la modification des facteurs de Cholesky (dans le cas d'une modification de rang 1) est de$2n^2+4n-2$, ce qui est nettement inférieur.

Nous allons maintenant présenter une méthode itérative originale, basée sur un préconditionnement spécifique de l'allgorithme du gradient conjugué préconditionné.

2.4. Gradient conjugué pour la résolution de systèmes successifs

2.4.1. Présentation du problème

Considérons n systèmes linéaires consécutifs, chacun de taille n par n :

$$A_i x_i = b_i, \ 1 \le i \le N, \qquad (1)$$

où les matrices A_i sont "proches" les unes des autres, ie. $A_{i+1}=A_i+\Delta_i$, avec Δ_i tel que les solutions x_i soient proches les unes des autres au sens de la norme 2, pour presque toutes les valeurs de i. Ce problème a déjà été discuté dans [CoTr] pour les applications au traitement du signal dans lesquelles Δ_i est définie positive.

En filtrage adaptatif et en contrôle de processus, les matrices de covariance A_i sont généralement mises à jour par des matrices de rang 1 : Δ_i, et les équations résultantes (1) sont résolues en utilisant le lemme d'inversion des matrices [Hayk]. Cela conduit à l'algorithme des moindres carrés récursifs [Hayk]. Malheureusement, il s'avère que cet algorithme devient moins intéressant pour des mises à jour de rang p ($p \ge 2$ ou 3) à cause de sa complexité en $8pn^2+4p^2n+p^3/3$, comparée à la résolution directe qui prend $\frac{n^3}{3}+O(n^2)$ opérations arithmétiques [CoTr]. Notons qu'il existe une version plus rapide, mais elle n'est pas toujours numériquement stable [Ciof], contrairement à l'algorithme proposé ici.

De plus, le problème de mise à jour apparaît aussi en contrôle avec identification de modèle [AsWi] (temps partagé ou temps réel, en contrôle adaptatif), et dans le filtrage adaptatif par bloc [ClMP]. Dans la plupart des cas, la taille de A_i est relativement petite (de l'ordre de n=100), mais peut aussi être assez grande dans certains cas (entre 200 et 1000).

Pour tous ces problèmes, les modifications d'une étape à l'autre peuvent être de rang beaucoup plus grand que un, et mêmes pleines.

2.4.2. Résolution itérative

Les méthodes itératives ont été originellement introduites pour les systèmes creux [Mant][MeVa] ; notons que nous nous intéressons ici à des matrices pleines. Un algorithme a été proposé dans [CoTr], il correspond en fait au préconditionnement de l'algorithme du gradient par la matrice A_1 sous sa factorisation par la méthode de Cholesky. En d'autres termes, nous formons :

$$A_i=A_1+R_i \qquad \text{avec } R_i=\sum_{k=2}^{i} \Delta_k, \text{ pour } i\geq 2.$$

Ainsi, alors que de plus en plus d'itérations se déroulent, il peut devenir nécessaire de mettre à jour la matrice de préconditionnement M (égale au produit $L_1.L_1^t$), en utilisant une autre matrice A_{i_0} de plus grand indice ($i_0>1$) ; et ainsi de suite. Nous pouvons appeler la mise à jour de M une "réinitialisation". Dans [CoTr], une étude de complexité prouve que cet algorithme est meilleur qu'une simple résolution par la méthode de Cholesky en terme du nombre d'opérations arithmétiques.

2.4.3. Description de l'algorithme du gradient conjugué

L'algorithme du gradient conjugué a été proposé par Hestenes et Stiefel, il y a 40 ans, pour résoudre les systèmes linéaires Ax=b, avec des matrices A symétriques définies positives [HeSt]. L'idée clé de la méthode est de considérer la résolution du système Ax=b comme le calcul du minimum de la fonction F définie par :

$$F(x) = 1/2\ x^t.Ax - x^t.b$$

où b est un vecteur de IR^n et A une matrice de taille n par n supposée symétrique définie positive.

Minimiser F et résoudre le système linéaire Ax=b sont des problèmes équivalents (en effet, le minimum de F est obtenu au point $x=A^{-1}b$). On résout ce nouveau problème en utilisant une méthode usuelle de descente. Cela consiste à choisir une direction de descente d_k et calculer x_{k+1} itérativement au $(k+1)^{eme}$ pas par la relation suivante :

$$x_{k+1} = x_k - \rho_k . d_k$$

Le paramètre réel ρ_k est choisi de manière optimale dans la direction d_k ; cela conduit à l'expression :

$$\rho_k = \frac{(r_k)^t . r_k}{(A.d_k)^t . d_k}$$

où r_k est le vecteur gradient en x_k que l'on calcule itérativement. La figure ci dessous illustre le principe de cette méthode de descente.

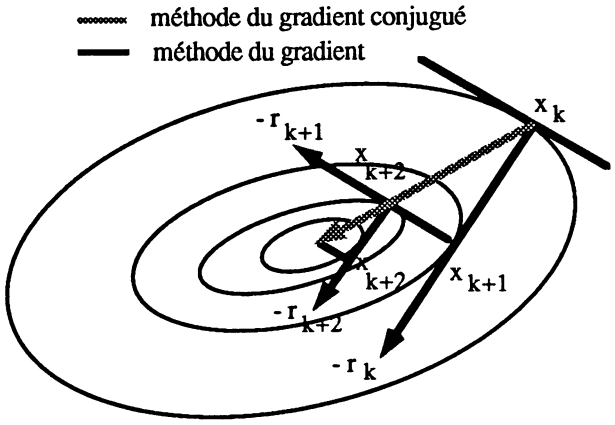


Figure 6 : principe des méthodes du gradient et du gradient conjugué

De même, la nouvelle direction d_{k+1} est calculée itérativement comme une accélération de la méthode usuelle du gradient sous la forme :

$$d_{k+1} = r_{k+1} - \beta_{k+1} d_k$$

de telle sorte à être conjuguée à d_k (i.e. orthogonale au sens du produit scalaire défini par A). Une propriété fondamentale de cet algorithme est que les directions sont toutes 2 à 2 conjuguées.
Pratiquement, on part du vecteur initial x_0=0 (qui n'est pas un choix restrictif). L'algorithme est donné ci-dessous (en pseudo-code) :

```
Initialisation (x,r,old,d)
pour k=1, jusqu'à n répéter
   v = Ad
   rho = old / vt.d
   x = x - rho*d
   r = r - rho*v
   new = rt.r
```

```
    beta = new / old
    d = r - beta*d
    old = new
jusqu'à sqrt(r^t.r)<précision
```

La méthode du gradient conjugué ainsi définie converge pour les matrices symétriques définies positives. De plus, elle s'arrête (converge) après au plus n itérations pour un système de taille n par n ([GoVa][AxBa]). La terminaison de l'algorithme est garantie par les propriétés de conjugaison des directions.
De nombreux résultats, tant expérimentaux que théoriques, montrent que la vitesse de convergence de la méthode dépend de l'inverse C_A du conditionnement de la matrice (défini pour la norme L_2 dans le cas symétrique par le rapport des valeurs propres extrêmes). En effet, si l'on note e_k l'erreur $x-x_k$ au k^{eme} pas (x désignant la solution vraie), on a la majoration suivante :

$$\|e_k\|_A \leq \left(\frac{1 - \sqrt{C_A}}{1 + \sqrt{C_A}}\right)^{2k} \|e_0\|_A$$

En d'autres termes, plus le conditionnement est proche de 1, meilleure est la convergence. On peut montrer plus précisément que la convergence est de plus influencée par la répartition des valeurs propres [GoVa].

Le lien entre rapidité de convergence et conditionnement permet d'imaginer une amélioration intéressante du gradient conjugué. Soit L une matrice non singulière, telle que la matrice $L^{-1}AL^{-t}$ ait un meilleur conditionnement que A. Alors, plutôt que de résoudre directement le système Ax=b, il sera plus rapide d'appliquer le gradient conjugué au nouveau système :

$$L^{-1}AL^{-t}y=L^{-1}b \qquad (S)$$

Après application de la procédure du paragraphe précédent au système (S) et le changement de variables $y=L^tx$ pour se ramener à x, on obtient la procédure suivante (écrite en pseudo-code) qui sera par la suite notée `PCG (A,M,x,b,n,k)` :

```
Initialisation (x,b,r,z,old,d)
pour k=1 jusqu'à n  répéter
   v = A*d
   rho = old / v^t.d
   x = x - rho*d
   r = r - rho*v
   résoudre Mz = r
   new = r^t.z
   beta = new / old
```

```
    d = z - beta*d
    old = new
jusqu'à  sqrt(r^t.r) < précision
```

Dans cette nouvelle procédure, un système supplémentaire est à résoudre à chaque itération. On remarque que ce n'est plus L qui intervient, mais la matrice de préconditionnement $M=LL^t$. La question fondamentale est de "bien" choisir M. Typiquement, M doit vérifier trois propriétés :

(1) le système Mz=r doit être facile à résoudre de sorte que la diminution du nombre d'itérations ne soit pas trop pénalisée par un nombre excessif de calculs supplémentaires
(2) M doit être symétrique définie positive pour assurer la convergence
(3) et enfin, M doit être une *approximation* de A

Le vecteur gradient r est calculé itérativement (ce qui permet d'économiser un produit matrice-vecteur Ar). Il se peut qu'alors le cumul des erreurs entraîne une dérive numérique sur les directions qui ne sont plus tout à fait conjuguées. Il peut être nécessaire de réinitialiser de temps en temps le vecteur gradient, par exemple en le calculant par la formule directe plus coûteuse [GoVL]. Ce phénomène n'est cependant pas critique dans les cas qui nous intéressent.
Le test d'arrêt porte sur la norme L_2 du gradient, et non pas sur $r^t.z$ (on a donc un produit scalaire supplémentaire à calculer à chaque étape). Le critère de convergence est identique à celui du gradient conjugué simple pourvu que la matrice M soit symétrique définie positive.

On trouve dans la littérature de nombreuses stratégies de préconditionnement, chacune adaptée à un problème spécifique (on peut par exemple consulter [GM], [Gree], [GoMe], [MeVa], [Try1]). Quelque unes sont données ci-dessous à titre d'exemple, mais il n'existe pas en fait de règle à appliquer systématiquement ; ainsi, chaque problème doit être étudié séparément.

Diagonal : c'est le plus simple. L'approximation de A par sa diagonale n'est pas très bonne, mais la résolution du système préconditionné est triviale (et intrinséquement vectorielle) et ne requière aucun stockage.

Cholesky incomplet : ce préconditionnement est très employé. Le principe est d'approximer A par sa factorisation de Cholesky astreinte à respecter la structure initiale de A. Ce préconditionnement nécessite un calcul préalable coûteux et cela double le stockage mémoire, mais l'approximation est meilleure. De toute façon, les matrices dont il est question ici ne sont pas creuses, et de plus, ce préconditionnement n'est pas robuste (les matrices peuvent être singulières).

D'autres approches sont envisageables ...

Chaque itération d'un gradient conjugué comporte : trois produits scalaires, trois saxpys, la résolution d'un système élémentaire et un produit matrice-vecteur.
Pour les grands systèmes dont les matrices sont pleines, le coût d'une itération est donc réduit au produit matrice-vecteur, pourvu que la résolution du système reste négligeable devant celle du produit matrice-vecteur.

2.4.4. L'algorithme du gradient conjugué préconditionné pour les systèmes consécutifs

L'idée de base de la méthode du gradient conjugué préconditionné pour les systèmes linéaires consécutifs est la suivante : à l'étape i_0 on effectue une factorisation de Cholesky de la matrice A_{i0} (ainsi que les résolutions triangulaires afin d'obtenir la solution du système), puis tant que le temps de résolution reste inférieur à la factorisation de Cholesky on résout les systèmes suivants par une méthode de gradient conjugué préconditionné, le préconditionnement étant la factorisation de Cholesky de la matrice A_{i0}. Cet algorithme est détaillé ci-dessous.
Dans la présente approche, la matrice A_1 est initialement choisie comme préconditionnement, sous sa forme factorisée par la méthode de Cholesky. L'algorithme proposé ci dessous (écrit en pseudo-code) utilise la procédure PCG (définie 2.4.3) à chaque étape i :

```
i:=1
io:=i
tant que i<>N-1 faire
   factoriser Aio en L.L^t
   résoudre L.z = bio et L^t.x = z
   i:=i+1
   répéter
      PCG(Ai,Aio,x,bi,n,nbiter)
      i:=i+1
   jusqu'à i=N-1 ou nbiter=nmax
   si nbiter=nmax alors io=i
```

Notons que la résolution est aisée puisque le préconditionnement (une factorisation de Cholesky) est effectué une fois pour toutes, ou au moins jusqu'à la prochaine réinitialisation. La figure suivante illustre sur un exemple ce processus de réinitialisation (avec une réinitialisation tous les 7 systèmes linéaires).

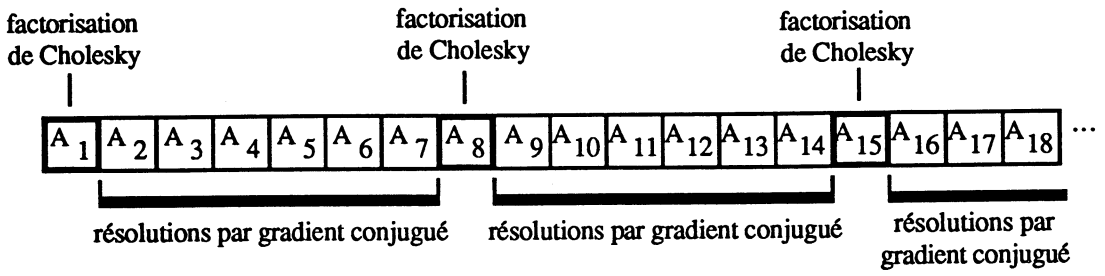


Figure 7 : principe de réinitialisation à une direction

La détermination pratique du pas de la réinitialisation dépend de plusieurs facteurs. Le but théorique est clair : il faut que sur un pas (c'est à dire entre deux réinitialisations par la méthode de Cholesky) les résolutions avec les gradients conjugués préconditionnés prennent moins de temps que ces mêmes résolutions par des factorisations de Cholesky.

Tout d'abord lorsque l'algorithme est utilisé sur une machine séquentielle classique, on peut déterminer principalement deux stratégies.
En pratique, le nombre d'itération `nbiter` est comparé à un seuil `nmax`, choisi à l'avance pour s'assurer que la complexité reste plus petite qu'une résolution directe par Cholesky.
Une autre stratégie consiste pour le premier pas à mesurer directement les temps de résolution, et à déterminer ainsi le nombre de systèmes linéaires dont les résolutions par la méthode du gradient conjugué préconditionné reste moins couteuse qu'une résolution directe par une factorisation de Cholesky. Notons que dans ce cas la taille du pas choisi est fixe (ce qui peut poser des problèmes si les modifications des systèmes successifs ne restent pas constantes).

Sur des machines parallèles, les contraintes spécifiques à l'implantation peuvent entrer en ligne de compte. Par exemple, dans le cas des implantations réalisées sur le FPS T40 en 6.2.2, il est très intéressant que le pas soit un multiple du nombre de processeurs ; les résolutions se faisant en parallèle, on peut se permettre que la résolution des derniers systèmes prennent plus de temps qu'une résolution directe par la méthode de Cholesky. En fait, il est possible de jouer sur le pas de la résolution, quitte à ce que certaines résolutions par le gradient conjugué soient plus coûteuses qu'une résolution directe par la méthode de Cholesky, mais globalement on doit obtenir un gain de temps par rapport à des résolutions directes par la méthode de Cholesky.

Le nombre d'itérations, qui est borné par la taille du problème en arithmétique exacte, est connu pour être plus petit avec le gradient conjugué préconditionné qu'avec le gradient conjugué car la vitesse de convergence est beaucoup plus grande [GoVa]. Plus précisément, plus le système est grand, plus intéressant est l'algorithme du gradient conjugué préconditionné, bien que chaque pas du gradient conjugué préconditionné requiert un peu plus de calculs. Cependant, pour des systèmes mal conditionnés, la méthode peut conduire à la solution en plus de n itérations.

2.4.5. Complexité de la méthode

Chaque réinitialisation nécessite une nouvelle factorisation de Cholesky et la résolution de deux systèmes triangulaires. Le nombre de multiplications est donc $n^3/6+n^2+O(n)$.
La complexité de l'algorithme du gradient conjugué préconditionné peut être bornée par ce qui suit, dans la mesure où il dépend de la qualité du préconditionnement. Le nombre d'étapes requises varie (et augmente de fait) d'un système à l'autre. Si nbiter étapes ont été effectuées, un total de $O(2nbitern^2)$ multiplications sont requises car $2n^2+4n$ opérations flottantes sont effectuées à chaque passage dans la boucle.
Ainsi, la complexité totale est presque tout le temps (sauf à chaque réinitialisation) bornée par $O(2nmaxn^2)$. Ceci est confirmé par les temps d'exécution obtenus en 2.4.7.1.

2.4.6. Améliorations supplémentaires

Une autre amélioration intéressante peut être ajoutée. Le nombre d'itérations requises augmente pour la résolution d'un système par le gradient conjugué préconditionné lorsque ce système "s'éloigne" du système factorisé par Cholesky. Il est possible de diminuer le nombre de réinitialisations en appliquant une meilleur stratégie expliquée ci-aprés.
Dans l'algorithme précédent, lorsqu'un système $A_{i_o}x=b_{i_o}$ est résolu directement (réinitialisation), $M=L_{i_o}.L_{i_o}^t$ est alors utilisé pour résoudre les autres systèmes pour $i>i_0$. Remarquons que les réinitialisations ne doivent pas être espacées de i_0, mais peuvent l'être de i_1. Puisque les plus coûteuses des opérations sont les réinitialisations (ie. moins il y en a, mieux cela est), nous pouvons choisir $i_1 = ai_o$, $1<a\leq 2$. Notons que dans ce cas, le nombre d'itérations peut excéder nmax. L'amélioration réside dans le fait que, une fois la résolution directe effectuée, disons à l'indice i(k), il est possible de résoudre successivement les systèmes de matrice A_i, non seulement pour i>i(k), mais aussi pour i<i(k), en utilisant la matrice $M=A_{i(k)}$ (pour autant que ces systèmes n'aient pas déjà été résolus). Cette méthode permet d'effectuer a fois moins de réinitialisations pour SPCG décrit en 2.4.4. Ce nouvel algorithme peut être formulé comme suit :

1) utiliser l'algorithme décrit en 2.4.4 jusqu'à la première réinitialisation.
2) cela donne une valeur de i_o (et $i_1 = ai_o$), que nous décidons de garder constant.
3) effectuer une réinitialisation (factorisation de Cholesky) à chaque $i(k)=1+k[2(i_1-1)+1]$.
4) utiliser l'algorithme du gradient conjugué préconditionné, une fois vers la droite en partant de i(k)+1 jusqu'à $i(k)+i_1-1$, et une fois vers la gauche en partant de i(k)-1 jusqu'à $i(k)-i_1+1$.

Le choix du pas à l'étape 1 dépend de l'objectif désiré par l'utilisateur (principalement le nombre de systèmes linéaires successifs, leurs comportement et la machine utilisée), voir 2.4.4 pour une discution du choix du pas de la réinitialisation.

Cette méthode est illustrée sur la figure ci-dessous avec $i_1=2*i_o$ et $i_o=4$.

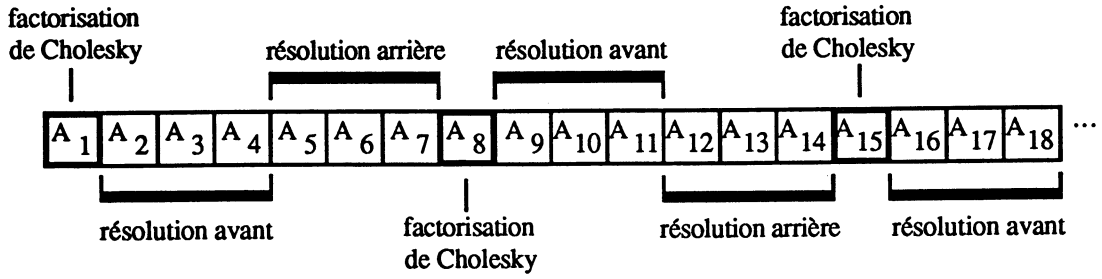


Figure 8 : principe de réinitialisation à deux directions

2.4.7. Expérimentations numériques

Ces simulations ont été implantées sur un Sun3, en langage C. Les systèmes ont été simulés de façon à être proches les uns des autres au sens de la norme 2 :

- A1 est diagonale dominante, avec un conditionnement plus grand que la taille n
- n valeurs sont aléatoirement légérement modifiées pour obtenir A_i à partir de A_{i-1} avec une petite augmentation de la norme 2 (moins de 1%) ; notons que les matrices successives restent symètriques définies positives et que les expérimentations suivantes ont été effectuées pour des modifications de rang plein

2.4.7.1. Réinitialisation à une direction

La réinitialisation à une direction est une implantation de l'algorithme décrit en 2.4.4. Le programme est écrit de façon à ce qu'une réinitialisation soit effectuée à chaque fois que l'algorithme du gradient conjugué préconditionné pour les systèmes successifs prend plus de temps CPU que la factorisation de Cholesky précédente. Cela garanti une meilleure complexité que la factorisation de Cholesky dans tous les cas. Cette stratégie est appelée "réinitialisation à une direction" parce que les systèmes $A_ix_i=b_i$ sont résolus pour des valeurs croissantes de i.

Les résultats numériques obtenus sont donnés ci-dessous pour diverses tailles de matrices.

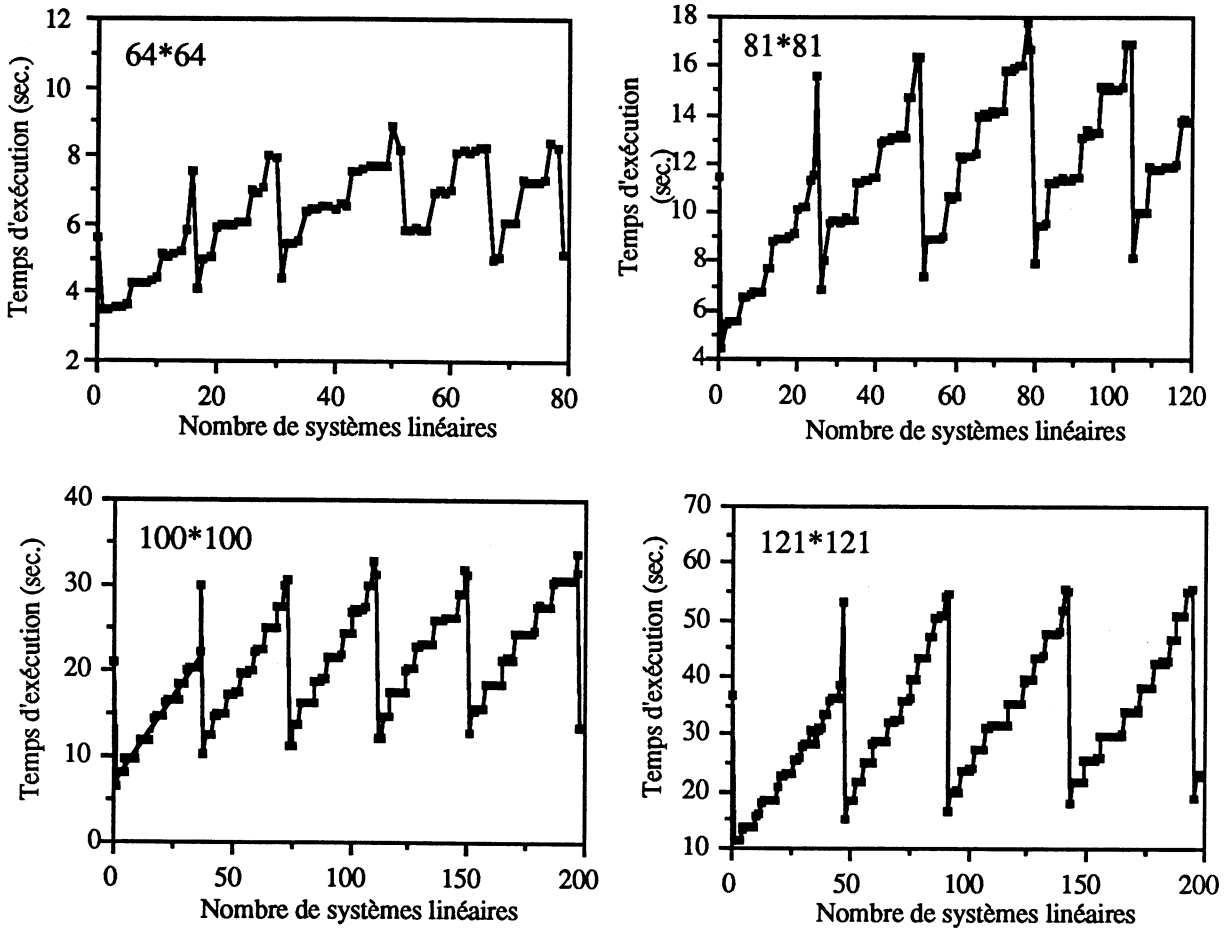


Figure 9 : réinitialisation à une direction

Les "sauts" que l'on peut constater sur la figure ci-dessus correspondent à un pas de l'algorithme du gradient conjugué préconditionné pour les systèmes linéaires successifs (c'est à dire aux résolutions par le gradient conjugué préconditionné effectuées entre deux réinitialisations). Dans chaque cas, les les résolutions avec les temps d'exécution les plus élevées (celles qui sont espacées par un pas) sont les factorisations par la méthode de Cholesky.

Une seconde classe d'expérimentation est maintenant présentée avec des matrices mal conditionnées et bien conditionnées afin d'étudier l'influence du conditionnement des matrices sur le pas des réinitialisations.
Le conditionnement des matrice a été modifié en faisant varier la norme des modifications qui permettent de passer d'un système à l'autre.

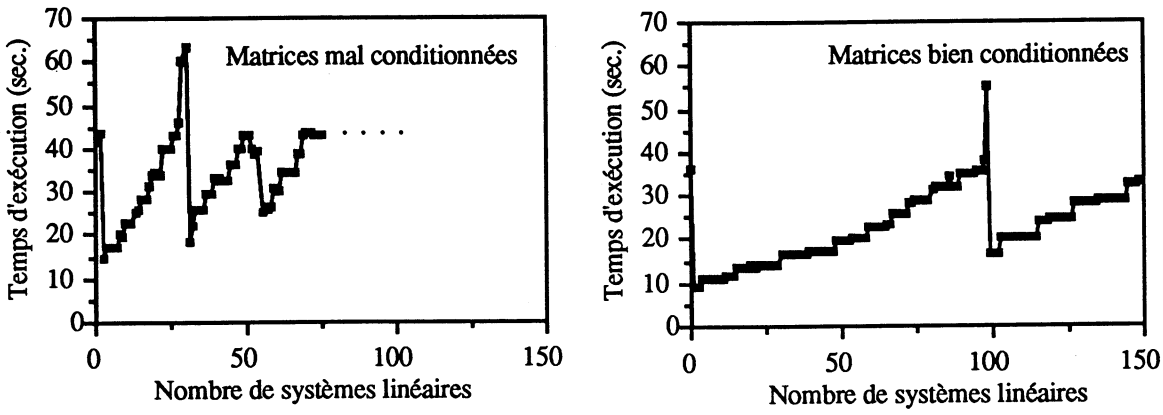
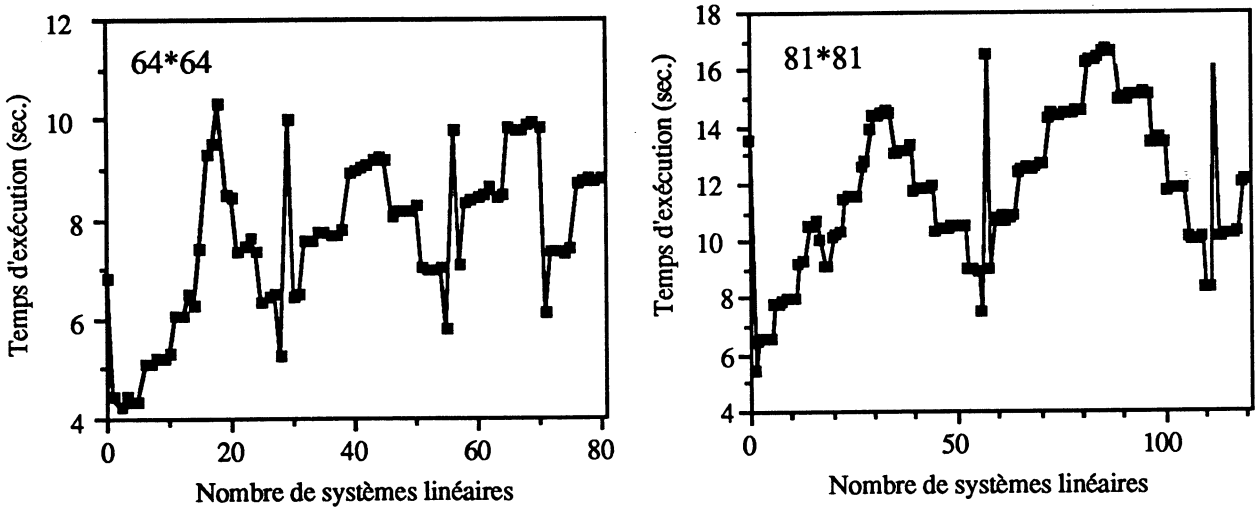


Figure 10 : expérimentations sur le conditionnnement (matrices 121x121)

On remarque que dans le cas de matrices mal conditionnées le pas entre deux réinitialisations est d'environ 30 systèmes, et que dans le cas de matrices bien conditionnées il est de près de 100 systèmes.
Il apparaît donc que la méthode du gradient conjugué préconditionné pour les systèmes linéaires successifs est d'autant plus intéressante que les matrices sont bien conditionnées : on peut alors résoudre un nombre très élevé de systèmes linéaires avec le même préconditionnement.

2.4.7.2. Réinitialisation à deux directions

La réinitialisation à deux directions est une implantation de l'algorithme amélioré décrit en 2.4.6. Ici, les systèmes $A_i x_i = b_i$ sont résolus à la fois pour des valeurs de i croissantes et décroissantes. Les résultats numériques obtenus avec les mêmes matrices qu'au paragraphe précédent sont donnés figure 11.



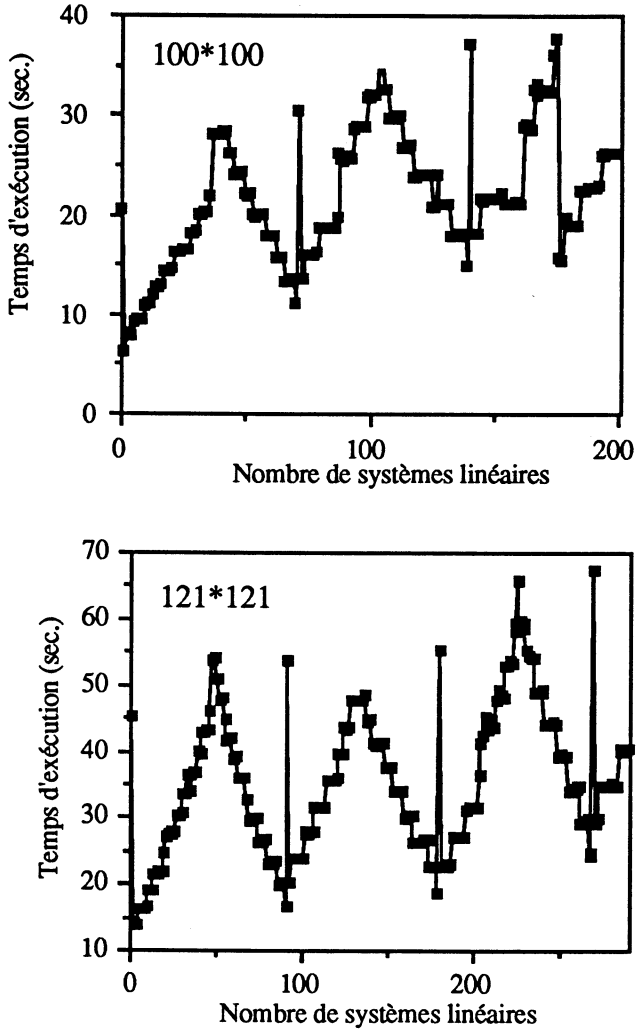


Figure 11 : résultats numériques pour la réinitialisation à deux directions

Dans les résultats ci-dessus, comme $i_1=2i_0$ les coûts des deux méthodes du gradient conjugué préconditionné pour les systèmes linéaires successifs sont approximativement les mêmes (l'algorithme du gradient conjugué préconditionné calculé dans le pire des cas a le même coût qu'une résolution par la méthode de Cholesky). En pratique, on réinitialise plus souvent de façon à obtenir de plus petits intervalles avec le même préconditionnement). Le choix du "meilleur" intervalle est trés dépendant du problème ; la meilleure valeur peut être obtenue heuristiquement. La figure ci-dessous montre l'amélioration en prenant un intervalle de longueur a=2/3 de l'initial (par exemple $i_1=35$ pour une taille 121).

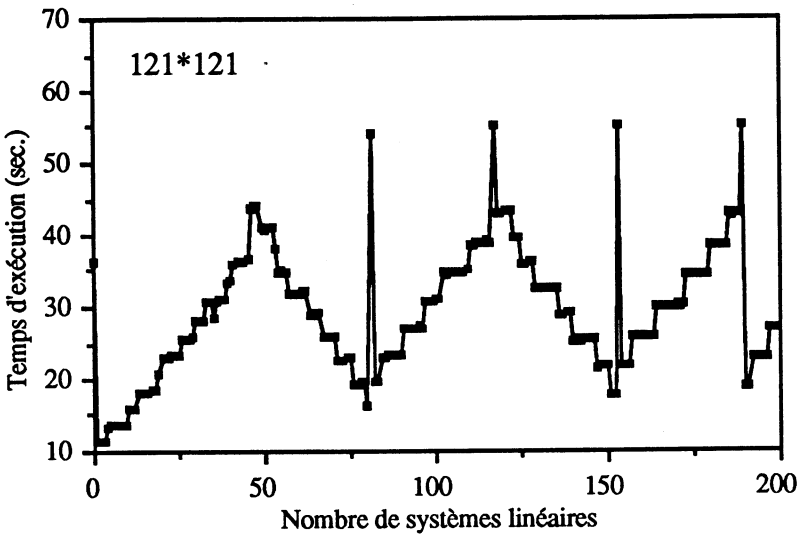


Figure 12 : réinitialisation 2-D avec un intervalle de plus petite longueur

Le gain entre une réinitialisation à deux directions et une réinitialisation à une direction est décrit sur la figure 13.

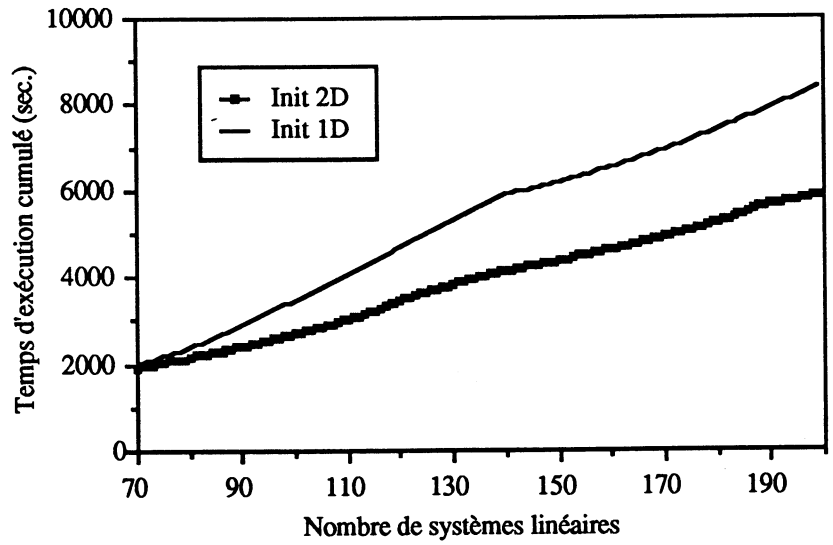


Figure 13 : comparaison entre réinitialisations 1-D et 2-D (n=121)

2.4.7.3. Comparaison avec les méthodes usuelles

La figure ci-dessous montre l'énorme gain en temps d'exécution lorsque l'on compare la résolution par la méthode de Cholesky sans prendre en compte la structure proche des matrices et l'algorithme du gradient conjugué préconditionné pour les systèmes linéaires successifs.

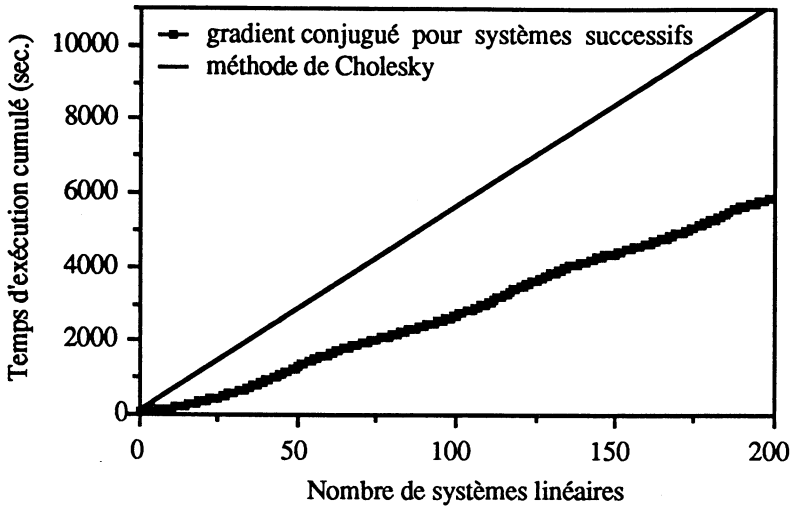
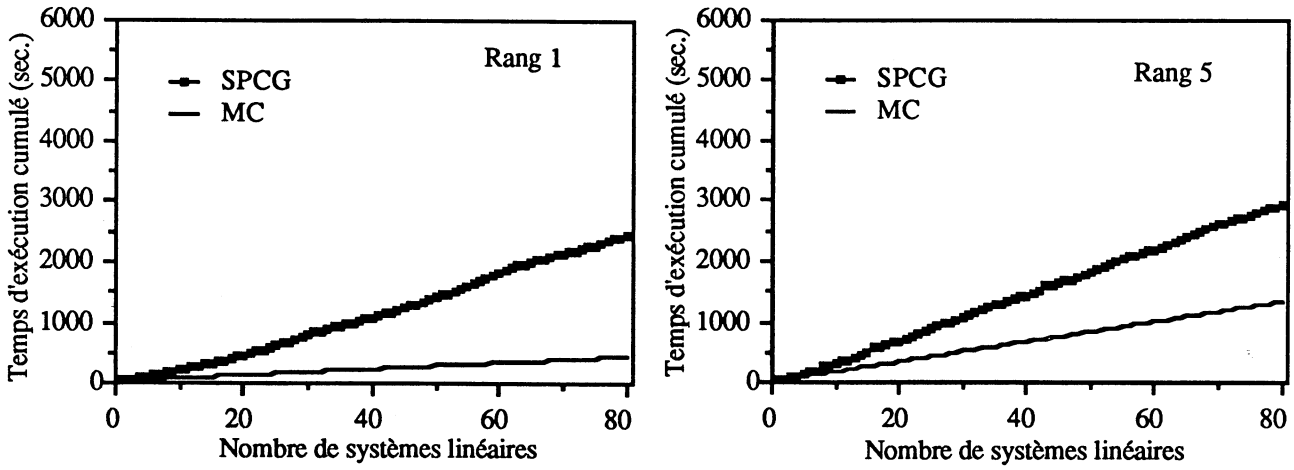


Figure 14 : comparaison entre méthode de Cholesky successifs et gradient conjugué préconditionné pour systèmes linéaires successifs (n=121)

Des méthodes directes peuvent être utilisées pour résoudre ce problème. Une comparaison entre les algorithmes de factorisation de Cholesky modifiée (méthodes de Fletcher et Powell, de Bennett décrites en 2.2.1 et 2.2.2, désignée sur la figure ci-dessous par MC) et de gradient conjugué préconditionné pour systèmes linéaires successifs (désignée sur la figure ci-dessous par SPCG) est donnée sur la figure 15 (pour des matrices de taille 121x121).



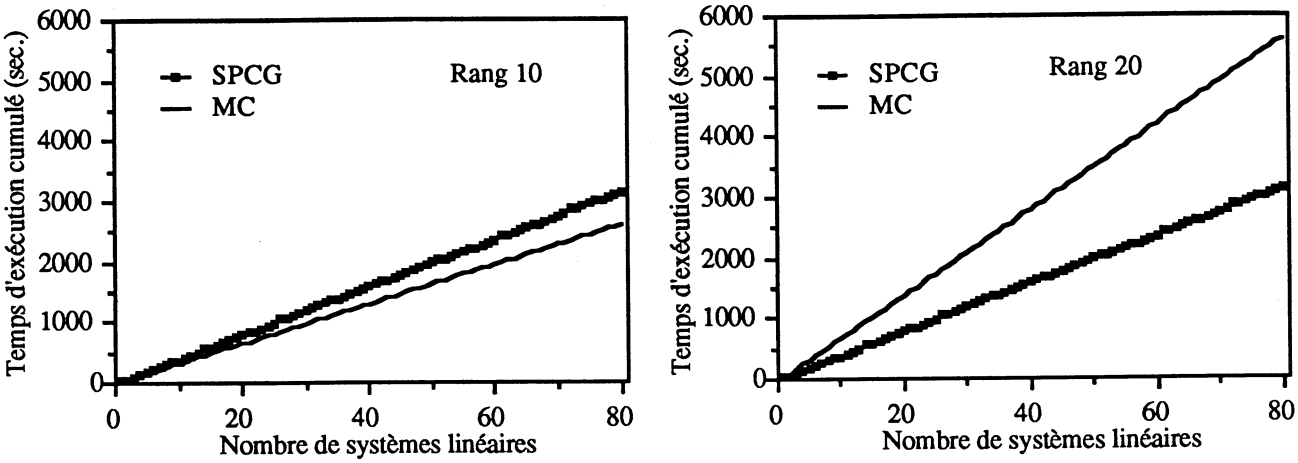


Figure 15 : comparaison entre différentes méthodes

L'algorithme du gradient conjugué préconditionné pour systèmes linéaires sucessifs apparaît donc comme meilleur que la factorisation de Cholesky modifiée pour des modifications de rang élevé.

Chapitre 3

Présentation du FPS T40

Où l'on présente le FPS T40, en insistant sur les deux aspects particuliers de cette machine : les calculs vectoriels sur chaque processeur et les communications. Le Transputer est aussi présenté en détails (ce dont on s'affranchira dans le chapitre 8).

3.1. Architecture du FPS T40

3.1.1. Généralités

Le FPS T40 est une machine comprenant 32 processeurs vectoriels, la mémoire est distribuée sur chaque processeur, à raison de 1 Mo pour chacun. L'idée de base de la série T réside dans la topologie des communications : les connexions forment un hypercube (un hypercube de dimension n est une topologie de 2^n nœuds, chacun étant relié à n voisins). L'intérêt majeur de ce type de configuration est qu'il réalise un bon compromis entre le relativement faible nombre de liens matériels et la proximité entre les différents nœuds (le chemin entre deux processeurs est au plus de Log(n)). Notons qu'il existe d'autres graphes de connexion possibles, à nombre de voisins directs constants, à distance maximale en Log(n), etc.., mais ces topologies sont souvent non symétriques et plus difficiles à mettre en œuvre pratiquement.
Le FPS T40 (40 est égal à 32 en octal) est constitué d'un hypercube de dimension 5, représenté sur la figure suivante.

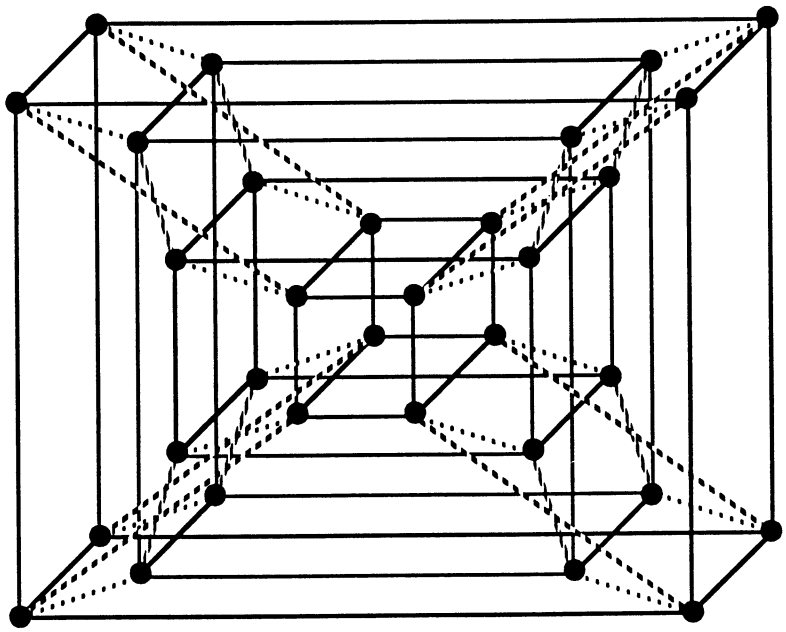


Figure 16 : hypercube de dimension 5

Il n'existe pas de synchronisation globale entre les processeurs du T40, il peut donc être considéré dans la classification de Flynn comme une machine MIMD asynchrone, à mémoire distribuée où les processeurs communiquent par échange de messages.

3.1.2. Structure du FPS T40

Les ordinateurs de la série T sont mono-utilisateurs et doivent être reliés à une machine hôte qui sert à l'interface avec les utilisateurs. La conception est modulaire : chaque module est constituée d'un 3-cube (soit 8 processeurs) qui est relié par un bus à un processeur spécialisé chargé des opérations systèmes (figure 17). Les processeurs-systèmes sont reliés entre eux par l'anneau-système, et les processeurs sont munis de facilités vectorielles et sont reliés par une structure hypercube.
La modularité de ces machines permet aisément de construire des modèles plus importants (par exemple il existe un T200 de 128 processeurs à Los Alamos, USA) en augmentant la dimension de l'hypercube par ajout de modules supplémentaires.

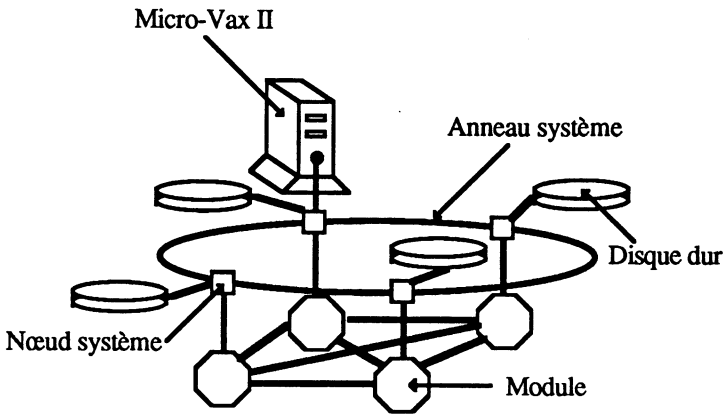


Figure 17 : structure générale du FPS T40

Chaque nœud système possède un disque dur Winchester (d'une capacité de stockage de 85 Mo). Les entrées-sorties entre le T40 et la machine hôte s'effectue via un bus connecté au nœud système 0 (cf figure 18).

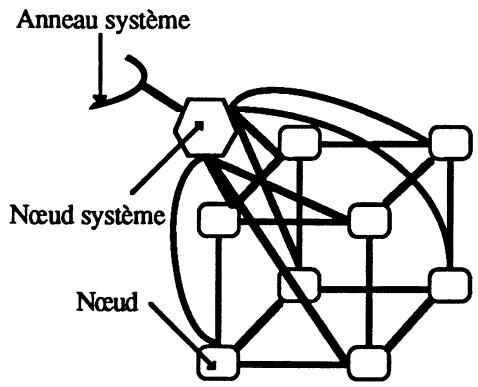


Figure 18 : détail d'un module

3.1.3. Environnement logiciel

L'environnement des machines hôtes de la série T est classiquement un système d'exploitation de type Unix. Les langages de programmation actuellement disponibles sont C et Fortran, auxquels sont ajoutés diverses primitives permettant d'exploiter le parallélisme. Le premier langage disponible était Occam (le langage dédié au Transputer) qui permettait de mieux exprimer le parallélisme et surtout de gagner énormément en performances au niveau des communications. Le passage aux langages de haut niveau a simplifié l'approche du parallélisme et ainsi élargi le nombre des utilisateurs potentiels, mais a perdu beaucoup en efficacité !
La machine étant mono-utilisateur, une séance typique se déroule ainsi : l'utilisateur se connecte à la machine hôte, écrit son programme (en C ou Fortran) et le compile (en utilisant un compilateur propre au FPS). Une fois le fichier exécutable crée, l'utilisateur doit soit s'assigner le T40 pour une séance interactive, soit lancer son programme par la commande "trun".
Comme pour la plupart des programmes qui s'exécutent sous Unix, l'utilisateur peut dévier les entrées/sorties classiques (le clavier et l'écran) pour faire exécuter son programme sur des fichiers.

3.1.4. Description d'un nœud

Un nœud vectoriel de l'hypercube comprend trois parties principales (cf figure 19) :

- un Transputer (T414) qui assure les communications avec les processeurs voisins ou avec le nœud système, sert d'unité arithmétique et logique et assure le contrôle du programme
- une unité de calcul vectoriel qui permet d'effectuer des calculs flottants très rapidement sur des vecteurs codés sur 64 bits (format IEEE double précision), grâce à un additionneur et un multiplieur pipelinés
- une mémoire vidéo trés performante d'une capacité de 1 Mo

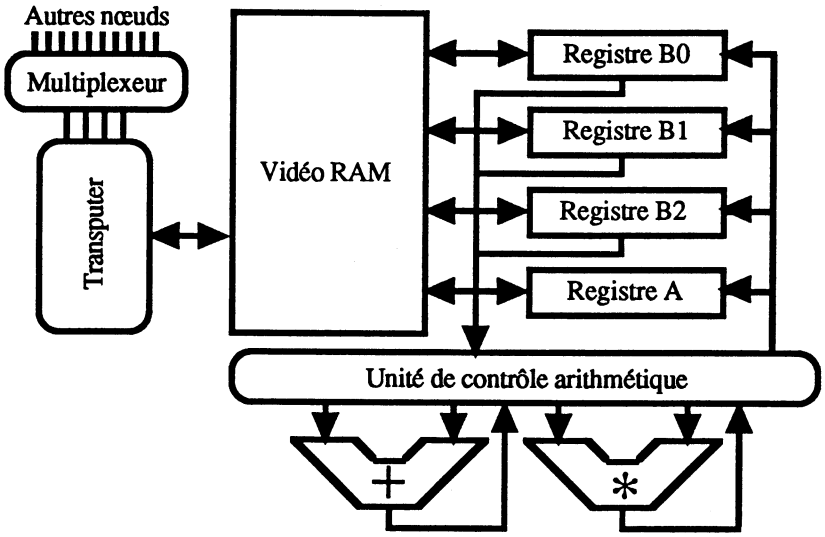


Figure 19 : schéma d'un nœud vectoriel

Un nœud système comprend, quant à lui, deux parties principales :

- un Transputer T414 assurant les diverses formes de communications (avec les processeurs vectoriels de son module et avec les autres nœuds systèmes)
- un disque dur Winchester d'une capacité de stockage de 85 Mo

Ce sont les nœuds systèmes qui assurent le chargement d'un programme : via le nœud système 0, chacun reçoit le code de la machine hôte et le charge sur les processeurs vectoriels de son module. Ainsi, tous les processeurs du T40 reçoivent le même code, mais ils connaissent leur numéro absolu dans l'architecture de la machine, et n'exécutent donc ainsi que la partie du code les concernant (à condition que celle-ci soit explicitement décrite par le programmeur).

3.2. Le Transputer

Chaque processeur du FPS T40 est constitué autour d'un Transputer IMS T414 qui est chargé des communications, de la gestion mémoire, des relations avec l'unité vectorielle (VPU) et d'un disque dur (uniquement dans le cas d'un nœud système). Le Transputer présente certaines caractéristiques intéressantes qu'il est bon de garder à l'esprit lorque l'on désire écrire un code efficace sur le T40. C'est le point de vue adopté dans les paragraphes suivants. De plus, ce composant est à l'origine de la plupart des machines parallèles distribuées qui sortent à l'heure actuelle. Notons cependant qu'il n'est pas nécessaire de le connaître à fond pour "bien" utiliser les machines parallèles construites à base de Transputers.

3.2.1. Introduction

Le Transputer est un micro-circuit, réalisé par INMOS qui permet la réalisation simple de systèmes multiprocesseurs (MIMD à mémoire distribuée). De plus, il comporte un automate micro-programmé (le "scheduler") implantant de façon totalement transparente la multiprogrammation. Le Transputer doit son succès à la cohérence de son modèle de programmation et à son très bon compromis performance-simplicité et performance-prix. Il a été conçu pour être programmé en Occam, langage de haut niveau issu de CSP qui est un formalisme de description de processus concurrents et communicants.
Le Transputer existe sous 4 formes : le T212 à processeur 16 bits, le T414 32 bits, le T800 32 bits et processeur scalaire flottant (FPU) et le très récent T9000. Le T212 n'est utilisé que marginalement (il ne dispose que de 64 Ko d'espace mémoire) pour des opérations trés simples, nous n'y ferons plus référence dans les pages qui suivent.

3.2.2. Architecture d'ensemble

Comme le schématise la figure suivante, le Transputer intègre en un seul circuit VLSI un micro-processeur 32 bits, un peu de mémoire rapide, une interface avec une mémoire externe et des dispositifs d'entrée/sortie.

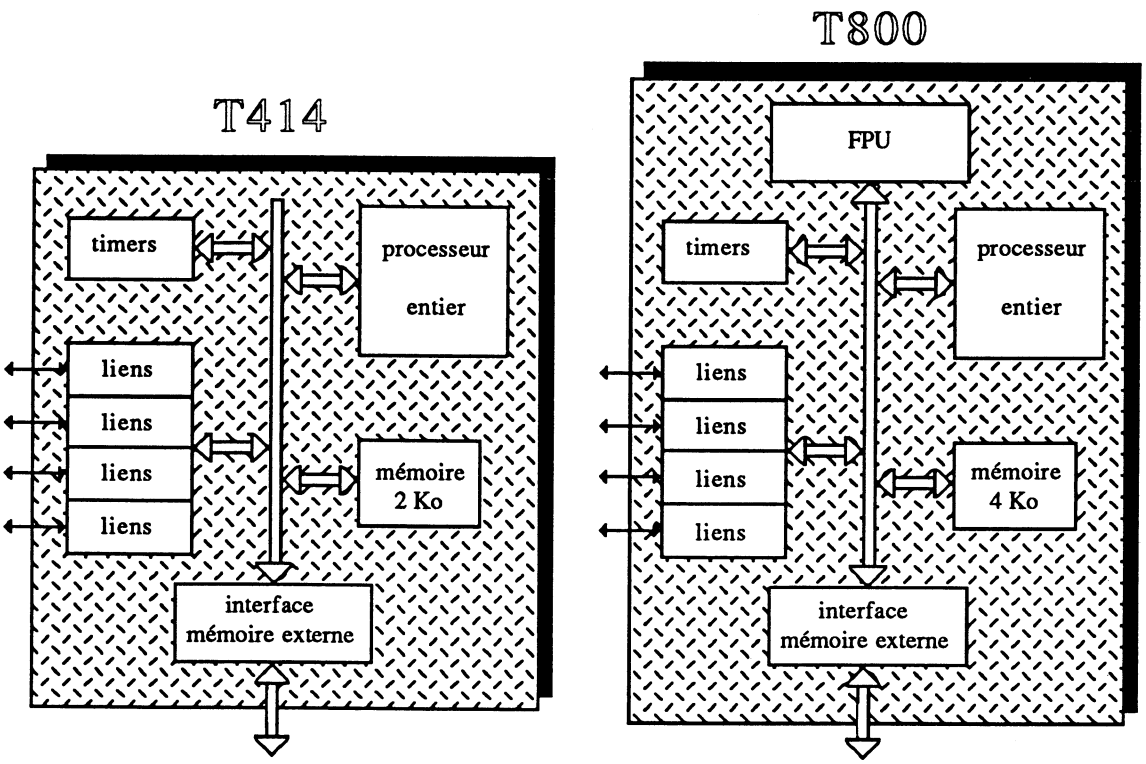


Figure 20 : schéma fonctionnel du T414 et du T800

Examinons brièvement un à un ces divers composants :

Processeur : il s'agit d'un processeur à architecture très simple, qui ne doit sa puissance qu'à sa rapidité et à l'efficacité de son jeu d'instruction (concept RISC) ; il est schématisé ci-dessous.

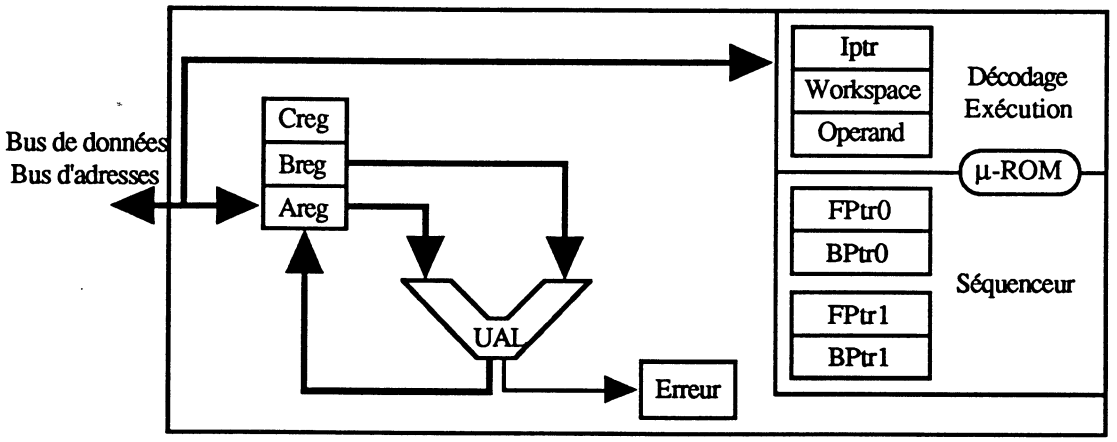


Figure 21 : le processeur du T414

Mémoires : le T414 possède 2 Ko de mémoire intégrée sur le circuit. Cette mémoire très rapide fonctionne à la vitesse du processeur et peut se comparer aux registres internes des micro-processeurs usuels. Une interface permet l'accès à une mémoire externe via un bus 32 bits de données et 32 bits d'adresse correspondant à 4 Go d'espace mémoire. L'accès à la mémoire externe est 3 fois plus lent qu'un accès à la mémoire interne, dans le meilleur des cas.

Liens : chaque Transputer dispose de 4 liens séries bidirectionnels. Un lien permet le transfert d'un bloc de données de la mémoire externe ou "on chip" d'un Transputer dans la mémoire d'un autre Transputer. Ces liens fonctionnent de façon autonome par l'intermédiaire d'un "Direct Memory Access" (DMA), la figure 22 représente un lien bidirectionnel.

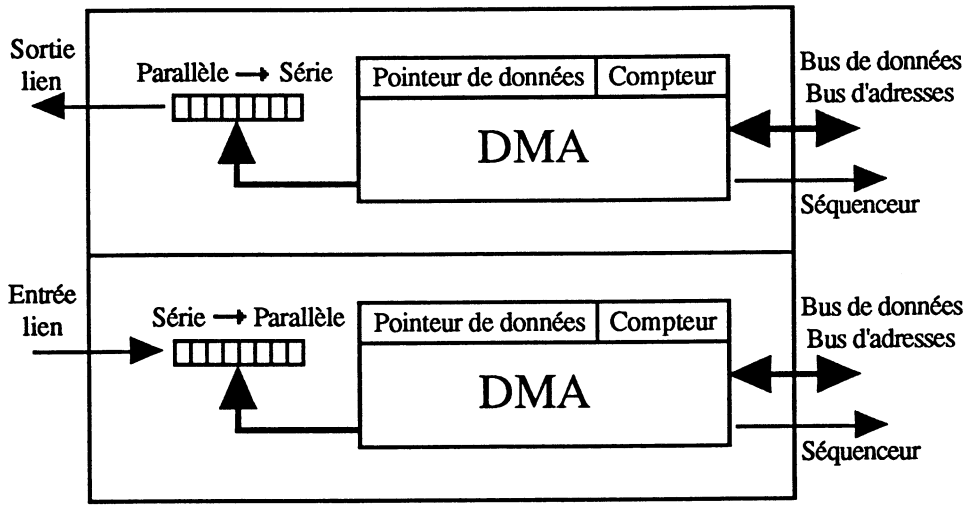


Figure 22 : un lien bidirectionnel

Timers : Le Transputer est équipé de 2 horloges cycliques 32 bits. L'une sert pour les processus de priorité 0 ainsi qu'au scheduler et a une résolution d'une micro-seconde. L'autre est l'horloge des processus de priorité 1 et s'incrémente toutes les 64 µs, soit 15625 "tops" par seconde. Ces horloges peuvent être lues par exemple pour des évaluations précises de temps de calcul (évaluations de performance) ou pour endormir un processus pour un temps prédéfini. Les deux timers sont schématisés sur la figure 23.

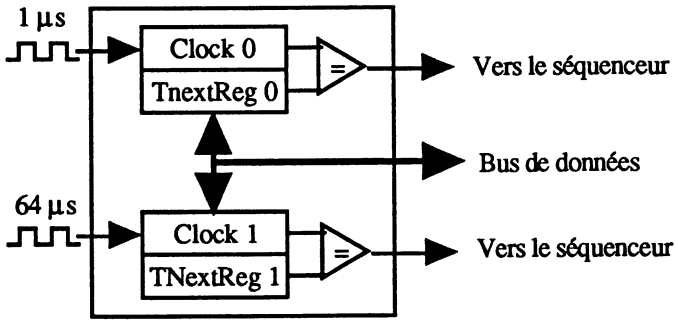


Figure 23 : les deux timers

3.2.3. Le scheduler

Une des originalités du Transputer réside dans le scheduler qui est un automate cablé et microprogrammé assurant la gestion des processus, du temps partagé et la réaction aux évènements extérieurs (fin de communication par exemple). Il implante la multiprogrammation Occam de façon transparente pour l'utilisateur.

Le scheduler maintient une liste des processus en attente. A chaque nouvelle tranche de temps CPU, le processus en cours d'exécution est endormi et placé en fin de liste, alors que le premier processus de la liste est réveillé pour être exécuté pendant la nouvelle tranche de temps CPU.

3.2.4. Registres et organisation de la mémoire

Tous les registres du processeur ont 32 bits (T414 et T800).

Registres de calcul :
l'UAL ne dispose que d'une pile de trois registres Areg, Breg et Creg. Cette pile sert aussi à recevoir les paramètres de certaines instructions non arithmétiques.

Registres systèmes :
- "Workspace" est un pointeur vers l'espace de travail local du processus courant.
- "Iptr" est le compteur ordinal : il pointe vers la prochaine instruction à éxécuter.
- "Operand" permet de construire des opérandes jusqu'à 32 bits par groupes de 4.
- "Error" est un bit d'erreur généré par l'UAL.

Registres du scheduler :
- FPtr0 et BPtr0 sont les pointeurs vers le début et la fin de la liste des processus actifs de priorité 0.
- FPtr1 et BPtr1 sont les pointeurs vers le début et la fin de la liste des processus actifs de priorité 1.

Organisation de la mémoire
Les adresses en mémoire sont comptées soit sur les octets de #80000000 (# indique une notation hexadécimale) à #7FFFFFFF, soit sur les mots de 32 bits à partir de 0. La mémoire interne se trouve aux adresses basses (word address# 0 à #1FF pour le T414 et #3FF pour le T800). Les 18 premiers mots sont réservés pour les liens, les pointeurs vers les listes de processus en attente et pour les sauvegardes des registres du processeur.

3.2.5. Jeu d'instruction et programmation

Les instructions du Transputer ont un format fixe sur 8 bits : 4 bits codent la commande, suivis de 4 bits pour l'opérande. On a donc 16 instructions *directes* et des opérandes compris entre 0 et 15, qui peuvent être étendus grâce au registre "operand".

Voici un survol par catégories du jeu d'instructions du Transputer :
- arithmétique entière : +, -, *, / et reste en arithmétique avec erreur de dépassement ou en arithmétique modulo
- arithmétique entière et logique en précision illimitée : +, -, *, /, décalages à gauche et à droite
- logique : and, or, xor, not, shift-left et shift-right
- manipulation de matrices
- entrées/sorties (canal Occam)
- timers
- contrôle d'exécution
- contrôle de process
- calcul flottant, par étapes sur le T414, direct sur le T800

Le langage "naturel" du Transputer est Occam. Ce langage permet d'exprimer très simplement des processus concurrents et leur communications.

3.2.6. Performances

L'architecture du type RISC (Reduced Instruction Set Computer) à code compacté mise en œuvre sur le Transputer lui confère une bonne rapidité d'exécution. De plus, la faible fréquence des accès mémoire pour lecture d'instructions (1 accès pour 4 instructions) laisse le bus disponible pour les accès aux données.
Le temps de cycle du T414-20 est de 50 ns. Les instructions les plus utilisées prennent 1 ou 2 cycles si elles n'accèdent qu'à la mémoire interne. Le débit d'instructions est alors de l'ordre de 10 MIPS (millons d'instructions par seconde). Les accès à la mémoire externe dépendent essentiellement du type de cette mémoire. En pratique, les performances sont entre 1,5 et 3 fois moins bonnes lorsque programme et données sont dans la mémoire externe.

Dans une optique d'efficacité maximale, les entrées-sorties par liens sont fortement couplées au processeur. En particulier, les instructions spéciales de communication en assurent le contrôle rapidement et simplement.
On peut sélectionner par hardware la vitese des liens parmi les valeurs 5, 10 et 20 Mbits/s.
Il faut noter que le T800 bénéficie d'un algorithme amélioré au niveau du protocole

d'échange qui augmente d'environ 20 % la vitesse de transfert.
Tout dispositif électronique (buffer ou multiplexeur) placé sur un lien dégrade fortement les performances. Ensuite, si deux transferts (un dans chaque sens) interviennent simultanément sur un même lien, la vitesse diminue. Enfin, lorsque plusieurs liens sont en service simultanément, un ralentissement du processeur est inévitable, dû à l'encombrement du bus.

Lorsque le programme et les données sont stockées dans la mémoire interne, la vitesse d'exécution en est très augmentée. Pour accélérer globalement un programme, il conviendrait donc de ranger les variables les plus utilisées et les sections de code les plus critiques en mémoire interne. A ce titre, un compilateur muni de directives de rangement ou d'un optimiseur serait un outil précieux. Une autre approche consisterait à ne pas utiliser la mémoire interne de façon statique, mais plutôt comme une mémoire cache entiérement gérée par le compilateur.

3.3. Les unités de calcul vectoriel du T40

Du fait de lenteur des communications sur les architectures MIMD à mémoire distribuée, l'existence d'unités vectorielles locales permet un parallélisme à grain assez gros, donc plus efficace. En effet, un grain plus fin ne semble pas être possible avec ce type de machines pour lesquelles les communications sont le point critique.

3.3.1. Présentation générale des unités vectorielles du T40

L'unité de calcul vectoriel Weitek du T40 permet le traitement efficace (grâce à un additionneur et un multiplieur pipelinés) de réels codés sur 64 bits au format standard IEEE double précision.
Trois niveaux de programmation du VPU permettent de réaliser différemment le compromis entre facilité de programmation et performances élevées.

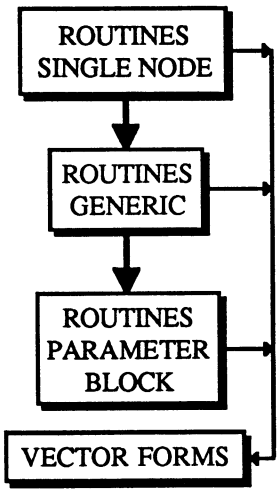


Figure 24 : les différents niveaux de programmation

Le niveau le plus élevé de programmation du VPU est constitué par les routines "single node" qui permettent d'effectuer des opérations complexes sur des vecteurs, comme par exemple les fonctions trigonométriques et l'exponentielle.
Le niveau intermédiaire, qui allie une relative facilité de programmation avec un bon niveau de performances, est celui des routines "generic" qui permettent d'effectuer sur des vecteurs la plupart des opérations simples : addition, multiplication, combinaison entre les deux etc..
Au plus bas niveau, le programmeur doit utiliser les "parameter blocks" qui permettent d'invoquer les "vector forms" (instructions microcodées qui commandent directement les pipelines). Ce niveau, qui implique des contraintes de programmation analogues à celles d'un assembleur, est celui qui permet d'atteindre les performances les plus élevées.

La figure ci-dessous présente les diverses composants (mémoire, registres à décalage, unité de contrôle arithmétique et pipelines) influant sur les performances de l'unité vectorielle.

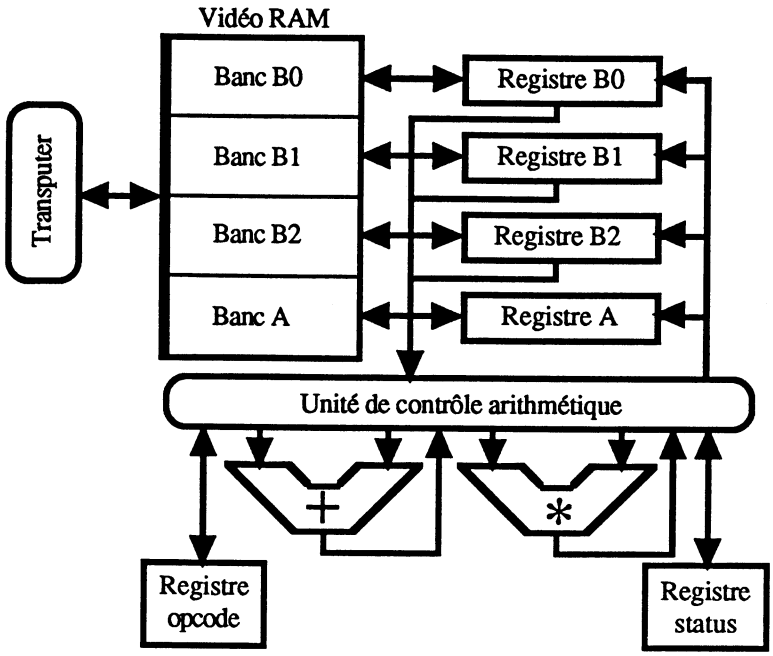


Figure 25 : schéma d'un nœud vectoriel

3.3.2. Les différents composants du VPU

3.3.2.1. La mémoire locale sur chaque nœud

La mémoire est le premier élément qui entre dans la composition du VPU, et elle est essentielle, car c'est souvent dans une architecture un "goulot d'étranglement" qui dégrade les performances.
Chaque processeur dispose donc de 1 méga-octet de mémoire vive rapide, accessible à la fois par le Transputer et le VPU. Elle est adressable par mots de 32 bits. La mémoire est divisée en 4 bancs de taille égale, chacun étant associé à un registre vectoriel à décalage. La mémoire principale est du type vidéo RAM et son accès est double : on peut accéder depuis le Transputer (accès aléatoire classique) au contenu de n'importe quel mot, l'accès depuis le VPU (accès série) se fait en chargeant une tranche mémoire complète dans l'un des registres vectoriels à décalage.

Les 4 bancs sont numérotés A, B2, B1, B0, les adresses allant de 00000 à FFFFF. Les extrémités de la mémoire (bas du banc B0 et sommet du banc A) sont réservées au système qui les utilise lors des appels des routines "generic" ou "single node". Le code du programme est chargé dans la partie basse du banc B0 (au dessus de la tranche réservée au système).

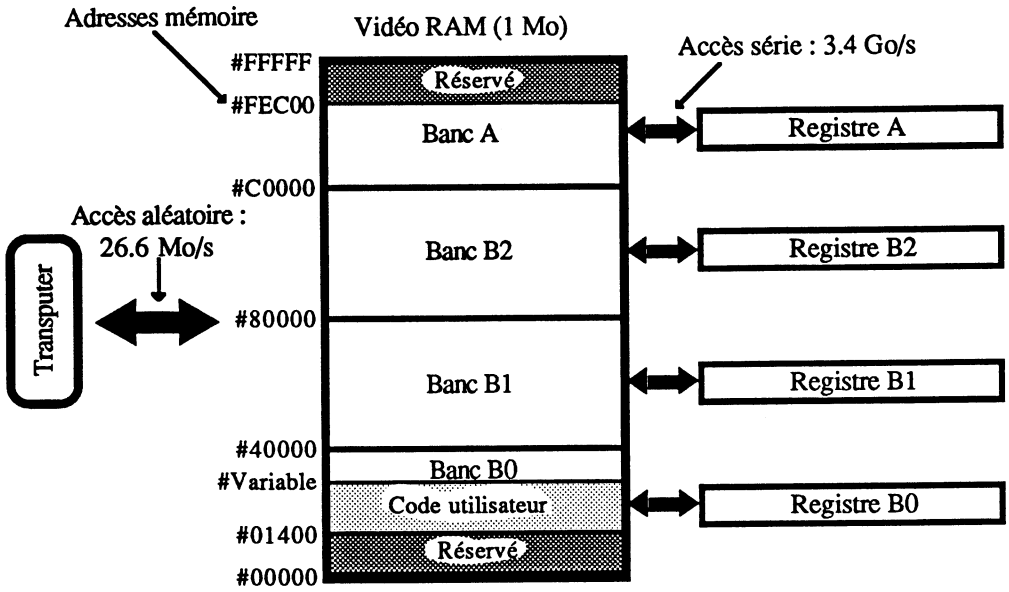


Figure 26 : la mémoire de chaque nœud vectoriel

Le passage des paramètres pour les routines du VPU se fait par adresse pour les vecteurs et par valeur pour les scalaires.

L'accès aléatoire (i.e. vis à vis du Transputer) se fait en 3 cycles du VPU (soit 300 ns) pour 4 octets, ce qui correspond à 26.66 Mo/s. Le jeu d'instructions du Transputer permet d'effectuer des transferts de blocs ou d'éléments à incrément régulier dans la mémoire.
L'accès série se fait via les registres vectoriels. A chaque banc de la mémoire est associé un registre. Les bancs sont découpés en 256 tranches de 1024 octets ; lors des accès série (en lecture ou en écriture) une tranche complète de 1024 octets est chargée (respectivement déchargée) dans un registre (respectivement dans la mémoire). La vitesse de transfert est importante : une tranche de 1024 octets est transmise en 3 cycles du VPU (300 ns), on a donc un débit de 3.4 Go/s.

3.3.2.2. Les registres vectoriels à décalage

Ils constituent l'interface entre la mémoire vidéo et l'unité de calcul. L'accès série est considérablement plus rapide que l'accès aléatoire pour alimenter les registres. Une fois dans les registres vectoriels, les données sont transférées vers l'unité de calcul à partir de l'une des 4 sources (il y a 32 réels entre chaque source). Ensuite les données sont décalées à chaque fois que l'une d'entre elles est communiquée à l'unité de calcul ; de même, les résultats sont rangés par l'arrière du registre : c'est le principe des registres à décalage.

3.3.2.3. L'unité de calcul

Elle consiste en un additionneur et un multiplieur pipelinés Weitek à 6 et 7 étages respectivement (voir figure 27). Elle possède 2 entrées, l'une à partir du registre A et l'autre à partir des registres B. La sortie est dirigée vers les 4 registres.
L'additionneur et le multiplieur ont chacun 2 entrées couplées sur les registres A et B (voir figure ci-dessous). L'enchainement des opérations (sortie d'un pipeline en entrée de l'autre pipeline) est, quant à lui, limité :

- pour l'additionneur : une entrée à partir du registre A ou de sa propre sortie, et l'autre à partir d'un registre B ou de la sortie du multiplieur,

- pour le multiplieur : une entrée à partir d'un des registres B ou de sa propre sortie, et l'autre à partir du registre A ou de la sortie de l'additionneur

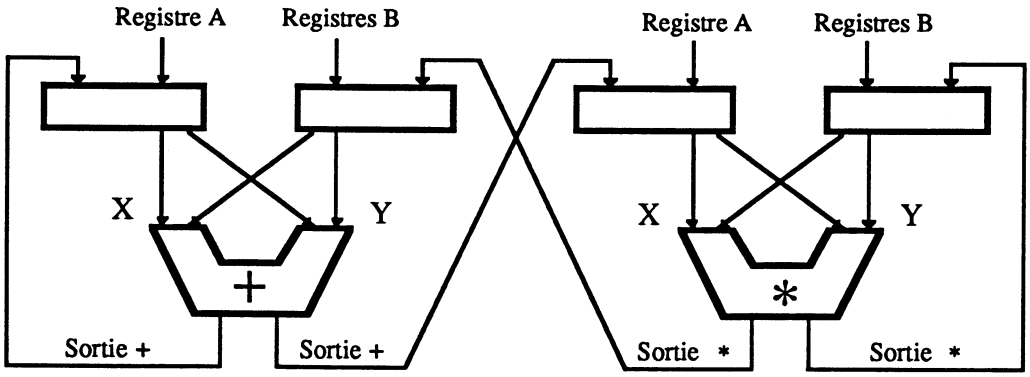


Figure 27 : chaînage des unités vectorielles

Il est possible à l'unité de calcul vectoriel de fonctionner simultanément avec le Transputer, lorsque celui-ci exécute des communications ; les deux unités communiquent via des interruptions et les registres scalaires "opcode" et "status". Le Transputer déclanche une opération vectorielle en écrivant son code dans le registre "opcode", le status de l'opération terminée est retourné par le VPU dans le registre "status" (ce qui permet la prise en compte des erreurs). Notons qu'un des canaux de sortie du Transputer est réservé pour la gestion des erreurs du VPU.
Lorsqu'un programme appelle des routines de calcul vectoriel, il se crée un "processus vectoriel" qui est exécuté au plus haut niveau de priorité du Transputer. Il faut en effet assurer :

- la gestion des interruptions, un message en provenance du Transputer ne pouvant interrompre les pipelines en cours d'opérations,
- le chargement et le déchargement des registres vers la mémoire en un temps fixé (les éléments de mémorisation du coté VPU sont dynamiques)

3.3.3. Exemple de programmation du VPU

Voyons maintenant sur un exemple concret, le calcul du produit scalaire de deux vecteurs, comment utiliser le VPU à l'aide des différents niveaux de programmation disponibles (l'exemple traité est écrit en langage C).
Un produit scalaire sur des vecteurs de taille n se présente classiquement sous la forme suivante :

```
Scal = 0.0;
pour i=1 jusqu'à n
   Scal=Scal+X(i)*Y(i);
```

On suppose que les données sont placées de la façon suivante en mémoire :

- le vecteur `X` est dans le banc A.
- le vecteur `Y` est dans le banc B.
- le réel double précision `Scal` est dans le banc A

Le plus haut niveau de programmation du VPU est consitué par les fonctions "single node" ; une fonction est spécialement dédiée au produit scalaire :

```
dotpr(&X[0],1,&Y[0],1,&Scal,n);
```

Les deux vecteurs sont passés par adresse, ce qui est le cas pour tous les niveaux de programmation. Il en est de même du scalaire, mais cela est propre à certaines fonctions "single node". Les deuxième et quatrième paramètres précisent le pas (stride) des vecteurs ; il faut noter que dès que cette valeur n'est pas prise ègale à 1 (éléments consécutifs), les performances obtenues décroissent trés nettement.

Intéressons nous maintenant aux fonctions "generic". Celles-ci adoptent un mnémonique en notation postfixée : le produit scalaire est une opération du type ((vecteur-vecteur opération)-réduction opération), ce qui correspond à la routine générique VVORO, les opérations effectuées étant fournies dans les paramètres de la fonction. La boucle écrite précédemment s'écrit donc ainsi en fonction "generic" :

```
VVORO(&X[0],&Y[0],MF_XYMUL,AF_XYADD,Scal,n);
```

On remarque que les deux vecteurs sont passés par adresse et que le scalaire est passé par valeur. Le type d'opérations effectuées est fourni par les deux constantes `MF_XYMUL` (multiplication vectorielle) et `AF_XYADD` (addition vectorielle).

Cette routine s'écrit de la manière suivante avec des "parameter blocks" (en langage C) :

```
VP_BLK0[VP_OP_CODE]=VF_ABMULRADD|VP_func(AF_XYADD)|
   VP_func(MF_XY_MUL)|128;   (1)
VP_BLK0[VP_SLOAD]=VF_ALOAD1;   (2)
VP_BLK0[VP_SRCMSK]=VP_SV1V2;   (3)
VP_BLK0[VP_SRCS]=&Scal;   (4)
VP_BLK0[VP_SRCV1]=&Y[0];   (5)
VP_BLK0[VP_SRCV2]=&X[0];   (6)
pb_start(VP_BLK0);   (7)
VP_BLK1[VP_OPCODE]=VF_RBODY|VP_func(AF_XYADD)|
   VP_func(MF_XY_MUL)|128;   (8)
VP_BLK1[VP_DSTMSK]=VP_NODST;   (9)
P_BLK1[VP_SRCMSK]=VP_V1V2;   (10)
VP_BLK1[VP_DSTV1]=VP_BLK0[VP_SRCV1] + 1024;   (11)
VP_BLK1[VP_DSTV2]=VP_BLK0[VP_SRCV2] + 1024;   (12)
for (i=2;i<=n/128;i++)   (13)
   {
    pb_cont(VP_BLK1);
    VP_BLK1[VP_SRCV1] += 1024;
    VP_BLK1[VP_SRCV2] += 1024;
   }
VP_BLK2[VP_OPCODE]=VF_RCOLLAPSE|VP_func(AF_XYADD)|
    VP_func(MF_XY_MUL); (14)
VP_BLK2[VP_DSTMSK]=VP_NODST;   (15)
VP_BLK2[VP_SRCMSK]=VP_NOSCR;   (16)
pb_cont(VP_BLK2);   (17)
VP_BLK2[VP_DSTMSK]=VP_D1;   (18)
VP_BLK2[VP_DSTV1]=&Z[0];   (19)
pb_finish(VP_BLK2);   (20)
```

Explicitons pas à pas les différentes étapes de ce calcul. En (1) on charge le code opérations du "parameter block 0". Ce code précise que l'on va réaliser la multiplication des éléments d'un vecteur se trouvant dans le banc B par les éléments d'un vecteur se trouvant dans le banc A et que l'on va faire la réduction du vecteur résultat.
A l'étape (2) on indique au "parameter block 0" par l'intermédiaire du "vector form" `ALOAD1` qu'on va utiliser un scalaire se trouvant dans le banc A. Il faut auparavant initialiser ce scalaire à 0. L'unique but de ce scalaire est d'initialiser l'additionneur.
Les instructions de (3) à (7) permettent de spécifier le nombre de sources, leurs adresses, ainsi que de lancer l'exécution du "parameter block 0".
L'instruction numéro (8) permet de charger le code opération du "parameter block 1". Ce code précise que l'on va continuer la réduction du vecteur. L'utilisation de ce "parameter block" est inutile lorsqu'on travaille sur des vecteurs de taille inférieure ou égale à 128

(largeur d'une tranche de mémoire, en réels double précision).
Les étapes de (9) à (12) permettent de spécifier les paramètres d'exécution du "parameter block 1".
En (13), on itére le procédé suivant : on exécute le "parameter block 1", et on charge les adresses des tranches suivantes. Ceci se répète autant de fois qu'il y a de tranches de 128 éléments dans les vecteurs `X` et `Y`.
L'étape (14) permet de charger le code opération du "parameter block 2". Ce code opération indique que l'on va regrouper les sommes partielles en un seul scalaire que l'on déchargera par la suite dans les quatre registres vectoriels. Les registres étant justifiés à droite.
Les instructions (15) et (16) indiquent que lors du regroupement des sommes partielles ils n'y pas de source ni de destination, c'est-à-dire que les sources de l'additionneur est contenu dans le registre en sortie du multiplieur. Le résultat est remis dans le registre A (une des sorties de l'additionneur).
En (17) on commence l'exécution du "parameter block 2".
Les étapes (18) et (19) précisent l'adresse où sera mis le résultat.
Enfin, en (20) on décharge le résultat dans la mémoire. Le résultat du produit scalaire se trouve alors à l'adresse `&Z[127]`. Il est possible de ramener ce résultat à l'adresse `&Z[0]` en utilisant un quatrième "parameter block" qui aurait pour code le "vector form" `VF_ABSHIFT` (cette opération permet de décaler le contenu des registres vectoriels).

Il faut noter que l'exemple donné ci-dessus ne peut être utilisé que sur des vecteurs dont la taille est un multiple de 128. Si l'on désire travailler sur un vecteur de taille quelconque, il est nécessaire d'utiliser des "vector forms" supplémentaires.
Pour comprendre ce qu'il faut faire dans ce cas, regardons sur un exemple comment s'organise le vecteur dans la mémoire. Prenons arbitrairement une taille égale à 270 ; ce vecteur se place en mémoire de la façon suivante :

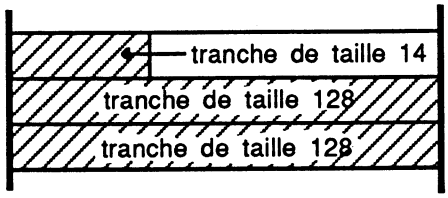


Figure 28 : placement d'un vecteur en mémoire

En effet 270 se décompose en 2*128 + 14. Il faut donc un "vector form" qui réalise le calcul sur la tranche de vecteur de taille 14 et un second qui réalisera le décalage, dans le registre vectoriel, de 114 positions (128-14).

3.3.4. Contraintes de programmation et performances

Le but de ce paragraphe est de décrire les contraintes que doit respecter l'utilisateur lorsqu'il utilise le VPU et d'indiquer quelques manipulations qui permettent d'obtenir des performances en vectoriel aussi élevées que possible.
Les remarques permettant d'optimiser le code en vue de son traitement par le VPU concernent principalement la mémoire. Le but de ces remarques est d'éviter aux routines de haut niveau d'avoir à effectuer un travail préparatoire avant le traitement proprement dit et au programmeur d'écrire des lignes supplémentaires de "parameter blocks" pour préparer ses données.

La contrainte la plus absolue concerne le placement des données en mémoire : seules les fonctions "single node" traitent le cas de variables à cheval sur deux bancs (et encore : les performances sont extrémement faibles), les fonctions "generic" et les "parameter blocks" rendent un code d'erreur lorsqu'on les fait exécuter sur de telles variables. La figure suivante donne un exemple de variable à cheval sur deux bancs : A et B2.

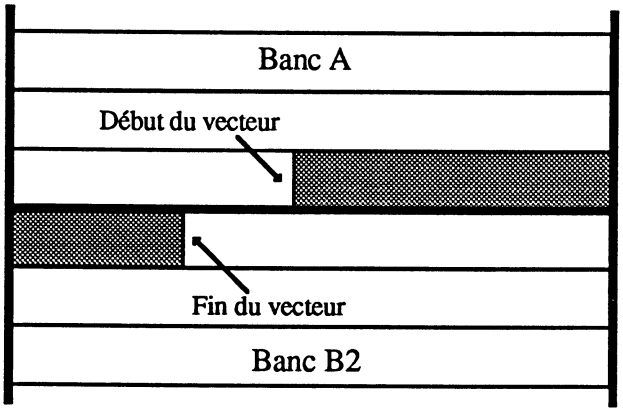


Figure 29 : variable à cheval sur deux bancs

Seules les fonctions "single node" permettent d'effectuer des calculs sur des éléments de vecteurs non contigus (avec des performances trés faibles cependant). Il faut donc écrire des algorithmes où ce type d'opération est bani si l'on désire utiliser avec profit le VPU.
Les deux opérateurs pipelinés (additionneur et multiplieur) ayant chacun une entrée connectée au banc A et l'autre aux bancs B, il faut que les données soient disposées suivant ces contraines : un opérande dans le banc A et l'autre dans l'un des bancs B (B0, B1 ou B2). Donc cela posera, d'une certaine manière, des problèmes pour utiliser la pleine capacité de la mémoire.
Les registres à décalage fonctionnant de droite à gauche, les données doivent être calées à gauche en début d'opération, cela signifie que si les données ne sont pas en début de tranche mémoire, il faudra les déplacer avant de les envoyer dans les registres à décalage. Ce cas est illustré sur la figure suivante.

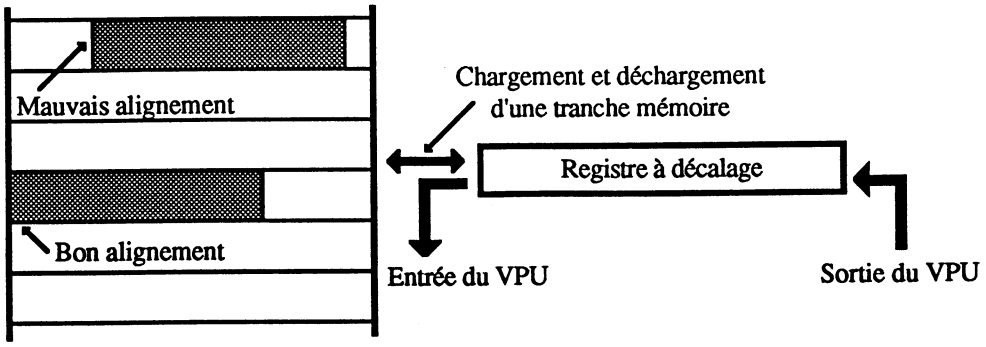


Figure 30 : alignement des données en mémoire

Remarquons pour terminer que cette unité est trés simple : elle ne comporte pas, par exemple, de diviseur.

Nous donnons en 4.1.1 une étude complète des performances qui souligne les gains considérables que l'on peut réaliser en utilisant de façon appropriée une unité vectorielle. Ce paragraphe montre la difficulté d'utilisation d'une unité vectorielle pour un programmeur non spécialisé. Un bon comportement devant ce style de machines consiste à développer des procédures standards avec une bonne gestion de la mémoire pour les utiliser de façon transparentes le plus efficacement possible avec le minimum de contraintes.

3.3.5. Algorithme de division vectorielle

On a vu en 3.3.2.3 que trois des quatre opérations arithmétiques de base sont directement accessibles au plus bas niveau de programmation du VPU : addition, soustration et multiplication. Par contre, la division n'est vectorisable qu'au niveau des fonctions singlenode, et donc beaucoup moins performante. Lors des expérimentations pratiques réalisées sur le FPS T40 (implantatations du gradient conjugué préconditionné décrites en 5.1 et 5.2), il est apparu que les performances obtenues pouvaient être améliorées de façon non négligeable en apportant un soin plus grand à la vectorisation de la division.

L'algorithme testé est très classique, il s'agit de l'itération de Newton suivante :

$$x_{n+1} = x_n(2-ax_n)$$

où a est le nombre dont on cherche l'inverse. La figure ci-dessous représente graphiquement l'obtention de l'inverse par cet algorithme.

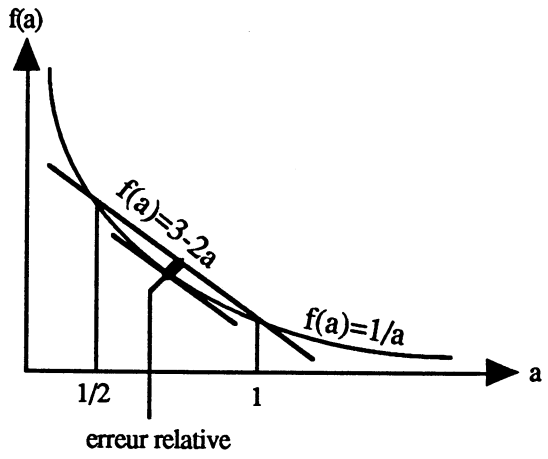


Figure 31 : l'algorithme de Newton pour la division

Le seul point délicat de cette méthode aplliquée au calcul vectoriel réside dans l'initialisation : pour que l'itération de Newton converge il faut que x_0 soit déjà une *pas trop mauvaise* approximation de la solution. On utilise pour ce faire le format de codage des réels : x_0 est pris égal à un nombre dont la puissance est celle de l'inverse de a et la mantisse celle de 3-2a. Il s'agit d'une bonne approximation qui minimise l'erreur relative et assure la convergence de la méthode en 4 itérations (pour des réels codés au format IEEE double précision). Par conséquent la valeur initiale choisie (en tenant compte des remarques précédentes) est :

$$x_0 = 7FDE0000 - a$$

Notons dès à présent qu'une telle initialisation est imposssible à effectuer en C sur les unités vectorielles du T40. Il faudrait pour cela avoir directement accès aux primitives du VPU Weitek, ce qui n'est pas possible sur le T40.

Les résultats expérimentaux sont les suivants (pour 128 réels au format IEEE double précision) :

- division *singlenode* : 0.768 ms
- méthode de Newton en *vector forms* : 2.944 ms dont 2.32 pour l'initialisation scalaire
- méthode Newton optimisée pour le T40 en *vector forms* : 1.54 ms dont 1.28 ms pour l'initialisation scalaire

Les optimisations de la seconde version en *vector form* portent à la fois sur les parties scalaires et vectorielles :

- en scalaire : on écrit la soustraction à l'aide des opérateurs C sur les bits

- en vectoriel : les calculs sont chaînés, et lorsque ce n'est pas possible, les résultats intermédiaires sont gardés dans les registres à décalage (au lieu d'être rangés en mémoire)

Il apparaît donc que, pour peu que l'on dispose de primitives vectorielles permettant d'effectuer l'initialisation, et que l'on optimise minutieusement le code, il est possible d'obtenir d'excellents résultats : le temps de calcul de l'inverse (hors initialisation problèmatique) est de 0.26 ms, contre 0.77 ms pour les fonctions *singlenode*.

3.4. Les communications

3.4.1. Principes généraux

Les ordinateurs FPS de la série T communiquent par échanges de messages entre processeurs.

La gestion des communications est assurée sur chaque nœud par le Transputer. Celui-ci possède 4 canaux de communications bi-directionnels (émission et réception en parallèle). Afin d'augmenter le degré de parallélisme du système, ces 4 sorties ont été multiplexées avec des éléments à 4 sorties. Ainsi, comme schématisé sur la figure suivante, chaque processeur dispose donc de 16 canaux de communications logiques, cependant seuls 4 d'entre eux peuvent être utilisés simultanément (un multiplexeur ne pouvant avoir qu'une seule de ses entrées actives).

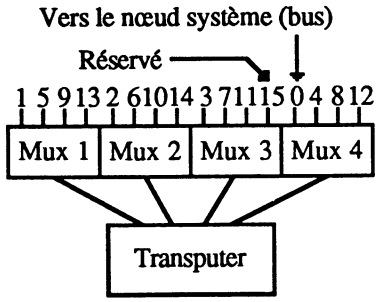


Figure 32 : multiplexage des liens physiques du Transputer

La première chose à faire lorsqu'on veut effectuer une communication entre deux processeurs ou plus, est de repérer les processeurs émetteurs et récepteurs. La façon la plus classique pour repérer un nœud dans un hypercube est d'utiliser les codes de Gray : deux processeurs sont "physiquement" voisins lorsque leur écriture en binaire diffère d'un seul chiffre (code de Gray), comme le montre la figure ci-dessous sur un 4-cube.

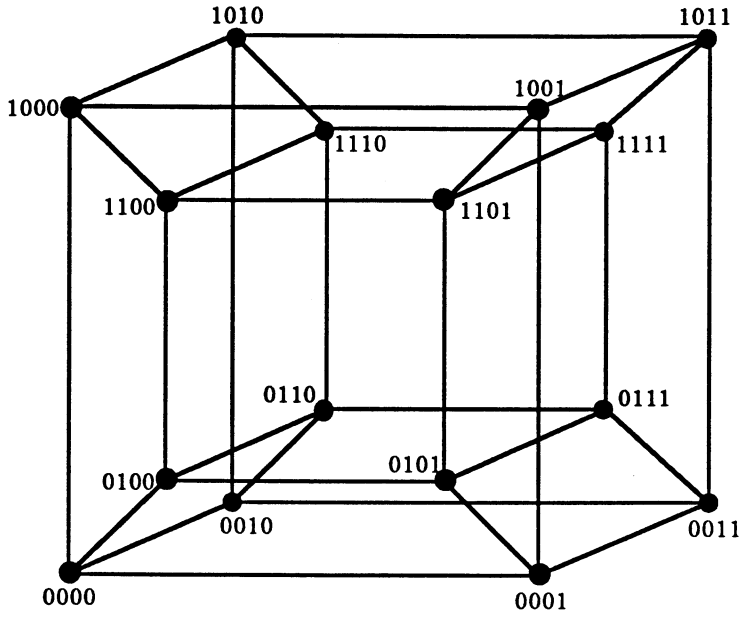


Figure 33 : codes de Gray d'un hypercube de dimension 4

Les différents types de communications (entre processeurs, entre processeurs et nœud système, entre nœud système et ordinateur hôte) disponibles sur le T40 ne se déroulent pas de la même façon, elles sont détaillées par type ci-après.

3.4.2. Communications entre processeurs

Ce type de communications est le plus couramment utilisé dans la pratique ; il sert essentiellement à faire communiquer des données entre les processeurs.

3.4.2.1. Description du fonctionnement

Une communication entre nœuds vectoriels s'effectue en trois étapes :

- choix la topologie
- définition les canaux de communication
- appel des fonctions de communication

Sur le FPS T40, la configuration logique de la machine dans une topologie n'est pas définitive : on peut en changer en cours de programme. FPS fournit trois configurations standards que l'on peut mettre en œuvre par l'appel d'une fonction :

- tore à une dimension (anneau), "config_torus_1d"
- tore à deux dimensions (grille de révolution), "config_torus_2d"
- hypercube, "config_hyp"

L'exemple ci-dessous configure le T40 en hypercube de degré 5 :

```
config_hyp(5,&Hyp_td)
```

`Hyp_td` est un identificateur qui fera référence à la topologie choisie (cf ci-dessous), ici un hypercube.
Il est possible de connaître (par l'appel de la fonction `config_pos`) l'emplacement relatif d'un processeur dans la topologie considérée. Cela permet par la suite de trouver, par exemple, le numéro de ses voisins. Le choix d'une topologie de configuration permet en effet d'effectuer une correspondance entre les numéros absolues des processeurs et leurs numéros relatifs dans cette topologie, plus facilement exploitables par l'utilisateur.

La définition des canaux de communication permet de choisir le type de communication, si l'on veut une communication blocante ou non (le processeur doit-il attendre la fin de la communication avant de continuer à travailler, ou peut-il le faire simultanément). On peut aussi choisir, avec une incidence notable sur les performances, le nombre maximum de communications pouvant se faire en parallèle.
Poursuivons l'exemple précédent en préparant une communication entre deux processeurs voisins :

```
LinkDescr = open_l (Hyp_td, 4, OTOX)
```

`LinkDescr` est un descripteur de fichier de liens qui permet de spécifier les communications. Le premier paramètre est le descripteur de topologie `Hyp_td` fourni par la fonction "config_hyp" précédemment invoquée, le second argument spécifie le nombre maximal de communications que l'on peut effectuer en parallèle, et le dernier argument précise le type de communications que l'on va effectuer (ici "OneToOne" : entre deux processeurs voisins).

Il existe différentes procédures de communication permettant chacune d'atteindre un nombre divers de processeurs. On peut ainsi communiquer :

- `oto_x` : d'un processeur à un autre processeur, permet de communiquer entre deux processeurs voisins ou entre deux processeurs non voisins mais appartenant à la même dimension d'un tore
- `ota_ld` : d'un processeur à tous les autres, ne fonctionne que pour des processeurs appartenant à la même dimension d'un tore
- `ato_ldo` : de tous les processeurs à un seul d'entre eux, permet une communication ordonnée dans une dimension d'un tore (les messages arrivent dans l'ordre dans lequel les processeurs sont définis dans cette dimension)
- `ata_ldo` : de tous les processeurs à tous les autres, permet une communication ordonnée dans une dimension d'un tore (les messages arrivent dans l'ordre dans

lequel les processeurs sont définis dans cette dimension)
- `ata_ldu` : n'ordonne pas les messages mais est plus rapide que la précédente
- `rota_ld` : décale les données d'un nombre donné de processeurs dans une dimension fixée d'un tore
- `swap_x` : échange des données entre deux processeurs de la même dimension d'un tore

Reprenons à nouveau l'exemple précédent, et faisons communiquer deux processeurs voisins dans l'hypercube :

```
oto_x(Lfd,2,0,1,&Depart,&Arrivee,NbOctets)
```

`Lfd` est le descripteur de fichier de liens obtenu via le `open_l` précedent. Le second argument définit la dimension dans laquelle la communication va avoir lieu, les troisième et quatrième paramètres sont les numéros des processeurs de départ et d'arrivée dans cette dimension. Les deux paramètres suivants sont les adresses de départ et d'arrivée des données à transmettre. Le dernier paramètre est le nombre d'octets transmis. Ici, on communique donc du processeur - - 0 - - (troisième dimension de bit égal à 0) au processeur - - 1 - - (troisième dimension de bit égal à 1).
Notons qu'il faut que le processeur recevant une communication en soit informé : son code doit aussi comporter la fonction de communication.

3.4.2.2. Restrictions d'utilisation

Le routage (communication entre deux processeurs non voisins) n'a été prévu par FPS que dans une dimension d'un tore. Avec des restrictions d'utilisation : on ne peut masquer les communications aux processeurs de la direction du routage.
Ainsi, lorsqu'on veut établir une communication entre deux processeurs n'appartenant pas à la même dimsension d'un tore, ou lorsque la topologie choisie n'est pas un tore, il faut construire un chemin qui se déplace de proche en proche parmi les processeurs voisins.

Remarquons enfin que l'utilisateur n'a pas à sa disposition de procédures de communications élaborées : diffusion (un processeur communique un vecteur à tous les autres) ou répartition (chaque processeur reçoit un vecteur, différent de celui des autres, tous en provenance d'un seul vecteur émetteur) par exemple.
Il lui faut donc soit les construire à partir des communications globales de FPS : `ota` (pour OneToAll : un processeur communique avec tous les autres dans une dimension d'un tore), soit les écrire de toutes piéces à partir des fonctions FPS `oto`.
La construction de fonctions de communications élaborées à partir de celles fournies par FPS pose un problème important : comme l'utilisateur n'a pas accès au micro-code, ses communications seront donc très lentes. Une alternative intéressante à ce problème consisterait à utiliser des processeurs spécialisés dans le routage.

3.4.2.3. Exemple de communications

L'algorithme proposé en exemple permet les rencontres de données sur tous les processeurs suivant des "perfect shuffle" successifs. Cet algorithme se retrouve dans de nombreux problèmes tels que les tris, la FFT (transformée de Fourier rapide) etc ...
Le principe permettant les rencontres de données pour un 3-cube est illustré sur la figure ci-dessous.

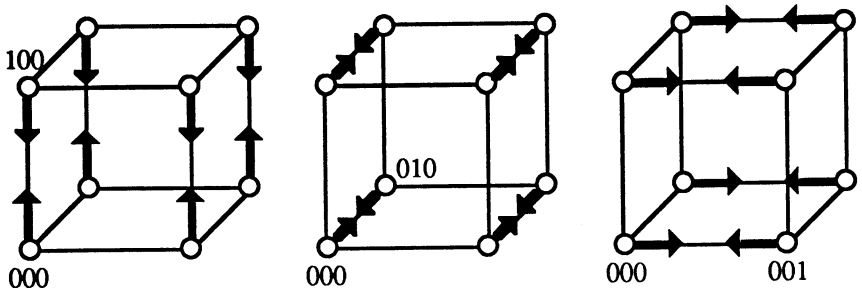


Figure 34 : échanges synchrones sur un 3-cube

L'algorithme est donné ci-aprés en pseudo-code :

```
DimHyp=5
/* initialisations */
config_hyp(5,&Hyp_td)
LinkDescr=open_link(Hyp_td,2,OTOX)
getinfo(&Proc,&Proc,&NbProc)
/* première phase : calculs locaux */
k=Proc
... calculs locaux ...
/* communications */
pour k=0 jusqu'à DimHyp
   /* échange des valeurs de a et b */
   oto_x(LinkDescr,k,0,1,&a,&b,Long)
   oto_x(LinkDescr,k,1,0,&a,&b,Long)
   wait_1(LinkDescr)
   si Proc=k
      ... calculs locaux ...
```

La phase d'initialisation configure le T40 en hypercube de degré 5, on obtient ensuite un descripteur de fichier de liens pour deux communications parallèles entre processeurs voisins, et enfin chaque processeur prend connaissance de sa position dans la topologie hypercube.
On peut ensuite effectuer des calculs locaux, qui varieront suivant l'application programmée (par exemple, pour un tri on effectuera un quicksort entre les données "a" et

"b" qui viennent d'être échangées)
La phase de calculs globaux consiste en l'échange des données suivant le schéma de la figure précédente : une fonction pour le processeur émetteur et une pour le processeur récepteur, la routine `wait_l` permet la synchronisation des communications.

3.4.3. Communications externes

3.4.3.1. Avec l'ordinateur hôte

Les communications entre les 8 processeurs d'un cube et le nœud système associé sont assurées par un bus, et sont donc mono-utilisateur (l'accès au bus est géré par un arbitre). Comme précédemment, il est nécessaire d'établir les connexions physiques entre le multiplexeur et le bus, cela est réalisé par une procédure prédéfinie : *keep_h*. Il existe une autre procédure qui, utilisée en fin de communication, libère l'accès au bus : *release_h*.

Toutes les entrées-sorties à l'écran passent par l'ordinateur hôte (et donc par le nœud système 0) et sont effectuées via les fonctions standards du langage C.

3.4.3.2. Entre les nœuds systèmes et les processeurs

Chacun des nœuds systèmes possède un disque dur Winchester où il est possible pour l'utilisateur de stocker des fichiers (pourvu que ceux-ci soient au format adéquat).
On dispose ainsi des classiques procédures de traitement des fichiers : ouverture (*open_f*), lecture (*read_f*), écriture (*write_f*) et fermeture (*close_f*) du fichier.

Chapitre 4

Architecture à mémoire distribuée : analyse informatique

Où l'on présente une analyse des différentes composantes (calculs et communications) des programmes sur une machine MIMD à mémoire distribuée dans l'optique de guider l'utilisateur dans l'écriture de programmes "efficaces" sur une architecture de ce type (un exemple est donné avec le produit matrice-vecteur). Ce chapitre est appliqué au cas du FPS T40. Un algorithme original de diffusion asynchrone est présenté et expérimenté en fin de chapitre.

4.1. Méthodologie

Dans cette section, les performances des machines MIMD à mémoire distribuée (la machine cible étant le FPS T40 présenté au chapitre 3) sont analysées du point de vue de l'utilisation des procédures de bas niveau du VPU (de façon à proposer un modèle théorique des temps d'exécution) et des communications. Le modèle théorique est très proche des expérimentations pratiques et permet de prédire les temps d'exécution avec moins de 1% d'erreur.

4.1.1. Les calculs vectoriels

Nous allons dans ce paragraphe proposer des modèles pour les calculs vectoriels effectués sur le VPU. Ces modèles tiennent compte des spécificités de la machine qui ont été décrit précédemment.

Les performances du VPU sont exprimées en millions d'opérations flottantes par secondes (Mflops). Cette unité de mesure sert de référence pour comparer différentes machines.
Malgré cela, cette unité de mesure est à utiliser avec précaution. En effet la notion d'opération n'est pas très précise. Certains comptent comme une opération plusieurs opérations élémentaires (+,-,*,/), alors que d'autres comptent une opération pour chaque opération élémentaire effectuée. Nous considérerons ici une opération comme une opération arithmétique.

4.1.1.1. Coûts d'une opération élémentaire

Le coût d'une opération élémentaire est reporté dans le tableau suivant, pour leVPU (où l'on prend en compte l'initialisation de l'unité vectorielle et le premier résultat) et le Transputeur (T414).

	Vector Form	Générique	T414
multiplication	105	205	145
addition	105	201	105

Figure 35 : temps d'exécution d'une opération élémentaire (µs)

On constate que si l'on doit effectuer une seule addition, il est préférable d'utiliser le Transputeur plutôt que le VPU. Pour ce qui est de la multiplication, le T414 permet

d'obtenir un temps d'exécution raisonnable, en effet pour obtenir de meilleures performances sur le VPU il faut utiliser les "vector forms" (i.e. deux "parameter block" et neuf lignes de code!).

4.1.1.2. Performances du saxpy

Comme nous le détaillerons plus loin, le saxpy est une opération vecteur-vecteur fondamentale, elle correspond à la mise à jour d'un vecteur composante par composante.

Le premier phénomène qui intervient lorsqu'on observe les performances de cette opération sur le VPU est la taille des vecteurs : les performances ne sont pas linéaires pour des vecteurs de taille quelconque. En effet, lorsque la taille des vecteurs n'est pas un multiple de celle des registres vectoriels (128 réels double précision), il est nécessaire de décaler les données de façon à ce que les vecteurs soient alignés correctement dans les registres vectoriels. Ainsi, un saxpy sur un vecteur comportant 129 élément est bien évidemment plus coûteux que sur un vecteur de 128 éléments, mais est aussi plus coûteux que sur un vecteur de 256 éléments !

Néanmoins, pour des vecteurs dont la taille est un multiple de 128, les performances sont linéaires. Nous nous intéresserons pour la modélisation uniquement à de tels vecteurs (il est de plus facile, et même plus intéressant, de s'y ramener).

Il existe deux cas différents, suivant l'emplacement des variables en mémoire. Examinons tout d'abord le cas où les deux vecteurs sont placés dans des bancs mémoire différents (cas favorable).

On obtient les résultats expérimentaux suivants, consignés dans la figure 36.

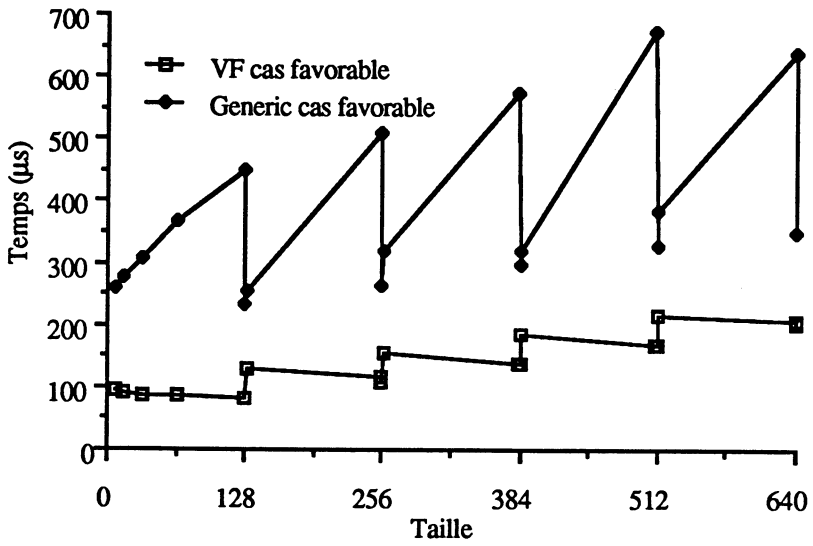


Figure 36 : saxpy sur des vecteurs placés dans deux bancs mémoire différents

On peut aisément se ramener au cas des multiples de 128 (beaucoup plus favorable car il évite des manipulations des résultats en mémoire) en complétant les vecteurs avec des 0, on obtient alors la courbe suivante. Les autres courbes ne font apparaître que les multiples

de 128, mais on observe ce phénomène dans tous les cas.

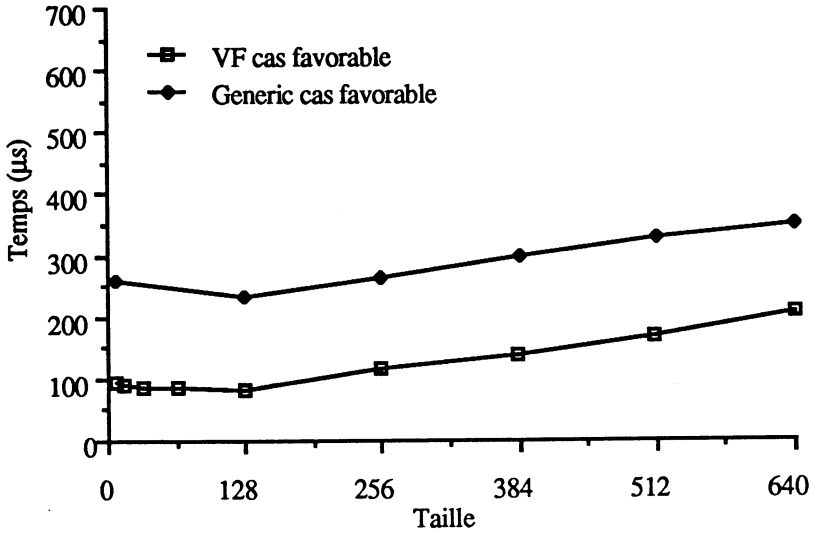


Figure 37 : saxpy sur des vecteurs placés dans deux bancs mémoire différents

On en déduit la modélisation suivante :

- avec les fonctions "generic"

$t1_{saxpy} = an + b$ avec a = 0.23 µs et b = 205.6 µs

- avec les "vector forms"

$t2_{saxpy} = a'n + b'$ avec a'= 0.23 µs et b'= 51.6 µs

Examinons maintenant le cas où les deux vecteurs sont dans le même banc mémoire (on peut être obligé de travailler dans ce contexte défavorable pour des raisons de place mémoire). On obtient alors les résultats expérimentaux suivants :

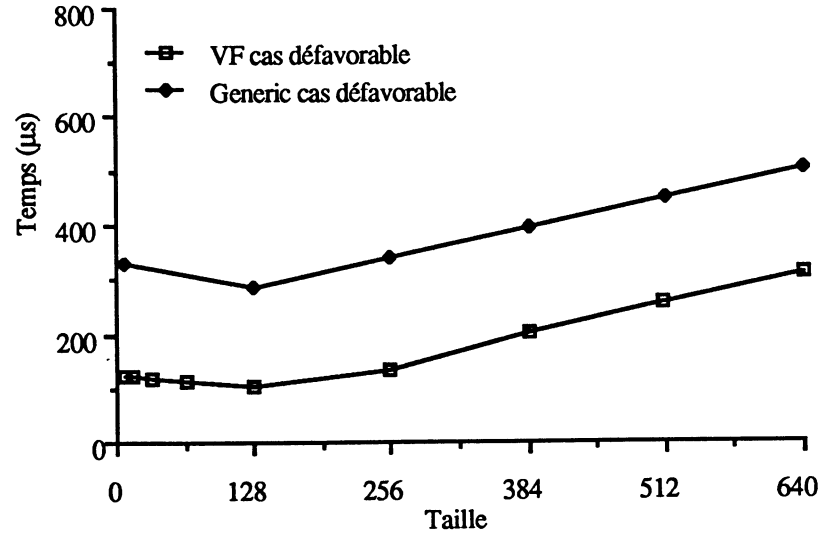


Figure 38 : saxpy sur des vecteurs placés dans le même banc mémoire

On en déduit la modélisation suivante :

- avec les fonctions "generic"

$t3_{saxpy} = cn + d$ avec a= 0.41 µs et b= 231.8 µs

- avec les "vector forms"

$t4_{saxpy} = c'n + d'$ avec a'= 0.40 µs et b'= 51.3 µs

Les performances de la machine sur cette opération varient de 0,17 à 8,34 Mflops avec les "vector forms" et de 0,06 à 7,39 Mflops avec les fonctions génériques pour le cas favorable. Les "vector forms" permettent d'obtenir un gain de performance de l'ordre de 11%.

Dans le cas défavorable, les vector forms" permettent un gain de 10% par rapport aux fonctions "generic".

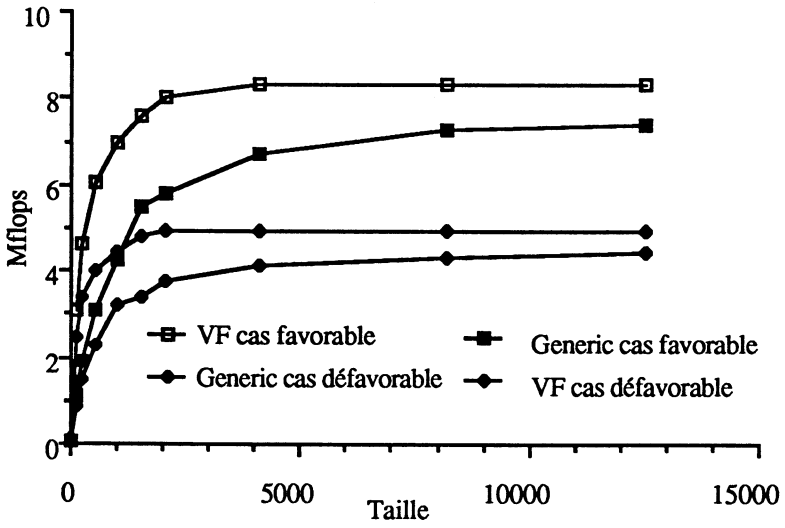


Figure 39 : performances en Mflops du saxpy en fonction de la taille

Ces performances sont loin d'égaler celles d'un Cray : sur un tel problème, les performances d'un Cray X-MP sont d'environ 178 Mflops et celles du Cray Y-MP de l'ordre de 219 Mflops. Remarquons de plus que ces deux machines obtiennent de telles performances pour des vecteurs de taille 400, alors que le T40 atteint sa puissance maximale pour des vecteurs dont la taille est grande, supérieure à 5000 éléments.

4.1.1.3. Performances du produit-scalaire

Le temps de calcul d'un produit scalaire n'est pas proportionnel à la taille des vecteurs. On constate en effet qu'avec les fonctions "generic", le temps d'exécution du produit scalaire est constant pour des vecteurs de taille inférieure ou égale à 128. Alors que pour les "vector forms" ce temps est croissant : aprés chaque multiple de 128, on observe un "petit saut" qui est dû à l'exécution du "vector form" `VF_RBODY` de réduction d'argument.

Si on ne s'intéresse qu'aux vecteurs de tailles multiples de 128 (i.e. la taille des registres vectoriels), on trouve que le temps d'exécution est linéaire. C'est à dire que le temps d'exécution d'un produit scalaire de taille n*128, avec n=1,2,3... est égal à n fois celui d'un produit scalaire de taille 128.

La figure suivante représente les temps d'exécution du produit scalaire en fonction de la taille des vecteurs.

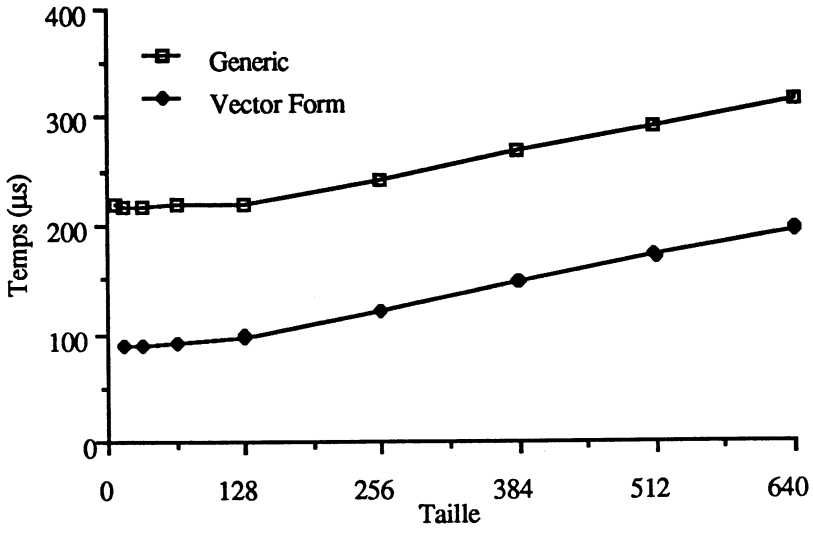


Figure 40 : temps d'exécution du produit scalaire

Cela permet de modéliser un produit scalaire sur le VPU (pour des vecteurs dont la taille est un multiple de 128) :

- avec des fonctions "generic"

  $t1_{dot} = an + b$ avec a = 0.19 µs et b = 192 µs

-avec des "vector forms"

  $t2_{dot} = a'n + b'$ avec a'= 0.19 µs et b'= 73 µs

Observons maintenant les performances correspondantes en Mflops :

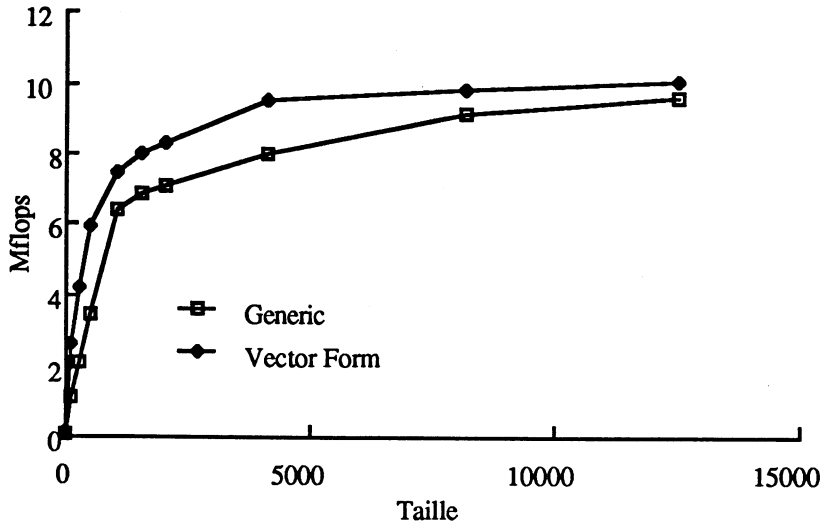


Figure 41 : performances du produit scalaire (Mflops)

Les performances de la machine sur cette opération varient de 0.17 à 10 Mflops avec les "vector"forms" et de 0.07 à 9.56 Mflops avec les fonctions "generic". Les "vector forms" permettent d'obtenir un gain de performance de l'ordre de 5%.
Cette opération est celle qui permet d'avoir les performances vectoriels les plus élevées sur le T40. En effet, on obtient 10.05 MFlops alors que la puissance de crête théorique du VPU est de 12 MFlops.
Comme pour le saxpy, si l'on compare avec les performances obtenues sur des supercalculateurs Cray, on obtient 160 Mflops pour un Cray X-MP et 220 Mflops pour un Cray Y-MP. Ces performances sont atteintes pour des vecteurs dont la taille est proche de 1000 éléments.

Ces résultats obtenus sur le VPU peuvent être comparés aux résultats obtenus sur l'unité de calcul arithmétique flottante du Transputer.
Comme le montre la figure suivante, le temps d'exécution de ces opérations fondamentales est linéaire, ce qui est normal puisque le Transputer est un microprocesseur séquentiel.

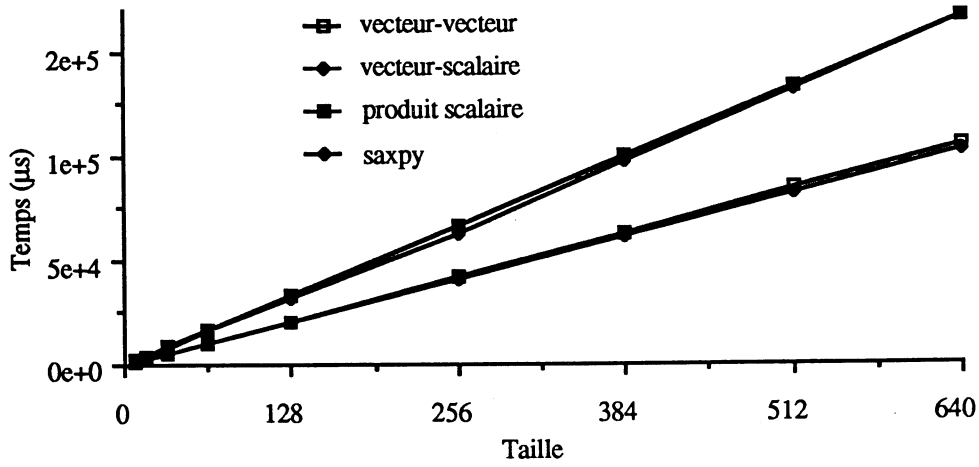


Figure 42 : temps d'exécution des opérations algébriques fondamentales sur le Transputer

Les temps d'exécution obtenus sur le Tranputer dépendent de deux facteurs : le temps de calcul d'une opération élémentaire et le temps des entrées-sorties entre le Transputer et sa mémoire externe. En effet, le code des opérations algébriques fondamentales testées ainsi que les données correspondantes sont stockées dans la mémoire externe du Transputer.

4.1.2. Les communications

Dans les ordinateurs à mémoire distribuée, les données transitent par échanges de messages. Ainsi, une étude fine des communications est d'une grande importance pour la modélisation d'algorithmes. Le Transputer possède seulement 4 liens physiques, permettant 4 communications parallèles, le comportement des communications doit être étudié précisément pour utiliser efficacement toutes les possibilités d'un hypercube de degré 5 (à 32 processeurs).
La plupart des algorithmes numériques utilisent des produits scalaires (usuellement référencé par *dot*), des *saxpy* (mise à jour de la forme : scalaire multipliée par un vecteur ajouté à un vecteur) ou des opérations sur des blocs. Ces procédures fondamentales impliquent des échanges de données d'un processeur à un autre (communication élémentaire pourvu que les deux nœuds soient voisins), de diffusion (un processeur communique un vecteur à tous les autres) ou de répartition (chaque processeur reçoit un vecteur, différent de celui des autres, tous en provenance d'un seul processeur émetteur). Ces différents types de communications sont étudiés en détail.

4.1.2.1. Communications élémentaires

La communication élémentaire sur des ordinateurs à mémoire distribuée est celle entre deux processeurs voisins. Sur la plupart des machines, le temps t(k) nécessaire pour

envoyer k octets à un processeur voisin peut être décomposé en une initialisation α (nécessaire à la configuration du matériel, ici les multiplexeurs) et un taux de transmission β :

$$t(k) = \alpha + \beta k$$

pour un ordinateur donné (par exemple le FPS T40), les valeurs de α et β dépendent de divers paramètres, comme le montre les figures suivantes.

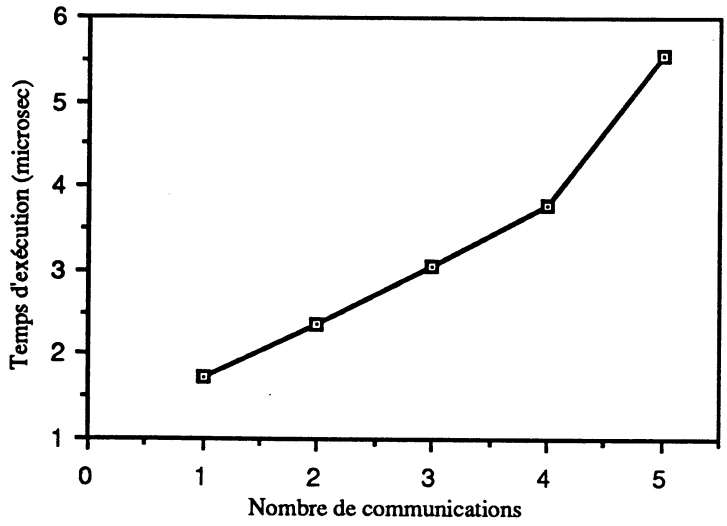


Figure 43 : variation du nombre maximum de communications simultanées
(sur un 5-cube pour 500 octets)

Ici, on envoie un message de 500 octets d'un processeur à ses voisins physiques. La loi est linéaire avec une pente assez faible jusqu'à 4 communications (nombre des canaux physiques), et les perfromances se dégradent pour un nombre supérieur de communications.
Examinons maintenant l'influence de la configuration du réseau de processeurs sur des communications entre voisins directs :

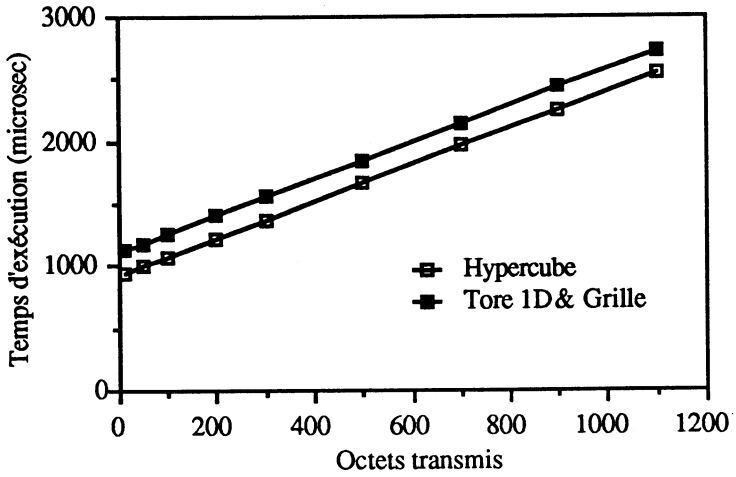


Figure 44 : influences de différentes configurations

Il apparaît ainsi que la configuration du réseau de processeurs en hypercube est la plus intéressante pour transmettre un message entre deux nœuds voisins. D'abord parce que c'est dans cette configuration que les communications élémentaires sont les moins coûteuses et ensuite parce que toutes les autres topologies peuvent être construites à partir d'un hypercube.
Pour le T40, les courbes expérimentales conduisent aux valeurs suivantes pour une communication entre deux processeurs voisins, avec i communications en parallèle :

$$t_i(k) = 768i + 160 + 1.6j\,k \quad (j=1 \text{ pour } 1 \leq i \leq 4 \text{ et } j=2 \text{ pour } i=5)$$

où k est le nombre d'octets transmis (le temps est exprimé en µs).

4.1.2.2. Communications plus élaborées

La valeur très importante du temps d'initialisation des multiplexeurs lors de communications élémentaires implique des performances faibles pour les communications complexes construites par l'utilisateurs à base des ces communications élémentaires. Il est donc nécessaire d'utiliser aussi souvent que possible les routines de communications fournies par FPS (qui s'affranchissent partiellement de ce temps d'initialisation prohibitif puisqu'elles sont écrites en langage de bas niveau).

Routage

Le premier type de communication élaboré que l'on peut obtenir à partir de communications élémentaires est le routage (c'est-à-dire faire communiquer entre eux deux processeurs non voisins). Le routage n'existe pas sur toutes les machines distribuées, il est cependant d'un grand intérêt pour le programmeur, mais les

communications sont très coûteuses (à titre d'exemple, la Connection Machine ou l'IPSc possèdent une couche de routage dont les performances sont de l'ordre d'une dizaine de millisecondes pour communiquer un octet !). FPS fournit une solution très partielle à ce problème : il n'est possible de faire communiquer deux processeurs non voisins que s'ils appartiennent à la même dimension d'un tore (anneau ou grille). La figure ci-dessous fournit les temps de communications pour la plus grande dimension d'une grille 4*8. Le processeur 0 émet vers les autres sur une grille comme l'indique la figure 45.

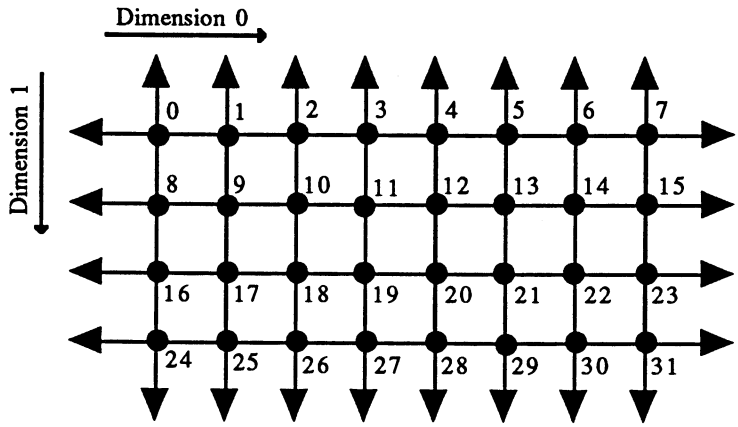


Figure 45 : numérotation sur la grille

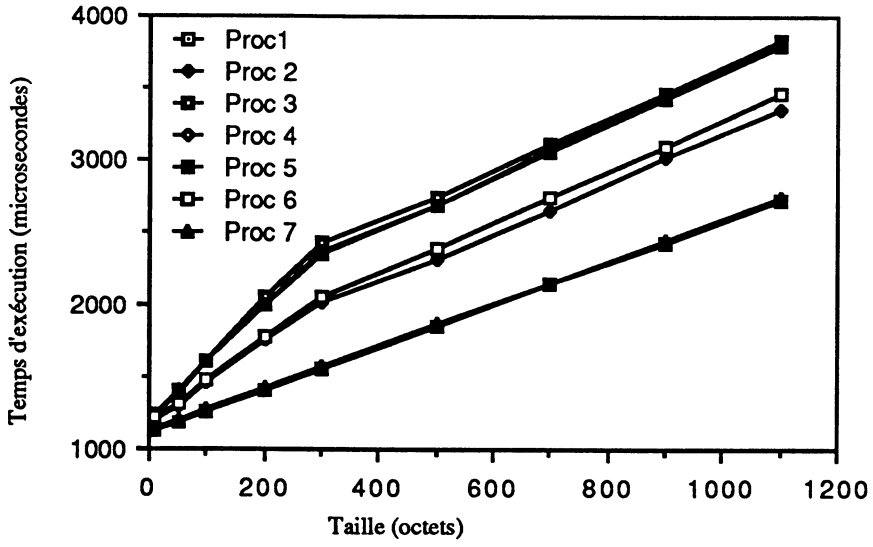


Figure 46 : routage sur la grille

Sur la figure précédente, il apparaît que les processeurs sont groupés suivant leur distance du processeur émetteur. Chaque groupe de processeurs peut alors être modélisé par une fonction affine par morceaux ($\alpha+\beta k$, suivant les valeurs de k).

Finalement, on peut modéliser le routage sur le T40 configuré en tore par :

i=1, 7	t(k) = 1124 + 1.46k	
i=2, 6	t(k) = 1181.2 + 2.88k	pour k ≤ 256
	t(k) = 1478.2 + 1.72k	pour k ≥ 256
i=3, 4, 5	t(k) = 1209.7 + 3.95k	pour k ≤ 256
	t(k) = 1757.5 + 1.81k	pour k ≥ 256

où t(k) est le temps nécessaire pour envoyer k octets.

Diffusion

Le but d'une diffusion (*broadcast*) est d'envoyer une même donnée sur tous les processeurs. L'algorithme de diffusion rotative utilisé ici est dû à Johnsson & Ho [JoHo]. Nous utilisons la notion d'arbre binaire de recouvrement. Il s'agit de trouver le meilleure répartition (la plus équilibrée possible ...) des envois dans l'hypercube. La figure ci-dessous représente un tel arbre dans le cas d'un hypercube de degré 4.

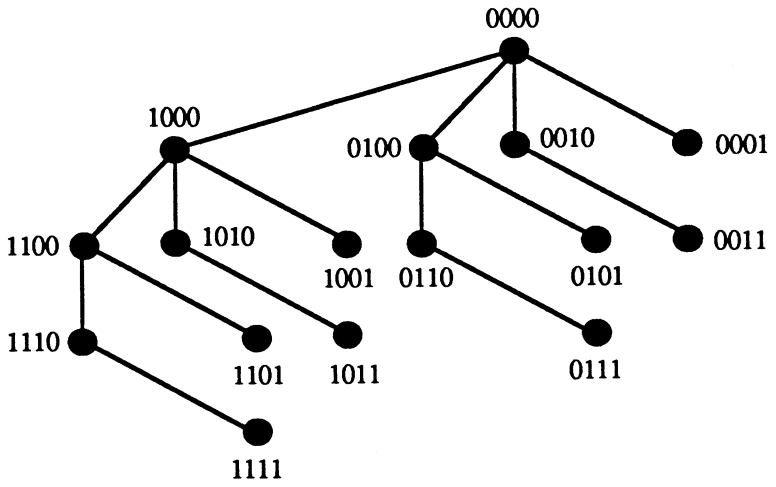


Figure 47 : arbre binaire de recouvrement sur un hypercube de degré 4

Une première stratégie (décrite en 4.3.1.2) consiste à découper les données en paquets de façon à pipeliner les envois.

De plus, plusieurs arbres de recouvrements peuvent être utilisés simultanément pour exploiter la possibilité des liens en parallèle. La figure suivante schématise une diffusion dans le cas d'un hypercube de dimension 3, étape par étape.

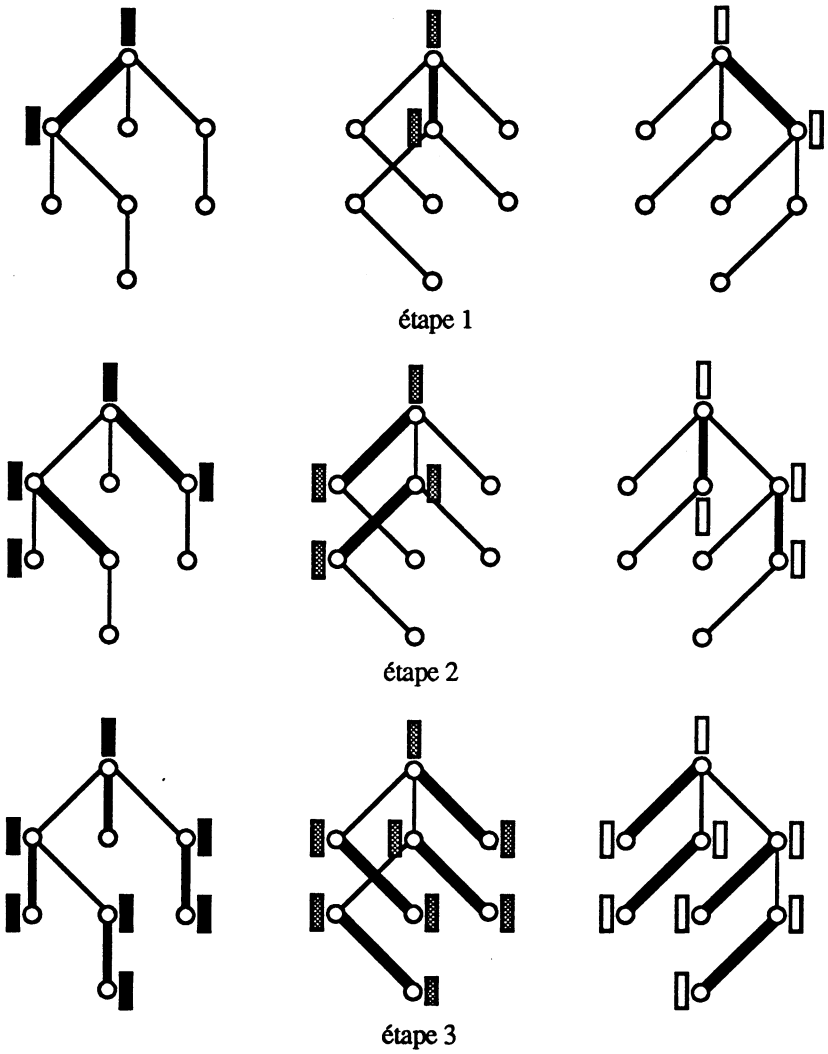


Figure 48: diffusion rotative sur un 3-cube

On peut ainsi modéliser le temps d'exécution d'une telle diffusion :

$$t(k) = p\beta + k\alpha$$

Les données sont divisées en p paquets de taille égale et sont diffusées en un temps de $\beta + k\alpha/p$ sur chacun des arbres de recouvrement binaires, k étant le nombre d'octets diffusés. On trouve, sur le FPS T40, les valeurs numériques suivantes (déduites des valeurs expérimentales de la figure 49), exprimées en μs :

$$t(k) = 3477.5 + 1.5\,k \qquad \text{pour } k \leq 32768$$

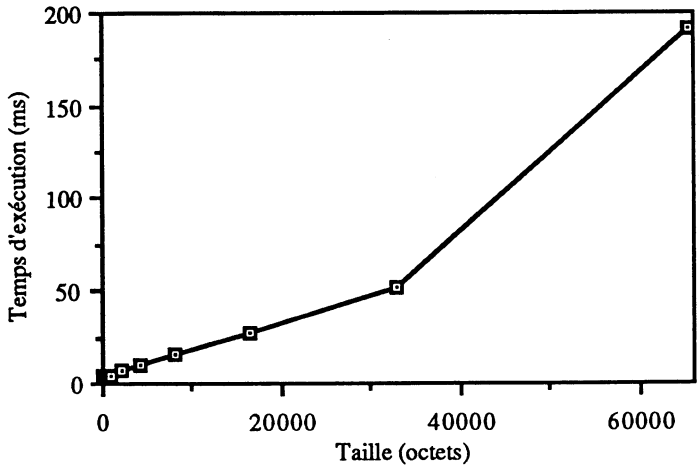
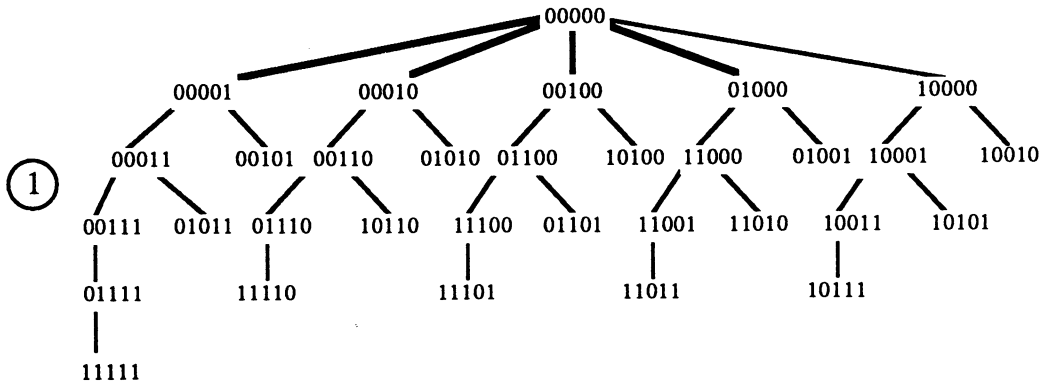
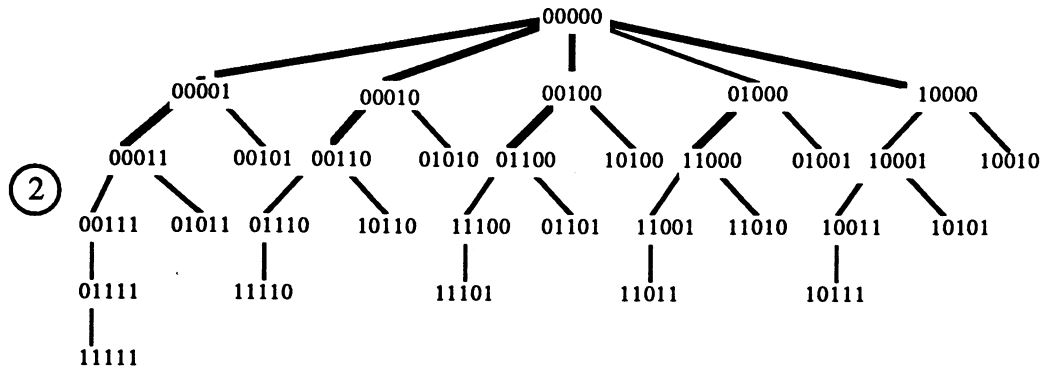


Figure 49 : performance de la diffusion rotative avec 4 arbres de recouvrement

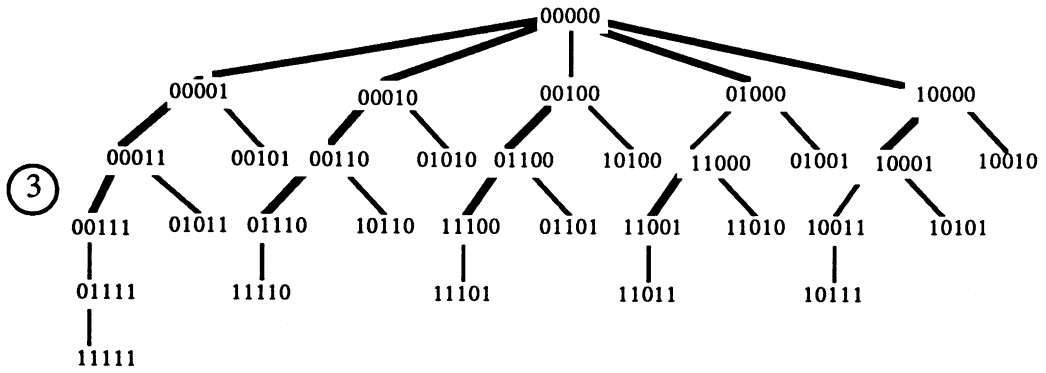
Répartition

Une répartition (dispatch) permet d'envoyer aux autres processeurs des morceaux différents d'un même vecteur stocké dans un processeur émetteur. Ici aussi, la structure logique adéquate est un arbre binaire de recouvrement sur un hypercube de dimension 5. Notons que le Transputer possède 4 liens pour les communications ; nous allons donc construire un algorithme permettant 4 communications simultanées à chaque pas. Le principe est décrit sur la figure 50.

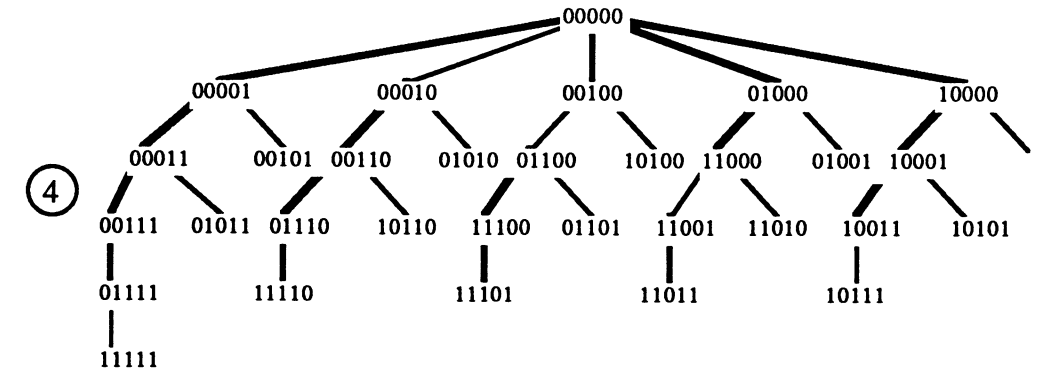



00000
00001 00010 00100 01000 10000
00011 00101 00110 01010 01100 10100 11000 01001 10001 10010
00111 01011 01110 10110 11100 01101 11001 11010 10011 10101
01111 11110 11101 11011 10111
11111

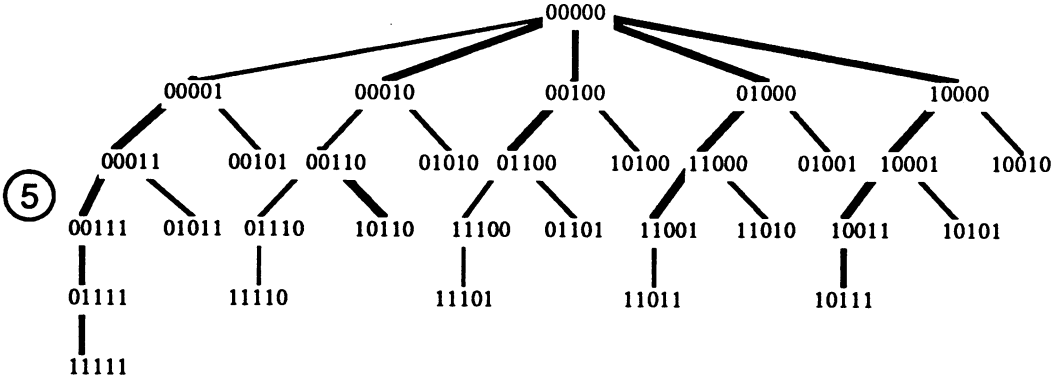
2


00000
00001 00010 00100 01000 10000
00011 00101 00110 01010 01100 10100 11000 01001 10001 10010
00111 01011 01110 10110 11100 01101 11001 11010 10011 10101
01111 11110 11101 11011 10111
11111

3


00000
00001 00010 00100 01000 10000
00011 00101 00110 01010 01100 10100 11000 01001 10001
00111 01011 01110 10110 11100 01101 11001 11010 10011 10101
01111 11110 11101 11011 10111
11111

4


00000
00001 00010 00100 01000 10000
00011 00101 00110 01010 01100 10100 11000 01001 10001 10010
00111 01011 01110 10110 11100 01101 11001 11010 10011 10101
01111 11110 11101 11011 10111
11111

5

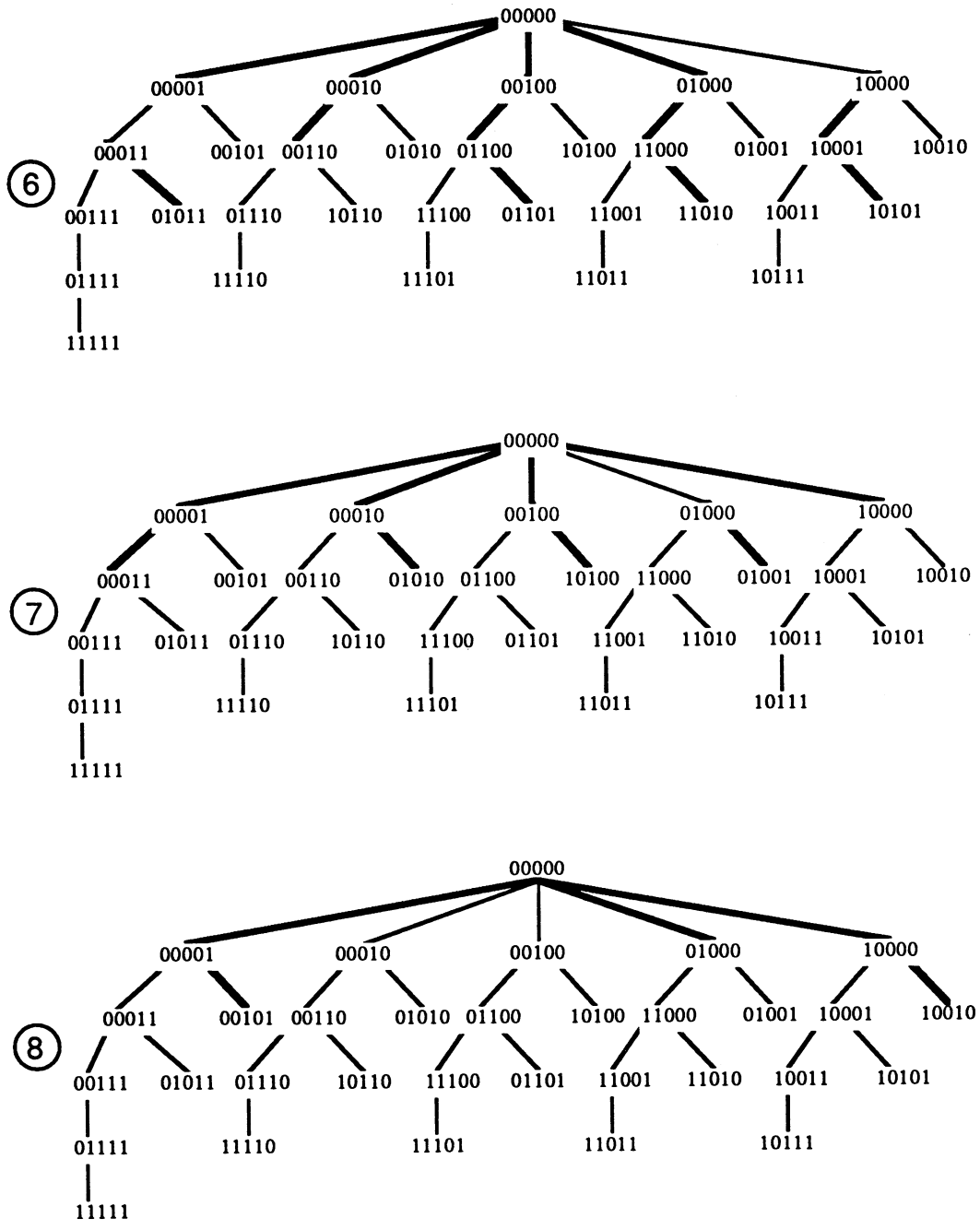


Figure 50 : les 8 étapes du dispatch avec 4 voisins

On peut modéliser ainsi l'algorithme : on effectue 7 étapes avec chacune 4 communications simultanément (pas tout à fait parallèle en pratique comme nous l'avons vu), et la dernière étape comporte seulement 3 communications en parallèle, soit 31 au total. Le temps d'exécution peut donc être exprimé ainsi :

$$t(k) = 7(\alpha_4 + \beta k) + \alpha_3 + \beta k$$

où α_3 et α_4 sont les temps d'initialisation pour respectivement 3 et 4 communications parallèles, β est le taux de transmission, et k le nombre d'octets transmis. Ce qui donne pour le T40 le temps d'exécution suivant (en μs) :

$$t(k) = 25088 + 12.8k$$

Néanmoins, chaque nœud d'un hypercube de dimension 5 ayant 5 voisins, il est intéressant d'observer le comportement d'un algorithme construit en tenant compte de cette particularité. Le même arbre binaire de recouvrement peut être utilisé, seules les communications à chaque étape changent : on s'autorise à faire 5 communications en parallèle au lieu de 4. Ces changements conduisent à un algorithme en 7 étapes (au lieu de 8 pour l'algorithme avec 4 voisins, qui se répartissent en 6*5+1).
La modélisation est analogue au cas précédent, on obtient alors les résultats suivants pour transmettre k octets :

$$t(k) = 6(\alpha_5 + \beta_5 k) + \alpha_1 + \beta k$$

où α_5 désigne le temps d'initialisation et β_5 le taux de transmission pour 5 communications en parallèle, correspondant aux 6 dernières étapes du dispatch avec 5 voisins, α_1 est le temps d'initialisation et β le taux de transmission usuel.
Pour le FPS T40, toute expérimentation et calcul effectués, on obtient alors les valeurs numériques suivantes (en μs) :

$$t(k) = 24928 + 20.8k$$

L'algorithme avec 4 voisins est donc théoriquement plus intéressant à partir des 24 octets transmis (soit 3 réels double précision). La figure suivante illustre les résultats expérimentaux obtenus.

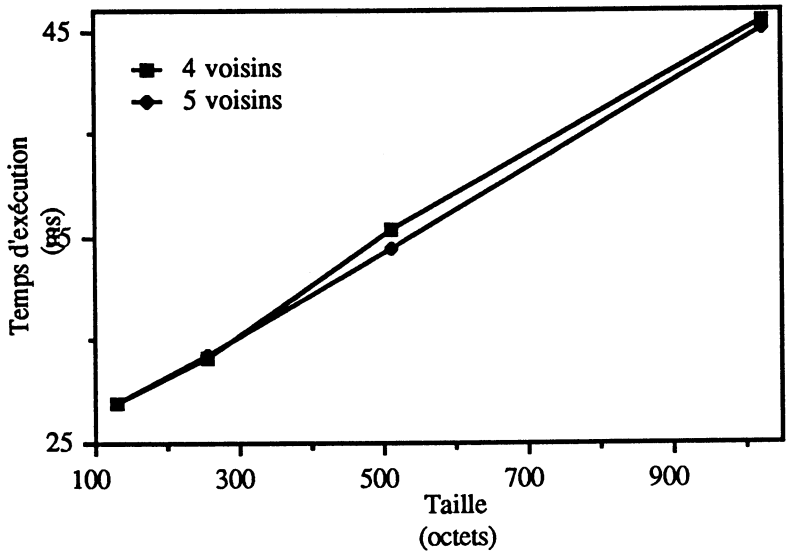


Figure 51 : répartition avec 4 et 5 voisins

Il apparaît qu'expérimentalement l'algorithme avec 5 voisins est le plus performant, ce qui ne serait pas en adéquation avec le modèle théorique adopté. Cette contradiction peut s'expliquer en analysant précisément les communications des deux algorithmes et en faisant un rapprochement avec l'architecture du Transputer.

Commençons par analyser l'algorithme avec 4 voisins : notre modèle prévoit 4 communications simultanées, ce qui est parfaitement réalisable sur un Transputer possèdant 4 canaux de communication. Cependant sur un hypercube de dimension 5, il se peut que deux communications se fassent sur la même entrée du multiplexeur, i.e. sur le même lien du Transputer comme le montre la figure suivante :

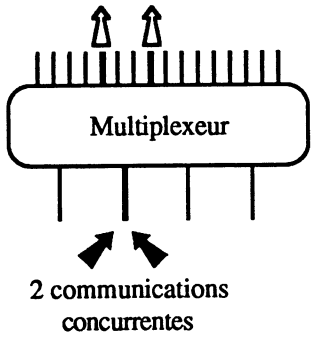


Figure 52 : communications concurrentes dans l'algorithme avec 4 voisins

Dans le cas de l'algorithme avec 5 voisins, les communications font intervenir 5 communications "simultanées" ; le Transputer ayant 4 canaux physiques, les quatre premières communications vont pouvoir se dérouler en parallèle et la dernière communication se fera ensuite séquentiellement. Ce phénomène, transparent à l'utilisateur (car géré par les fonctions de communication fournies par FPS), est bien pris en compte par notre modèle. De plus, un phénomène analogue à celui se produisant pour

l'algorithme avec 4 voisins, ici 3 communications en entrée d'un multiplexeur, ne peut se produire sur un hypercube de degré 5.
Remarquons pour finir, que le cas (parfaitement possible sur un hypercube de degré 5 avec 5 communications simultanées) où deux communications entrent en conflit sur une même entrée d'un multiplexeur se reproduit sur plusieurs liens d'un Transputer est aussi pris en compte par notre modèle : dans un premier temps, une des deux communications s'exécute (sur chacun des liens où existe un conflit) et la seconde communication est ensuite traitée séquentiellement. La figure suivante schématise cette éventualité.

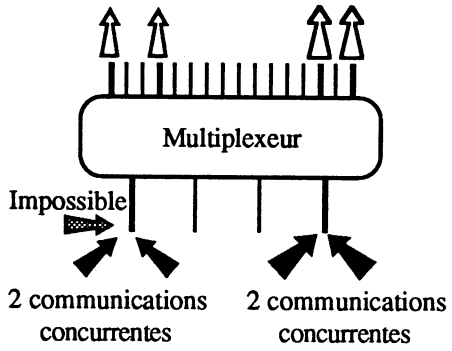


Figure 53 : communications dans l'algorithme avec 5 voisins

4.2. Exemple : calcul du produit matrice-vecteur

Le but de cet exemple est d'étudier pratiquement le processus de parallélisation du produit matrice-vecteur. L'approche choisie a été de chercher systématiquement les meilleures versions possibles parmi les différentes parallélisations possibles, avec une répartition équilibrée des données en considérant des phases successives (sans recouvrement) entre les communications et les calculs (cf [BlMAl]). Ainsi, il a fallu implanter une couche de routage pour réaliser les différentes stratégies.
Ce qui nous intéresse ici est de décrire un modèle théorique pour le temps d'exécution de chacune des versions.

4.2.1.Présentation générale

Le problème qui va nous servir d'illustration ici est le calcul du produit v=Ax, où A est une matrice dense de taille nxn, x et v deux vecteurs de dimension n ; p désigne par la suite le nombre de processeurs.

La matrice est d'abord distribuée sur tous les processeurs et reste en place jusqu'à la fin des calculs. Lors de l'utilisation pratique d'une telle procédure, les vecteurs x et v doivent

être redistribués globalement (par exemple à chaque itération si l'on considère une méthode de type gradient conjugué ou relaxation). Nous considérons les algorithmes centralisés et synchrones permettant des échanges globaux de données. Cette hypothèse est réaliste dans le sens où elle minimise les communications (il est plus efficace de collecter sur un seul processeur tous les composants d'un vecteur, même grand, et d'y calculer le produit scalaire sur l'unité vectorielle et de renvoyer le résultat aux autres processeurs, plutôt que de faire localement des petits produits scalaires et d'échanger leurs résultats partiels).
La multiplication matrice-vecteur est l'algorithme numérique le plus fondamental, il peut être écrit à base de produits scalaires (version orientée lignes), de saxpy (version orientée colonnes) ou par blocs (version intermédiaire entre les deux précédentes).

4.2.2.Version produit-scalaire

Présentons tout d'abord la version "naturelle" basée sur des produits scalaires successifs des lignes de A par le vecteur x (donnée ci-dessous en pseudo-code) :

```
{ version produit-scalaire }
pour i=1 jusqu'à n
   v[i]=0
   pour j=1 jusqu'à n
      v[i]=v[i]+A[i,j]*x[j]
```

Cette méthode est schématisée ci-dessous :

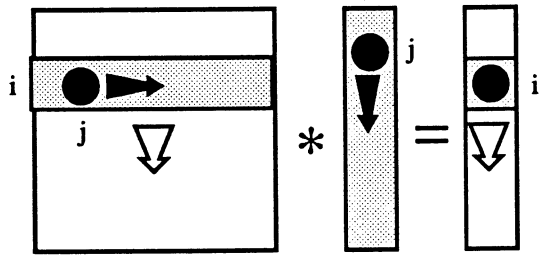


Figure 54 : produit matrice-vecteur forme produit scalaire

Les calculs peuvent alors être analysés en trois étapes :
1) diffusion du vecteur x à tous les processeurs (la matrice est en place, stockée par ligne)
2) calculs vectoriels locaux (n/p produits scalaires de vecteurs de taille n)
3) communications de n/p composantes depuis tous les processeurs vers un seul

4.2.3. Version saxpy

La multiplication matrice-vecteur peut aussi s'effectuer par colonne (comme décrit ci-aprés en pseudo-code) :

```
{ version saxpy }
pour i=1 jusqu'à n
   v[i]=0
pour j=1 jusqu'à n
   pour i=1 jusqu'à n
      v[i]=v[i]+A[i,j]*x[j]
```

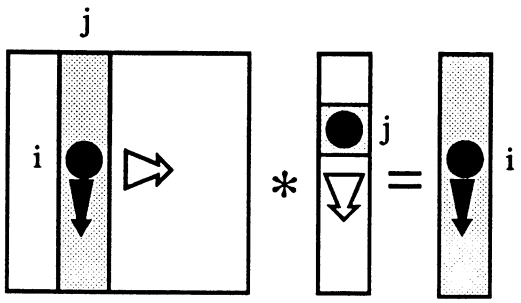


Figure 55 : produit matrice-vecteur forme saxpy

Comme dans la première version, les calculs peuvent être décomposés en trois étapes :

1) distribution du vecteur x sur les processeurs en p morceaux indépendants
2) calculs locaux (n/p saxpy sur des vecteurs de dimension n)
3) communication depuis tous les processeurs vers un seul et addition des n/p composants partiels de taille n du résultat v

4.2.4. Version par blocs

De nombreuses applications pratiques conduisent à des matrices avec des partionnements "naturels" en blocs. Les méthodes par blocs peuvent être vues comme une version mixte des deux versions précédentes (on a n=rq réparti en q^2 blocs de taille rxr). Pour simplifier ici, on peut prendre q=p ou au moins un sous-multiple de p. Dans l'algorithme suivant, les lettres capitales désignent des blocs) :

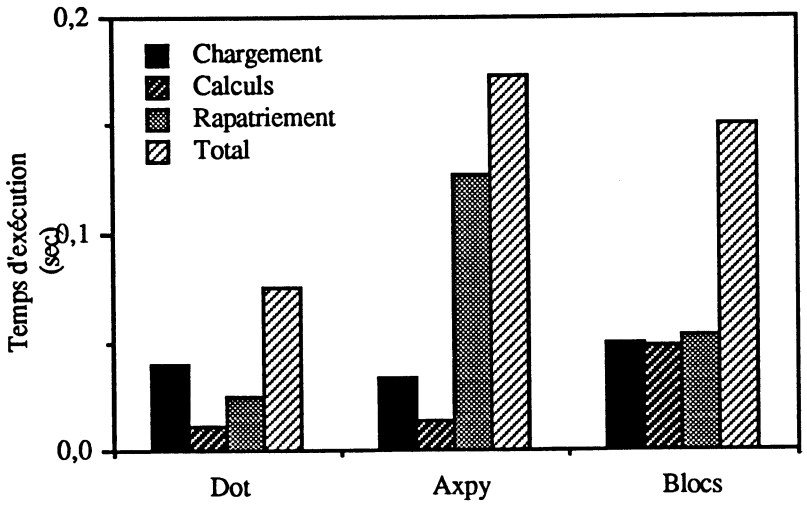
```
{ version bloc }
pour i=1 jusqu'à q
   Vr[i]=0
   pour j:=1 jusqu'à q
```

```
Vr[i]=Vr[i]+Ar[i,j]*Xr[j]
```

Comme dans la première version, les calculs peuvent être décomposés en trois étapes :
1) diffusion de morceaux du vecteur x à tous les processeurs d'une même dimension du cube
2) calculs locaux (produits matrice-vecteur)
3) communications des composantes partielles depuis tous les processeurs vers un seul

4.2.5. Implantation des diverses solutions

Rappelons que nous n'avons pas accès aux couches basses du système pour les communications, seul un routage sur la grille est possible. Nous avons expérimenté les trois solutions sur une grille en utilisant le routage pour écrire la diffusion et la répartition, puis nous avons simulé le comportement théorique en utilisant les primitives de routage sur l'hypercube que nous avons décrites, comme si l'on avait accès à la couche basse de programmation des communications.
La figure ci-dessous donne les résultats expérimentaux des différentes phases du calcul du produit matrice-vecteur de taille 1024, réparti sur 32 processeurs.



Figue 56 : comparaison expérimentale des différentes versions du produit matrice-vecteur

Quelques remarques s'imposent : tout d'abord, la taille locale des vecteurs n'est pas suffisante pour atteindre la pleine efficacité du VPU (seulement 6 Mflops localement), et d'autant pire pour la version par blocs. De plus, la phase de rapatriement des résultats dans la version saxpy est très longue à cause des sommes partielles à effectuer. Ces performances correspondent à 28 Mflops, c'est relativement faible, mais cela s'explique par un coût prédominant des communications (6 fois plus cher que le calcul pour la version dot).

La figure suivante reporte les simulations de ces différentes phases à partir des modèles théoriques que nous avons définis. Les résultats sont évidemment meilleurs.

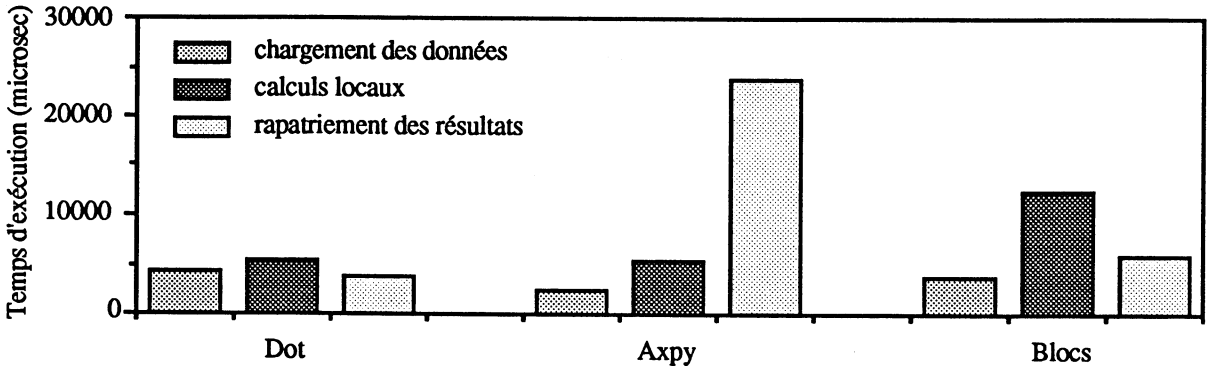


Figure 57 : comparaison entre différentes implantations du produit matrice-vecteur

Extension

Notons que d'autres méthodes peuvent être étudiées de la même manière (élimination de Gauss, Transformée de Fourier etc..) et conduire à de nouveaux algorithmes de communication.

4.3. Algorithmes de diffusion asynchrone

4.3.1. Etat de l'art

L'implantation de méthodes numériques sur un ordinateur parallèle à mémoire distribuée nécessite des échanges de données entre deux processeurs telles que la diffusion (ou broadcast : la même donnée est envoyée d'un processeur à tous les autres). Ces types de communications globales ont été récemment étudiés intensivement, aussi bien d'un point de vue théorique ([BlTr], [BlTV], [Frai], [JoHo], [StWa]) que des implantations pratiques développées pour résoudre des problèmes numériques ([SaS1], [RoTV]). Nous rappelons ci-après les différentes stratégies pour la diffusion, et généralisons le principe de la diffusion pipeline à multiples paquets à des stratégies asynchrones (les processeurs sont autorisés à calculer pendant la phase de communication sans attendre un point de synchronisation).

4.3.1.1. Définitions fondamentales et outils

Sur la plupart des ordinateurs parallèles à mémoire distribuée, les implantations efficaces sont celles qui se réfèrent à un principe de localité, utilisant un maximum de calculs locaux et tendent à minimiser les communications, qui sont de véritables goulots d'étranglement,

et prédominent souvent sur les calculs.
Toutes les formes de communications réfèrent aux communications de base entre deux processeurs voisins. Divers modèles ont été proposés, mais le plus souvent utilisé est celui qui exprime le temps c(n) nécessaire pour envoyer n données d'un processeur à un de ses voisins directs comme une initialisation β plus un taux de transmission τ :

$$c(n)=\beta+\tau n.$$

Nous considérons ici le modèle d'un ordinateur parallèle à mémoire distribuée (hypercube) avec des canaux de communication bidirectionnels, où les communications et les calculs se font sans recouvrement. Ce modèle suppose que les communications peuvent se faire en parallèle pour un processeur donné. Les canaux de communication sont bidirectionnels, mais les communications ne sont alors pas simultanées, ce qui est le cas du Transputer [DeTo].

Les arbres de recouvrement binaires sont un des outils fondamentaux pour l'implantation des schémas de communication : il s'agit d'arbres qui contiennent tous les nœuds du réseau et dont les arêtes forment un sous-ensemble de celles du réseau. Ils peuvent être définis récursivement comme suit : un arbre binaire de degré 0 possède un nœud, et un arbre binaire de degré n est construit à partir de deux arbres binaires de degré n-1 en ajoutant une arête entre les racines des deux arbres et en choisissant l'une des deux comme la nouvelle racine, il en existe alors n.

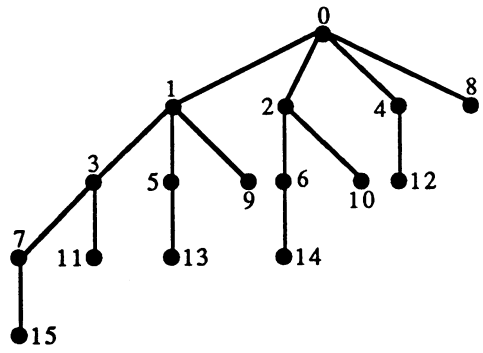


Figure 58 : exemple d'arbre de recouvrement binaire sur un 4-cube

4.3.1.2. Les différents algorithmes de diffusion synchrone

Le but d'une diffusion est d'envoyer la même donnée, stockée sur un processeur, à tous les autres. La diffusion a été récemment étudiée par [JoHo], [StWa], [SaS2] et [Try2]. Il y a quatre façons principales d'effectuer une diffusion : la plus simple consiste à envoyer les données en une seule fois, la seconde améliore cette première approche naïve en découpant le message en plusieurs paquets et pipeline les envois [SaS2], les deux dernières méthodes (qui sont les plus sophistiquées) consistent à découper les données et

à utiliser simultanément plusieurs arbres de recouvrement binaires ([JoHo], [StWa]).
Nous rappelons maintenant brièvement les principes des différents schémas de diffusion sur un réseau de p processeurs à topologie hypercube.

La première solution consiste à envoyer les données en une seule fois : en utilisant seulement les communications entre deux processeurs voisins, le temps $t_{broad}(n)$ nécessaire pour diffuser n données en un seul paquet sur un hypercube à p processeurs est $t_{broad}(n)=(\beta+\tau n).\log_2(p)$. L'inconvénient de cette méthode réside dans le fait que certains processeurs restent inactifs pendant un temps important.

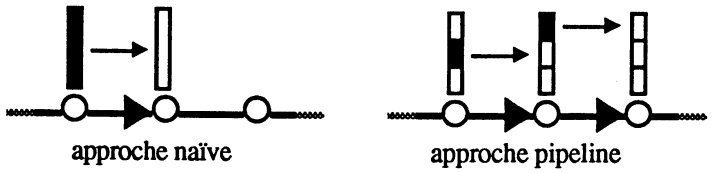


Figure 59 : stratégies de diffusion pipeline

La seconde solution consiste à découper les données en q paquets de taille k (n=qk), de façon à pipeliner les communications.
Le temps de la diffusion devient alors :

$$t_{broad}(n)=(q+\log_2(p)-1)(\beta+\frac{\tau n}{q}).$$

La taille optimale k des q paquets est :

$$k_{opt} = \sqrt{\frac{\beta}{(\log_2(p)-1)\tau n}}.$$

Malheureusement, certains canaux de communications restent cependant inutilisés.

Les deux dernières solutions (diffusions rotatives) consistent à utiliser simultanément plusieurs arbres binaires de recouvrement pour effectuer des communications en parallèle (voir figure 48 dans le cas d'un 3-cube). Pour envoyer n données, $\log_2(p)$ arbres binaires de recouvrement sont utilisés et les données sont découpées en $\log_2(p)$ paquets de taille égale.. A la première étape, les paquets sont successivement envoyés un sur chaque arbre et les envois sont alors permutés et progressent chacun dans leur arbre. Le temps de diffusion est donné par $t_{broad}(n)=\log_2(p)\beta+\tau n$. Ce schéma peut être amélioré en considérant des canaux des communications bidirectionnels simultanés (mais cela ne rentre pas dans le modèle choisi). On peut ainsi résumer les performances de ces différentes stratégies :

	Temps de communication (hypercube à p processeurs, taille n)
diffusion naïve	$\log_2(p)\beta + n\log_2(p)\tau$
diffusion pipeline	$(\sqrt{(\log_2(p)-1)\beta}+\sqrt{n\tau})^2 \quad k_{opt} = \sqrt{\frac{n\beta}{(\log_2(p)-1)\tau}}$
diffusion rotative	$\log_2(p)\beta + n\tau$

Figure 60 : temps d'exécution des différentes diffusions

4.3.2. La diffusion asynchrone

L'idée fondamentale consiste à permettre aux processeurs de commencer à travailler dès que possible, de façon à ce qu'ils n'aient pas à attendre de synchronisation (en d'autres termes, nous proposons de commencer à effectuer les calculs sur les processeurs inactifs pendant que les autres finissent de communiquer). Ainsi, notre but est de calculer la quantité de données à allouer à chaque processeur, de façon à ce qu'ils terminent leurs calculs en même temps (bien qu'ils aient commencé à travailler de façon asynchrone). Parmi toutes les stratégies, nous cherchons celles qui minimisent le temps total d'exécution $t_{tot}(n)$ (le temps $t_c(n)$ nécessaire pour communiquer les n données plus le temps $t_{ac}(n)$ nécessaire pour achever les calculs correspondants, après les communications). Bien sûr, la répartition des données n'est plus équilibrée parmi les processeurs. Considérons que les calculs sont divisés en noyaux fondamentaux d'un temps d(n), typiquement des produits scalaires dans le cas du produit matrice-vecteur. Cette hypothèse est raisonnable dans le cas d'algorithmes numériques.

Deux problèmes doivent être résolus : tout d'abord, nous devons déterminer la quantité de données à allouer à chaque processeur, et ensuite minimiser $t_{tot}(n)$. La réponse au premier problème est donné par les descriptions des stratégies à un seul ou plusieurs paquets. Ainsi, nous devons trouver le nombre K_i de noyaux de calculs fondamentaux à allouer au processeur i. Notons qu'une réponse incomplète au second problème est donné dans [SaS2] pour un problème donné (version diffusion de l'élimination de Gauss) avec une répartition équilibrée des données et en utilisant deux phases successives de communication et de calcul.

4.3.2.1. La stratégie à 1-paquet

Le principe de la stratégie à 1-paquet est d'envoyer les données en une seule fois le long d'un arbre binaire de recouvrement. La figure ci-dessous illustre ce principe (les processeurs grisés sont actifs, ceux en blanc sont inactifs, les arête en gras correspondent aux communications).

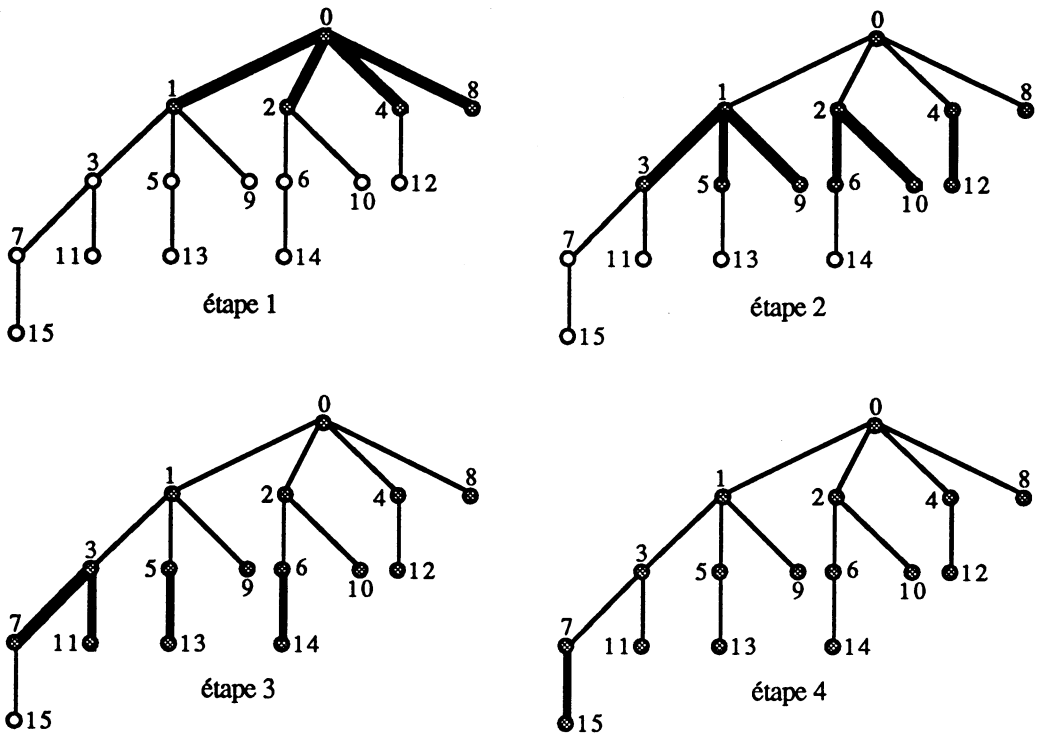


Figure 61 : la stratégie à 1-paquet sur un 4-cube

Dans la plupart des applications numériques les données sont réparties également entre les processeurs. Cela n'est pas ici le cas, et il faut trouver la quantité de données à assigner à chaque processeur.

Soit P_i l'ensemble des processeurs prêt à commencer les calculs à l'étape i (i=0 jusqu'à m), λ_i le cardinal de P_i, K_i le nombre de noyaux d'exécution attibués à P_i, d(n) le temps d'exécution de ces noyaux et c(n) le temps de communication entre deux processeurs voisins. Nous obtenons donc sur le 4-cube :

$P_0=\{0\}$ $\lambda_0=1$,
$P_1=\{1,2,4,8\}$ $\lambda_1=4$,
$P_2=\{3,5,6,9,10,12\}$ $\lambda_2=6$,
$P_3=\{7,11,13,14\}$ $\lambda_3=4$,
$P_4=\{0\}$ $\lambda_4=1$,

La figure ci-dessous détaille le temps d'occupation pour chaque processeur, notons que le temps d'occupation total a été synchronisé sur la fin des calculs locaux.

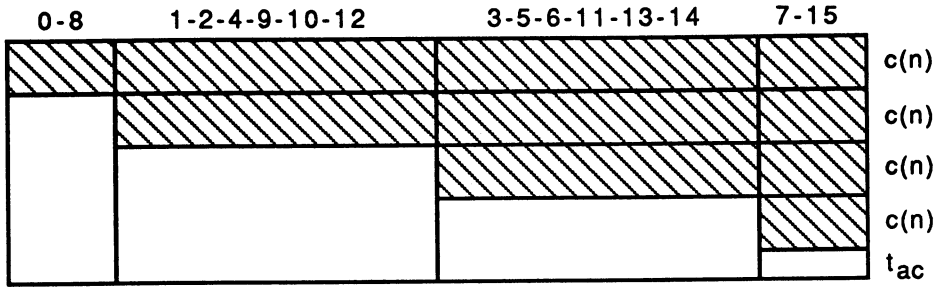


Figure 62 : temps d'occupation des processeurs

Les données doivent être distribuées aux processeurs et nous voulons calculer la quantité de données (le nombre K_i de noyaux de calculs fondamentaux) à allouer au processeur i. Une première équation est obtenue à partir de la quantité totale de travail distribuée aux processeurs (à chaque étape j) qui est égale à la taille du problème :

$\sum_{j=0}^{\log_2(p)} \lambda_j K_j d(n) = n$, où λ_j est le nombre de processeurs inactifs à l'étape j.

Ce qui donne ici $K_0 + 4K_1 + 6K_2 + 4K_3 + K_4 = n$.
Les autres équations sont trouvées en exprimant le temps d'exécution de chaque processeur i par rapport au dernier processeur inactif (rappelons que c(n) désigne le temps d'une communication fondamentale entre deux processeurs voisins pour envoyer n données) :

$$K_i d(n) = K_{\log_2(p)} d(n) + (\log_2(p)-i-1)c(n), \text{ pour i variant de 1 à } \log_2(p)-1.$$

Le système de 5 équations à 5 inconnues K_i obtenu peut être résolu facilement. Les solutions sont :

$$K_0 = \frac{n}{16} + 2\frac{c(n)}{d(n)}$$
$$K_1 = \frac{n}{16} + \frac{c(n)}{d(n)}$$
$$K_2 = \frac{n}{16}$$
$$K_3 = \frac{n}{16} - \frac{c(n)}{d(n)}$$
$$K_3 = \frac{n}{16} - 2\frac{c(n)}{d(n)}$$

Les solutions dépendent donc du rapport $\frac{c(n)}{d(n)}$ et les résultats pratiques dépendent donc vraiment de la machine : pour les calculateurs où c(n) est bien plus grand que d(n) (ce qui est le cas de la plupart des machines à topologie hypercube avec unités vectorielles par exemple) la répartition des données sur chaque processeur variera beaucoup. Mais sur les machines où c(n) et d(n) sont du même ordre (comme un réseau de Transputers par exmple), la répartition des données entre les processeurs sera sensiblement constante.

Le temps d'exécution total ($t_{tot}(n)=t_c(n)+t_{ac}(n)$) pour cette stratégie de diffusion est le suivant sur un 4-cube :

$$t_{tot}(n) = 4c(n) + K_4 d(n) = 2c(n) + \frac{n}{16} d(n)$$

Ce résultat peut être généralisé pour un hypercube de taille quelconque.

Proposition : le temps d'exécution total pour calculer en parallèle n noyaux d'exécution de taille d(n) sur un m-cube avec la stratégie de diffusion asynchrone à 1-paquet est :

$$t_{tot}(n) = \frac{m}{2} c(n) + \frac{n}{p} d(n)$$

Preuve : la définition de t_{tot} est $t_{tot}(n) = mc(n) + K_m(n)$.
Le système linéaire à résoudre pour obtenir les K_i est le suivant :

$$\sum_{i=0}^{m} \lambda_i K_i = n \text{ avec } \lambda_i = C_m^i$$
$$K_i d(n) = K_m d(n) + (m-i)c(n), \text{ pour } i=0 \text{ jusqu'à } m$$

Nous obtenons alors :

$$n = \sum_{i=0}^{m} C_i^m (m-i) \frac{c(n)}{d(n)} + \sum_{i=0}^{m} C_i^m K_m$$

Ce qui conduit à :

$$2^m K_m = n - \frac{c(n)}{d(n)} \sum_{i=0}^{m} C_i^m (m-i) = n - \frac{c(n)}{d(n)} m 2^{m-1}$$

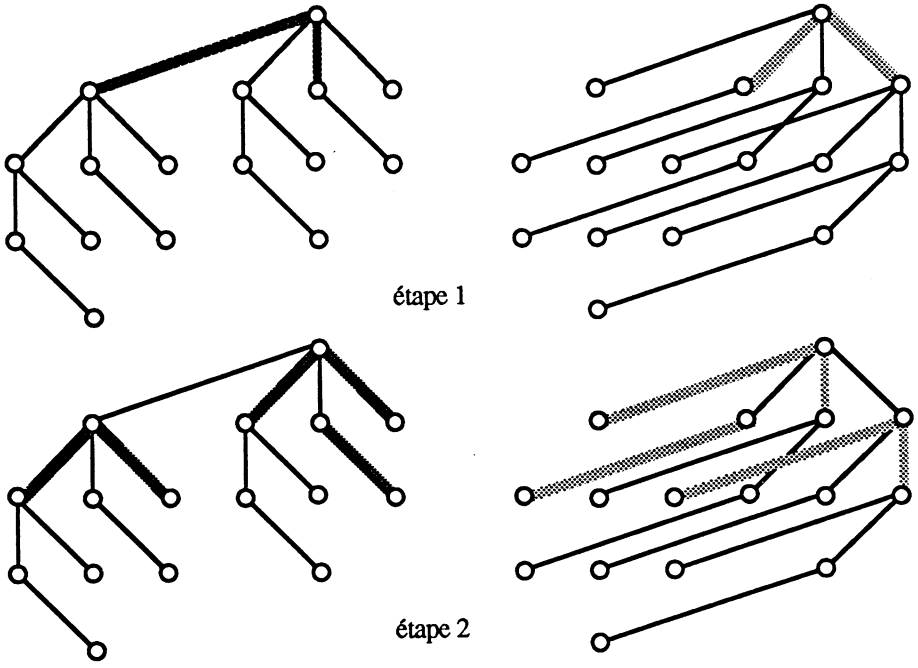
Et finalement :

$$K_m = \frac{n}{p} - \frac{m\, c(n)}{2\, d(n)}$$

qui permet de conclure.

4.3.2.2. Les stratégies à q-paquet

Pour $1 \leq q \leq \log_2(p)$, nous pouvons définir, par extension, des stratégies à q paquets, comme une amélioration de la statégie à 1-paquet, et un intermédiaire entre cette première stratégie et la méthode de diffusion rotative. L'amélioration consiste à diminuer les temps de communication en envoyant de plus petits messages et en les pipelinant. Les noyaux de calcul restent les mêmes, mais maintenant la taille des messages à envoyer a diminué. La façon de trouver les équations est la même.
Les stratégies à q-paquet sont tout d'abord présentées sur le même exemple que précédément, la figure ci-dessous détaille la stratégie 2-paquet dans le cas d'un 4-cube.



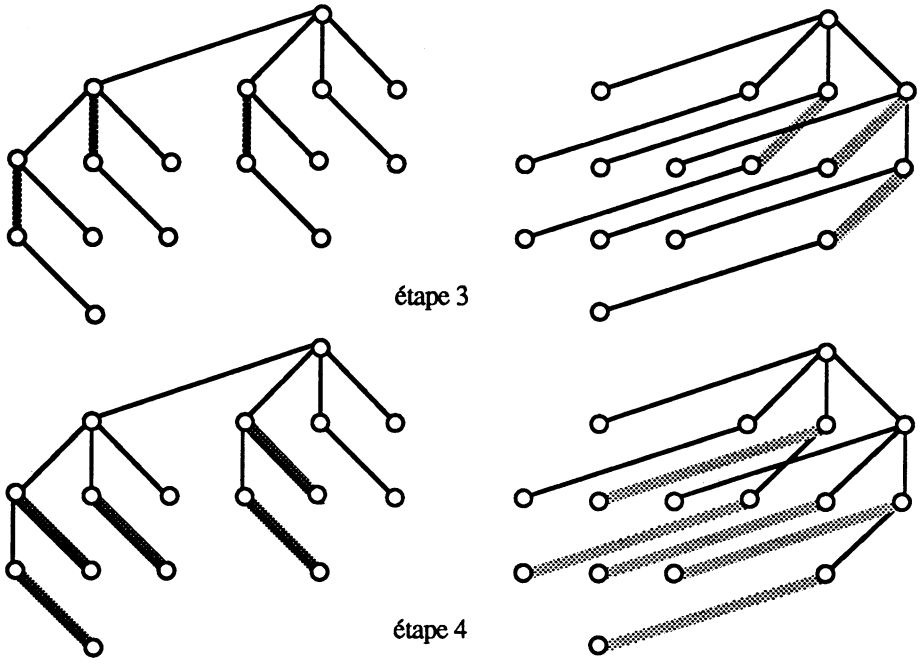


Figure 63 : stratégie de diffusion asynchrone à 2-paquet sur un 4-cube

La figure ci-dessous schématise les temps d'occupation des processeurs dans le cas de la stratégie à 2-paquet sur un 4-cube.

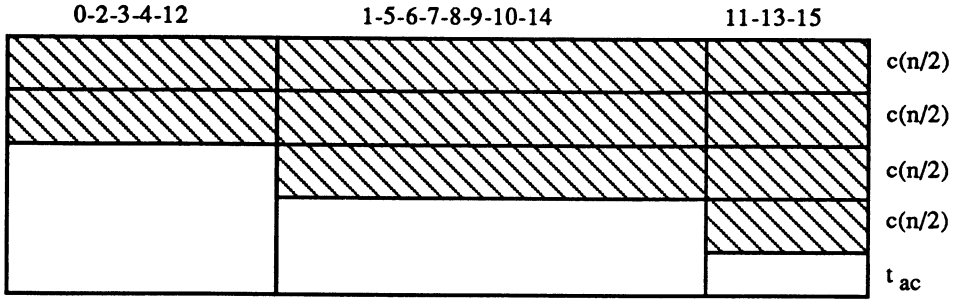


Figure 64 : temps d'occupation des processeurs pour la stratégie à 2-paquet sur un 4-cube

Sous les mêmes hypothèses que pour la srtatégie à 1-paquet et avec les mêmes notations, on obtient les valeurs suivantes pour les λ_i :

$$\lambda_1 = 0 \qquad \lambda_2 = 5 \qquad \lambda_3 = 8 \qquad \lambda_4 = 3$$

Les équations s'obtiennent comme précédément :

$$5K_2 + 8K_3 + 3K_4 = n$$
$$K_2 d(n) = K_4 d(n) + 2c(n/2)$$
$$K_3 d(n) = K_4 d(n) + c(n/2)$$

Les solutions correspondantes sont :

$$K_2 = \frac{n}{16} + \frac{7\,c(n/2)}{8\,d(n)}$$
$$K_3 = \frac{n}{16} - \frac{c(n/2)}{8\,d(n)}$$
$$K_4 = \frac{n}{16} - \frac{9\,c(n/2)}{8\,d(n)}$$

Le temps d'exécution total est alors de :

$$t_{tot}(n,2) = 4c(\frac{n}{2}) + \frac{n}{16}d(n) - \frac{9}{8}c(\frac{n}{2}) \text{ avec } c(\frac{n}{2}) = \frac{n}{2}\tau + \beta.$$

Cet algorithme garantit l'utilisation d'arêtes disjointes et peut ainsi être facilement généralisé à q-paquet (bien sûr si m est un multiple de q, l'algorithme sera beaucoup plus simple). Notons cependant que dans le cas de q=2, la meilleure stratégie est fournie par une progression gloutonne sur chaque arbre ; mais une telle stratégie n'est pas généralisable du fait des conflits possibles sur les arêtes.

On peut généraliser la stratégie à q-paquet sur un m-cube avec la propriété suivante.

Proposition : le temps d'exécution total pour calculer en parallèle n noyaux de calculs de taille d(n) sur un m-cube avec la stratégie de diffusion asynchrone à q-paquet est :

$$t_{tot}(n,q) = mc(\frac{n}{q}) + \frac{n}{p}d(n) + R(n,q)$$

où R(n,q) est une fonction négative et croissante en q. Dans cette expression, lorsque $m\frac{n}{q}\tau$ décroit, R(n,q) croit (avec q). Notons que :

$$R(n,m)=0 \qquad\qquad R(n,1) = -2\frac{c(n)}{p}\sum_{i=1}^{m} C_i^m(m-i).$$

Preuve : rappelons tout d'abord l'expression du temps total d'exécution :

$$t_{tot}(n,q) = t_c(n,q) + t_{ac}(n,q)$$

Le temps de communication $t_c(n,q)$ pour des envois de q paquets sur un m-cube peut s'exprimer comme suit :

$$t_c(n,q) = m(\frac{n}{q}\tau+\beta)$$

Nous pouvons donc en déduire que :

$$t_{tot}(n,q) = m\frac{n}{q}\tau + m\beta + t_{ac}(n,q)$$

Comme le volume total de calculs est nd(n), et que $pt_{ac}(n,q) \leq nd(n)$, nous avons alors :

$$t_{ac}(n,q) \leq \frac{n}{p}d(n) + R(n,q) \text{ où } R(n,q) \leq 0$$

Le dernier point à démontrer est la croissance en q de la fonction R. Rappelons pour ce faire :

$$S = nd(n) \qquad S = \sum_{i=q}^{m} S_i(q)$$

La surface $S_i(q)$ peut s'exprimer comme (voir figure 65 ci-aprés) :

$$S_i(q) = \lambda_i[(m-i)c(\frac{n}{q})+t_{ac}(n,q)] \qquad \text{où } c(\frac{n}{q}) = \frac{n}{q}\tau + \beta$$

Notons que $\lambda_i(q)$ est une fonction unimodale en i, que $\lambda_q(q)$ est une fonction croissante en q et finalement que $\lambda_m(q)$ est une fonction croissante en q.
L'expression précédente conduits aux deux équations suivantes :

$$nd(n) = \sum_{i=q+1}^{m} \lambda_i(q+1)[(m-i)c(q+1)+t_{ac}(n,q+1)]$$
$$nd(n) = \sum_{i=q}^{m} \lambda_i(q)[(m-i)c(q)+t_{ac}(n,q)]$$

Ce qui permet alors d'obtenir (avec $\sum_{i=q}^{m} \lambda_i(q) = \sum_{i=q+1}^{m} \lambda_i(q+1) = p$) :

$$p\big(t_{ac}(n,q+1)-t_{ac}(n,q)\big) = -\sum_{i=q+1}^{m} \lambda_i(q+1)(m-i)c(\frac{n}{q+1}) + \sum_{i=q}^{m} \lambda_i(q)(m-i)c(\frac{n}{q})$$

Comme c(q)>c(q+1), nous avons l'inégalité :

$$p\big(t_{ac}(n,q+1)-t_{ac}(n,q)\big) > -\sum_{i=q+1}^{m} \lambda_i(q+1)(m-i)c(\frac{n}{q}) + \sum_{i=q}^{m} \lambda_i(q)(m-i)c(\frac{n}{q})$$

Il est alors possible d'en déduire que :

$$t_{ac}(n,q+1) - t_{ac}(n,q) > 0$$

Comme $R(n,q) - R(n,q+1) = t_{ac}(n,q+1) - t_{ac}(n,q)$, nous avons finallement prouvé que

R(n,q) est une fonction croissante en q.

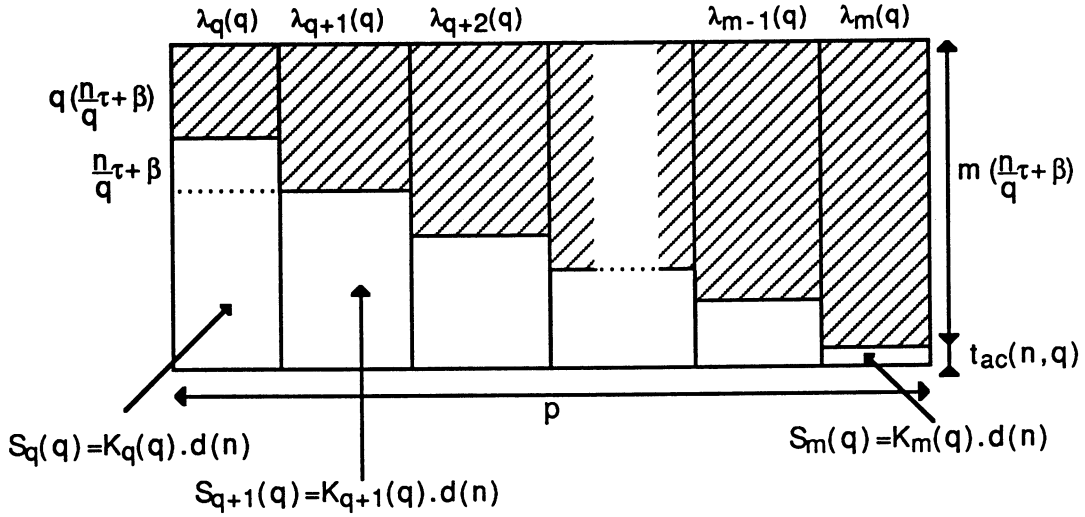


Figure 65 : temps d'occupation des processeurs pour la stratégie à q-paquet

- La figure 66 donne une comparaison des temps d'exécution entre les diverses stratégies asynchrones et la diffusion rotative, pour un hypercube à 16 processeurs.

Stratégies	Temps d'exécution
1-paquet	$\frac{5}{2}(n\tau+\beta) + \frac{n}{16}d(n)$
2-paquets	$\frac{3}{2}n\tau+3\beta + \frac{n}{16}d(n)$
3-paquets	$\frac{19}{16}n\tau+\frac{57}{16}\beta + \frac{n}{16}d(n)$
rotatif	$n\tau+4\beta + \frac{n}{16}d(n)$

Figure 66 : comparaison entre stratégies asynchrones et diffusion rotative

4.3.2.3. Quelques expérimentations

Les expérimentations numériques été effectuées sur l'hypercube à 16 processeurs FPS T20 sont présentées sur la figure ci-dessous.

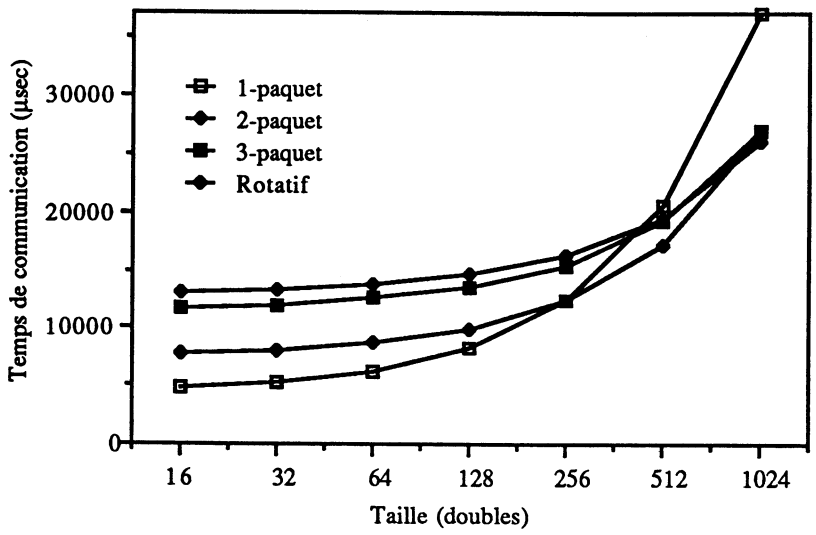


Figure 67 : expérimentation de différentes stratégies de diffusion sur FPS T20

Cette figure compare la diffusion rotative (qui correspond en fait dans le cas d'un 4-cube à une stratégie asynchrone à 4-paquet) aux 3 autres stratégies asynchrones sur le FPS T40. Ainsi plus le nombre de paquets est grand, meilleure est la stratégie, et il apparaît clairement que la stratégie rotative est meilleure pour de grands vecteurs.
Cependant toutes les stratégies asynchrones sont plus rapides si la taille du vecteur est inférieure à 1024, ce qui est dans la plupart des cas une valeur limite pour les problèmes pratiques (dû à la quantité de mémoire disponible sur chaque processeur). Pour les tailles des problèmes usuels (vecteurs de tailles 256 ou 512), la stratégie à 2-paquet est la plus efficace.

4.3.3. Avantages et inconvénients

La diffusion rotative correspond à la stratégie asynchrone à m-paquet : tous les processeurs deviennent libres à la dernière étape, de telle sorte qu'ils ne peuvent pas commencer les calculs locaux avant la fin des communications. Les nouveaux algorithmes asynchrones apparaissent comme très efficaces pour les petits problèmes (en fait de la taille des problèmes pratiquement implantables sur des ordinateurs parallèles à mémoire distribuée). Mais pour de très grands problèmes, la meilleure stratégie semble être la diffusion rotative. Cependant notre analyse montre qu'elle peut être considérée comme un cas particulier de la stratégie asynchrone à q paquets (avec q=m) et qu'elle est asymptotiquement optimale parmi toutes ces stratégies. Quoi qu'il en soit, dans ce cas, le facteur $\frac{n}{p}d(n)$ de $t_{tot}(n)$ devient prédominant.

Chapitre 5

Expérimentations sur le FPS T40

Où l'on expérimente tout d'abord une stratégie distribuée d'une résolution par le gradient conjugué préconditionné, puis au vu des résultats obtenus plusieurs stratégies parallèles avec résolutions locales sur les processeurs de la méthode du gradient conjugué préconditionné pour systèmes linéaires successifs sont comparées. Les problèmes des entrées/sorties (qui devraient être pris en compte dans une application en vraie grandeur) sont discutés.

5.1. Implantation du gradient conjugué

5.1.1. Exposé du problème

Le FPS T40 étant une machine MIMD à 32 processeurs vectoriels dotés de communications relativement peu performantes en comparaison avec les possibilité des unités de calcul vectoriel, toute stratégie privilégiant les calculs locaux au dépend des calculs répartis est très performante.
Dans le cas d'une résolution de systèmes linéaires successifs (ce à quoi peut se ramener un problème de répartition de charge dans un réseau électrique à topologie variable), il faut privilégier des résolutions locales (pour des problémes d'une taille inférieure à 128x128) s'effectuant en parallèle sur chacun des processeurs. Le cas dual, une résolution distribuée sur tous les processeurs, n'est envisageable que pour des problèmes ne tenant pas sur un processeur, et dans la pratique uniquement pour des problèmes de grandes tailles permettant une bonne charge de calculs sur chacun des processeurs.

Deux types d'implantations de l'algorithme du gradient conjugué préconditionné sont décrites dans cette section : la première, locale à chaque processeur est la plus intéressante si la taille du problème n'excède pas celle de la mémoire locale à chacun des processeurs ; la seconde, distribuée sur l'ensemble des processeurs n'est envisageable que pour des problèmes de grandes tailles.

5.1.2. Implantation locale sur un processeur vectoriel

Cette implantation locale a été uniquement réalisée dans le cas de problèmes de taille 128x128 : l'unité de calcul vectoriel est en effet optimisée pour des vecteurs de longueur 128 et les calculs vectoriels sont pénalisés pour des tailles inférieures. Le cas de tailles égales à des multiples de 128 n'a pas été envisagé pour des questions de place mémoire : chaque processeur ne dispose en effet que de 1 mégaoctet de mémoire.

Trois niveaux de vectorisation ont été implantés : scalaire (utilisant les Transputers disponibles sur le T40, c'est à dire des T414 dépourvus d'unités de calculs flottants), vectoriel de haut niveau (appel en C de fonction "generic") et vectoriel de bas niveau (utilisation des "vector forms"). Le préconditionnement choisi est ici de type diagonal (pour des raisons de simplicité, le préconditionnement définitif est exposé en 2.4). La vectorisation de l'algorithme est donnée ci-dessous.

```
/* initialisations */
x[]=0                                   affectation vectorielle
g[]=-b[]                                affectation vectorielle
```

```
/* résolution du système Mz=g avec M=D */
z[]=g[]/A[][]                                   division vectorielle
/* initialisations : fin */
d[]=z[]                                         affectation vecorielle
/* itérations du GCP */
répéter
   {
   NbIter=NbIter+1
   /* multiplication matrice-vecteur v=A.d */
   pour i variant de 0 à n-1
      v[]=A[i][]*d[]                            produit scalaire
   /* mise à jour de x et g */
   OldError=g[]*z[]                             produit scalaire
   Psgd=g[]*d[]                                 produit scalaire
   Psvd=v[]*d[]                                 produit scalaire
   Rho=Psgd/Psvd                                division scalaire
   x[]=x[]-Rho*d[]                              saxpy-like
   g[]=g[]-Rho*v[]                              saxpy-like
   /* résolution dyu système Mz=g avec M=D */
   z[]=g[]/A[][]                                division vectorielle
   /* mise à jour de d */
   NewError=OldError                            affectation scalaire
   Beta=NewError/OldError                       division scalaire
   d[]=z[]-Beta*d[]                             saxpy-like
jusqu'à sqrt(NewError)<Precision ou NbIter=n
```

Les temps obtenus pour une itération du gradient conjugué sont les suivants :

scalaire	2.88 s
generic	0.0396 s
vector-forms	0.0256 s

Cette version de l'algorithme du gradient conjugué avec préconditionnement diagonal requiert $2n^2+18n+4$ opérations arithmétiques. On a donc les performances suivantes (avec une matrice 128x128) :

scalaire	0.012 Mflops
generic	0.89 Mflops
vector-forms	1.38 Mflops

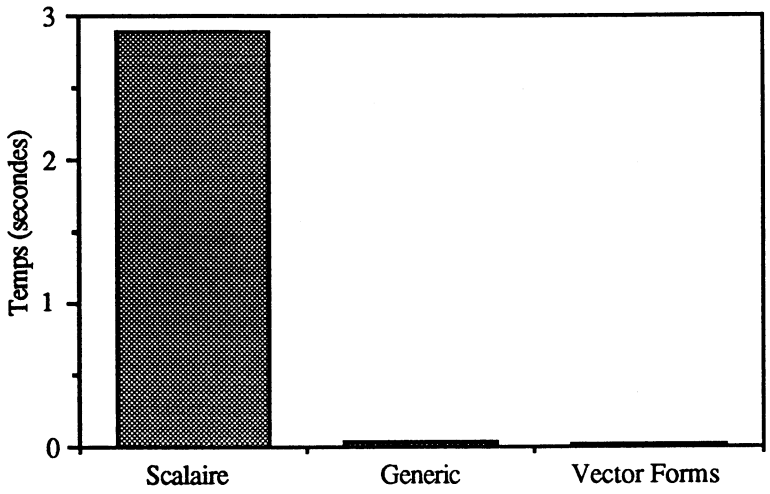


Figure 68 : temps d'exécution d'une itération du gradient conjugué local

La trés forte disproportion entre la version scalaire et les versions vectorielle est imputable à l'abscence de FPU sur le T414, les calculs flottants étant effectués de manière logicielle.

5.1.3. Implantation distribuée sur tous les processeurs

Les disproportions existant entre calculs vectoriels et communications font du FPS T40 une machine à gros grain : les noyaux de calculs qui coûtent n opérations arithmétiques sur un problème de taille n (tels les saxpys ou les dots) ne gagnent pas à être exécutés de façon répartie sur tous les processeurs, en effet le gain obtenu en parallélisant les calculs est bien plus faible que le coût des communications. Par contre, les noyaux Blas 2 ou Blas 3 [Dong], d'un coût de n^2 ou n^3 opérations arithmétiques (tels les produits matrice-vecteur ou matrice-matrice), peuvent être parallélisés avec profit. En effet, on suppose alors que pour de tels noyaux les données sont sur place (elles ne ne peuvent de toute façon plus tenir de façon réaliste sur un seul processeur).
Ainsi, l'implantation choisie du gradient conjugué distribué sur le FPS T40 ne parallélise que le produit matrice-vecteur (unique noyau d'exécution Blas 2 de l'algorithme).

Les explications suivantes sont fournies pour une implantation distribuée sur les 32 processeurs vectoriels du FPS T40. Pour des raisons techniques (le *dot* est plus rapide que le *saxpy* sur les unités de calculs vectoriels du T40), la version *dot* du produit matrice vecteur a été implantée sur des matrices de taille 1024x1024. Les 32 processeurs vectoriels du FPS T40 se partagent donc les 1024 lignes de la façon suivante.

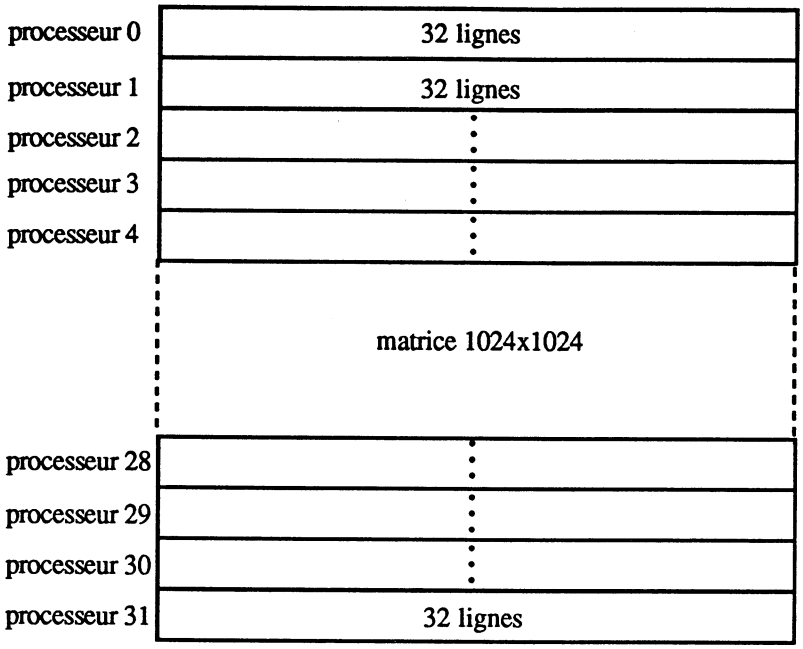


Figure 69 : répartition des données entre les processeurs

Chaque processeur stocke donc 256 ko de données, ce qui assure un volume de calculs locaux suffisants.
Le produit matrice vecteur x=A.b se déroule donc de la façon suivante :

- envois du vecteur b du processeur 0 aux 31 autres processeurs (diffusion)
- calculs locaux des produits scalaires locaux (produits scalaires)
- envois des résulats partiels au processeur 0 (regroupement)

Le processeur maître (ici numéroté 0) envoie donc tout d'abord un vecteur de taille 1024 à tous les autres processeurs, il s'agit donc d'une communication de type diffusion (ou broadcast). Les produits scalaires locaux sont ensuite effectués sur les unités de calculs vectoriels (avec une efficacité maximale puisque 1024 est un multiple de 128). Enfin, la collecte des résultats partiels (soit 32 sous-vecteurs de taille 32 à fusionner en un vecteur de taille 1024 sur le processeur 0) nécessite une communication de type regroupement (ou gather) afin que tous les processeurs fournissent leurs résultats partiels au processeur maître.
Les deux communications (diffusion et regroupement) sont implantées via des routines *one-to-all* et *all-to-one* car les routines de plus bas niveau type *one-to-one* ne peuvent pas servir de base à des implantations efficaces d'algorithmes de communications sophistiqués du fait de leur trop faibles performances. Plus précisément, l'hypercube a été configuré en grille 8x4 et l'on enchaine deux routines (*one-to-all* pour la diffusion et *all-to-one* pour le regroupement) comme indiqué sur la figure ci-dessous.

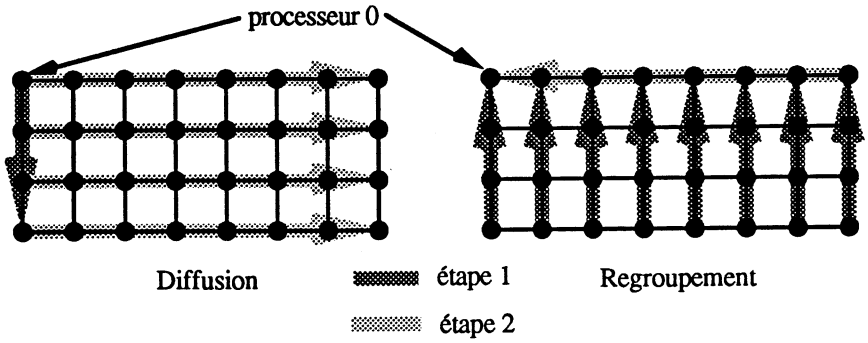


Figure 70 : implantation de la diffusion et du regroupement

Notons que sur une machine disposant de primitives efficaces de communications entre processeurs voisins, il aurait été plus intéressant d'utiliser par exemple un algorithme de diffusion asynchrone (voir 4.3.2 par exemple).

La configuration choisie est bien évidément une grille 8x4 (32 processeurs et matrice 1024x1024), les résultats obtenus sont les suivants (pour une itération du gradient conjugué) :

32 processeurs	1.0184 s

Cette implantation du gradient conjugué permet d'atteindre seulement 2.1 mflops. Ce qui s'explique par la proportion importante de code séquentiel : sur un temps d'exécution total de 1 s, seulement 0.163 s (soit 16 %) sont utilisées pour des calculs distribués.

Remarquons qu'une version vraiment distribuée (où tous les calculs vectoriels sont effectués de façon répartie) donnerait des performances sensiblement plus faibles sur le T40. En effet, la disproportion entre communications et calculs est telle qu'il est plus intéressant d'exécuter un calcul vectoriel (typiquement un dot ou un saxpy) localement que de le distribuer, et ce jusqu'à des vecteurs de taille d'environ 16000. Alors pour des problèmes de taille 128x128 ...

Les performances obtenues sur le gradient conjugué montrent que l'implantation distribuée des méthodes directes décrites au chapitre 2 ne sont pas envisageables sur ce type de machine : tous les noyaux de calculs de ces méthodes sont de grain n. Elles n'ont donc pas été implantées sur le FPS T40.

5.2. Implantations du gradient conjugué pour systèmes linéaires successifs

La description détaillée de la méthode de résolution choisie par résoudre ces systèmes linéaires successifs (par gradient conjugué avec un préconditionnement particulier) est donnée en 2.4.4. Elle nécessite un préconditionnement par factorisation de Cholesky dont l'implantation sur le FPS T40 est étudiée ci-aprés. L'implantation du gradient conjugué préconditionné pour systèmes linéaires successifs sur le FPS T40, dont les résolutions par le gradient conjugué s'effectuent localement sur chacun des processeurs, est décrite en 5.1.2. Aucune implantation du gradient conjugué préconditionné pour systèmes linéaires successifs distribué n'a été réalisée, les performances du gradient conjugué distribué sur tous les processeurs n'étant pas du tout prometteuses (cf 5.1.3).

5.2.1. Résolution vectorielle par la méthode de Cholesky

La factorisation de Cholesky permet de résoudre des systèmes linéaires Ax=b, où la matrice A (de taille nxn) est symètrique définie positive, c'est à dire :

$$\forall x \in \mathbb{R}^n,\ x \neq 0,\ \langle Ax, x \rangle > 0$$

où $< . , . >$ désigne le produit scalaire euclidien.
La méthode de Cholesky (ou de Crout [GoVL]) permet alors de décomposer la matrice A sous la forme $A=LDL^t$, où L est une matrice triangulaire inférieure à diagonale unité et D une matrice diagonale. La résolution du système Ax=b s'effectue ensuite en trois étapes :

$$L\,y = b$$
$$D\,z = y$$
$$L^t\,x = z$$

Le premier système (triangulaire inférieur) se résout par une substitution avant, le second (diagonal) par division et le dernier (triangulaire supérieur) par substitution arrière.

5.2.1.1. Vectorisation de la factorisation de Cholesky

La factorisation de Cholesky dont l'algorithme est composé de trois boucles imbriquées, peut être décrit à base de saxpy ou de dots. Pour des raisons techniques (les unités de calculs vectorielles du T40 sont plus efficaces sur des produits scalaires), la version dot de la factorisation de Cholesky a été privilégiée ; elle est donnée ci-dessous en pseudo-code (le vecteur D étant préalablement initialisé à 0) :

```
pour k variant de 1 à n
```

```
pour p variant de 1 à k-1
   R[p]=D[p]*L[k][p]
D[k]=A[k][k]
pour p variant de 1 à k-1
   D[k]=D[k]-L[k][p]*R[p]
pour i variant de k+1 à n
   L[i][k]=A[i][k]
   pour p variant de 1 à k-1
      L[i][k]=L[i][k]-L[i][p]*R[p]
   L[i][k]=L[i][k]/D[k]
```

La vectorisation de l'algorithme est triviale (on vectorise chacune des opérations de la boucle sur k, le pseudo-code ci-dessous est constitué de fonctions "generic") :

```
pour k variant de 1 à n
         vvo(L[k], D, multiplication et négation, R, k-1)
         R[k]=1.0
         vvoro(R, L[k], multiplication et addition, D[k], k-1)
         pour i variant de k+1 à n
            vvoro(R, L[i], multiplication et addition, L[i][k], k+1)
            L[i][k]=L[i][k]/D[k]
```

La version scalaire de la factorisation de Cholesky nécessite $\frac{n^3}{3}+O(n^2)$ opérations arithmétiques. L'unité de calcul vectoriel possédant un additionneur et un multiplieur pipelinés, il est possible d'effectuer une multiplication simultanément avec l'addition du pas précédent. Cela permet donc de diviser par deux le nombre d'opérations de la boucle la plus interne de l'algorithme. Le coût de l'algorithme vectoriel est donc en $\frac{n^3}{6}+O(n^2)$ opérations arithmétiques pipelinées.

5.2.1.2. Vectorisation des résolutions triangulaires

Il reste maintenant à vectoriser les résolutions triangulaires et diagonales.

Intéressons nous tout d'abord à la substitution avant (permettant de résoudre Ly=b) dont le code scalaire est donné ci-dessous (en pseudo-code) :

```
y[1]=b[1]
pour i variant de 2 à n
   y[i]=b[i]
   pour j variant de 1 à i-1
      y[i]=y[i]-L[i][j]*y[j]
```

La vectorisation de cet algorithme consiste uniquement à remplacer la boucle sur j par un produit scalaire de longueur i-1. Le coût de cet algorithme est de $O(n^2)$ opérations arithmétiques.

L'algorithme scalaire de la résolution diagonale(Dz=y) est donné ci-dessous (en pseudo-code) :

```
pour i variant de 1 à n
   z[i]=y[i]/D[i]
```

La vectorisation est immédiate : z est le résultat de la division vectorielle de y par D. Le coût de cet algorithme est de O(n) opérations arithmétiques.

Intéressons nous enfin à la substitution arrière (permettant de résoudre $L^t x=z$) dont le code scalaire est donné ci-dessous (en pseudo-code) :

```
x[n]=z[n]
pour i variant de 2 à n
   x[i]=z[i]
   pour j variant de n-1 à 1
      x[i]=x[i]-Lt[i][j]*x[j]
```

La vectorisation de cet algorithme consiste uniquement à remplacer la boucle sur j par un saxpy de longueur n. Le coût de cet algorithme est de $O(n^2)$ opérations arithmétiques.

5.2.1.3. Résultats expérimentaux

Les résultats expérimentaux suivants sont réalisés sur des vecteurs alignés en début de banc et des matrices dont chaque ligne est alignée en début de banc (afin d'optimiser l'utilisation du VPU, cf 3.3.4). L'unité de calcul vectoriel est programmée par des fonctions generic.

Taille	8	16	32	64	100	128
Temps (s)	0.0109	0.0382	0.1408	0.5385	1.2944	2.1080

On en déduit alors les performances suivantes.

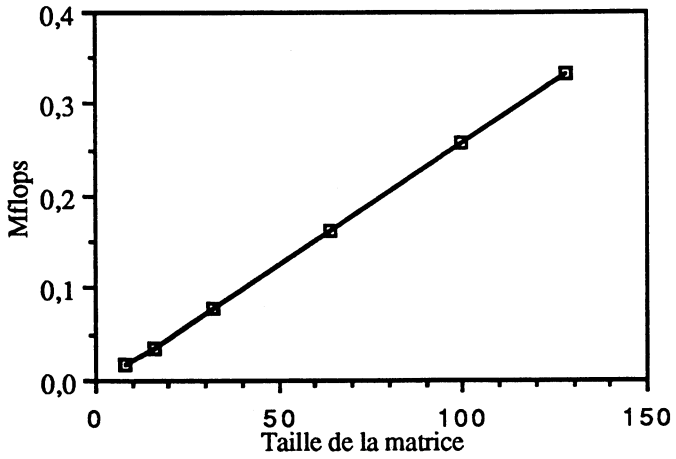


Figure 71 : performances de la méthode de Cholesky vectorisée

Le maximum, de 0.33 MFlops, est naturellement obtenu pour un problème de taille 128x128.

Remarquons que cette implantation comporte des noyaux d'exécution locaux en $O(n^2)$, ce qui n'est pas efficace sur cette machine où l'on ne doit exécuter localement que des calculs en O(n), et encore uniquement lorsque n est inférieur à 10000.

5.2.2. Le gradient conjugué pour systèmes successifs

Il apparaît de façon immédiate qu'une implantation parallèle synchrone du gradient conjugué préconditionné pour systèmes linéaires successifs ne pourra jamais être très bonne : pour une série de systèmes numérotés de k à $k+i_0$, le nombre d'itérations des $k+i_0-1$ gradients conjugués préconditionnés par la factorisation de Cholesky du système k va nécessairement varier. On a donc une majorité de processeurs qui attend les processeurs chargés des derniers systèmes (ceux les plus proches de $k+i_0$), comme le montre le diagramme d'occupation suivant (avec les 16 processeurs qui finissent de recevoir la factorisation de Cholesky du système k à l'instant t).

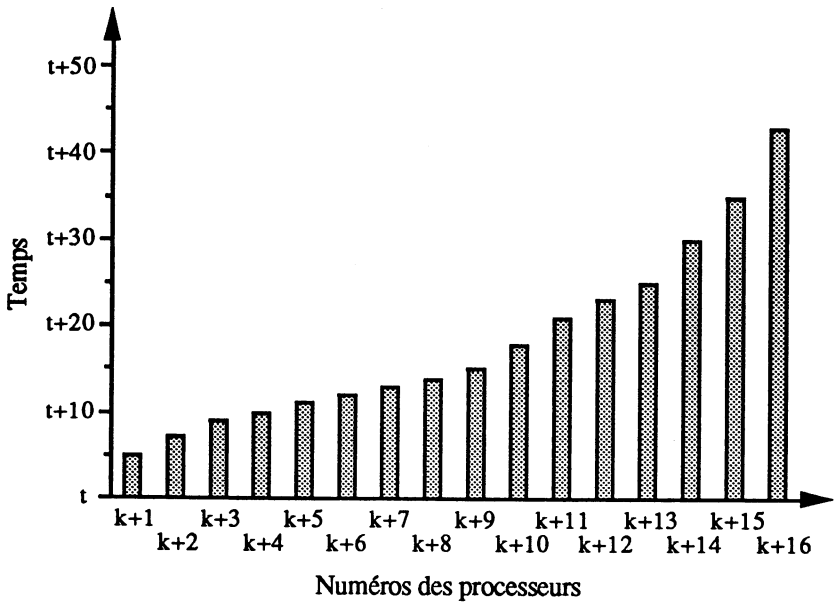


Figure 72 : diagramme d'occupation des processeurs pour la parallélisation synchrone du gradient conjugué préconditionné pour systèmes linéaires successifs

Il faut donc envisager une implantation asynchrone du gradient conjugué préconditionné pour systèmes linéaires successifs, en s'inspirant par exemple des algorithmes de [BlTr] (décrits en 4.3). Le cas qui nous intéresse est cependant différent des hypothèses décrites en 4.3.1.1 : ici toutes les communications entre processeurs voisins ont le même poids et le coût des noyaux d'exécution est fixé, et l'on ne peut jouer que sur leurs affectations sur les différents processeurs.

5.2.2.1. Les problèmes d'entrées / sorties

Avant d'étudier l'implantation du gradient conjugué préconditionné pour systèmes linéaires successifs sur le FPS T40, il faut étudier les contraintes liées aux entrées/sorties : les matrices correspondant aux systèmes linéaires successifs doivent être placées sur les processeurs, et les résultats de chaque système linéaire doivent être rapatriés vers le processeur privilégié (celui qui effectue les factorisations de Cholesky, le processeur 0 par exemple). Deux cas de figure sont envisageables.
On peut dans un premier temps générer sur place (c'est à dire sur chacun des processeurs) les modifications permettant de passer d'un système A_i à un système A_j. En effet ces modifications proviennent de changements dans la topologie du réseau et l'on peut donc effectuer localement les calculs nécessaires à l'obtention du nouveau système (cf 1.2). Ainsi l'obtention des nouvelles matrices ne nécessite aucune communication entre processeurs. Il est tout simplement nécessaire, aprés la résolution d'un groupe de systèmes linéaires par le gradient conjugué préconditionné, de prévoir une phase

supplémentaire de calculs locaux (voir figure 73).
La seconde méthode envisagée consiste à communiquer à chaque processeur les modifications permettant de passer de sa matrice actuelle à la matrice du prochain système linéaire qu'il aura à résoudre (voir figure 73). Ce cas de figure implique une modification de la phase des communications qui est propre à l'algorithme de communications utilisé. Par exemple, les modifications ne seront pas du tout les mêmes si la topologie des processeurs est une grille ou un arbre binaire : la symétrie de la grille fait que seule la taille des messages peut être augmentée, par contre dans le cas de l'arbre binaire tous les messages n'ont alors plus la même taille.

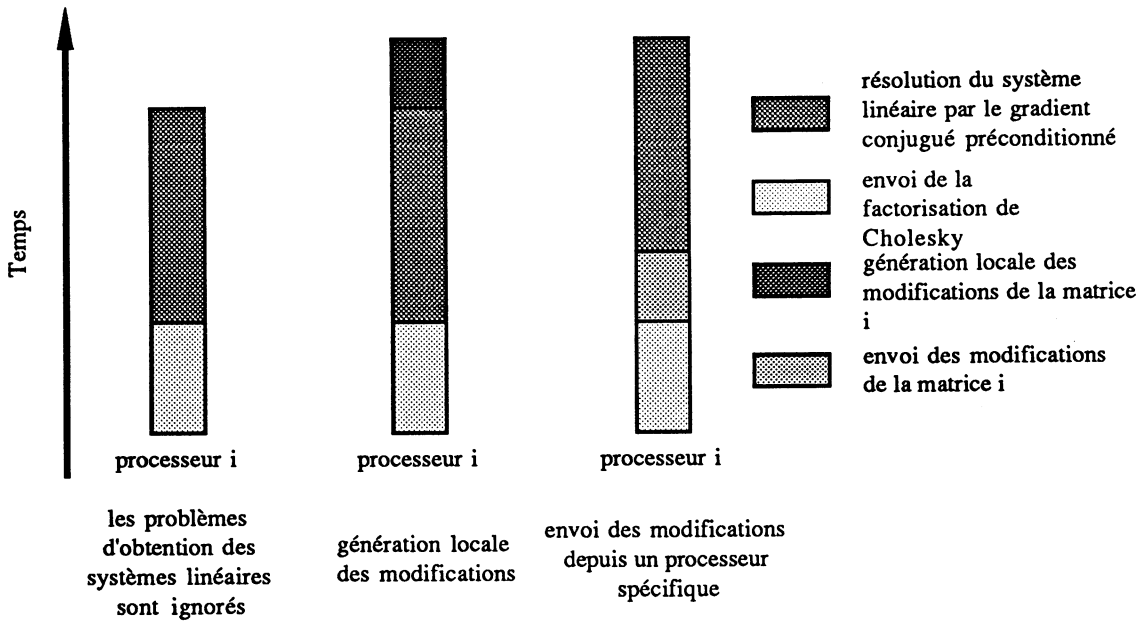


Figure 73 : deux approches des problèmes liés à l'obtention des systèmes linéaires successifs

Le rapatriement des résultats de chacun des systèmes linéaires sur le processeur privilégié n'est en fait qu'une communication de type regroupement. Afin de ne pas interférer avec la résolution des systèmes linéaires, on suppose (et c'est dans la pratique très souvent le cas) que le nombre N de systèmes linéaires à résoudre est tel que les résultats peuvent être gardés sur place et n'être rapatriés sur le processeur 0 que lorsque les N sytèmes linéaires ont été tous résolus.

5.2.2.2. Implantation asynchrone simple

L'idée de base consiste à placer les matrices nécessitant un nombre élevé d'itérations du gradient conjugué préconditionné d'autant plus loin du processeur qui a effectué la factorisation de Cholesky que ce nombre d'itérations est grand. Cependant, les coûts des communications ne sont pas uniformes :

- les processeurs chargés d'effectuer des résolutions par le gradient conjugué préconditionné recoivent la matrice A_i et la factorisation L_{i0} ($i_0<i$), soit $\frac{3n^2}{2}$ réels

- le processeur chargé de la factorisation de Cholesky reçoit seulement la matrice A_{i_1} (i_1>i), soit n^2 réels

Les difficultés rencontrées pour programmer une implantation efficace fait que le nombre de systèmes résolus successivement par le gradient conjugué préconditionné a été choisi comme un multiple de 16 (afin de faciliter l'implantation sur FPS T40). Cependant cette décision arbitraire a été guidée par les deux remarques suivantes.
Remarquons tout d'abord que le nombre de systèmes à résoudre entre chaque réinitialisation est un paramètre sur lequel il est possible de jouer : si l'on diminue le nombre de résolutions successives par le gradient conjugué préconditionné, celles-ci seront peu coûteuses (nombre d'itérations faible), cependant il faudra alors accepter plus souvent une réinitialisation, elle très coûteuse. Il est ainsi possible d'ajuster, dans certaines limites et en acceptant une perte d'efficacité théorique, la méthode du gradient conjugué préconditionné pour systèmes linéaires successifs au nombre de processeurs dont on dispose.
Ensuite, les expérimentations effectuées sur des machines séquentielles montrent que pour des matrices de taille 128x128 mal conditionnées, il est possible de résoudre successivement entre 25 et 30 systèmes linéaire par le gradient conjugué préconditionné, et que pour des matrices bien conditionnées ce nombre peut aller jusqu'à plus d'une centaine (voir figure 74).

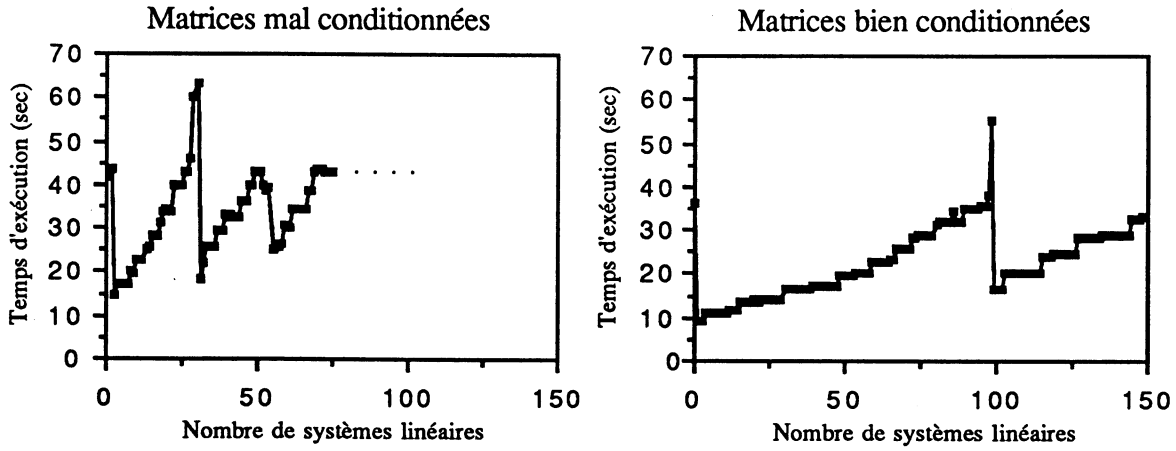


Figure 74 : influence du conditionnement des matrices

Les implantations exposées ci-aprés l'ont toutes été sur 16 processeurs afin de simplifier explications et figures. Bien entendu, l'implantation de ces méthodes sur 32 processeurs est tout à fait possible, bien que moins simple à décrire.

L'exemple ci-dessous détaille l'implantation sur 16 processeurs avec des diffusions asynchrones en 1-paquet, en supposant que les Δ_i sont générés sur place. Les réinitialisations sont donc effectuées tous les 16 systèmes ; on suppose pour l'exemple que le système A_{i-1} est résolu par la méthode Cholesky, ensuite les systèmes A_i jusqu'à

A_{i+14} sont résolus par le gradient conjugué préconditionné, et enfin le système A_{i+15} est résolu par la méthode Cholesky.
La figure 75 illustre l'algorithme de diffusion.

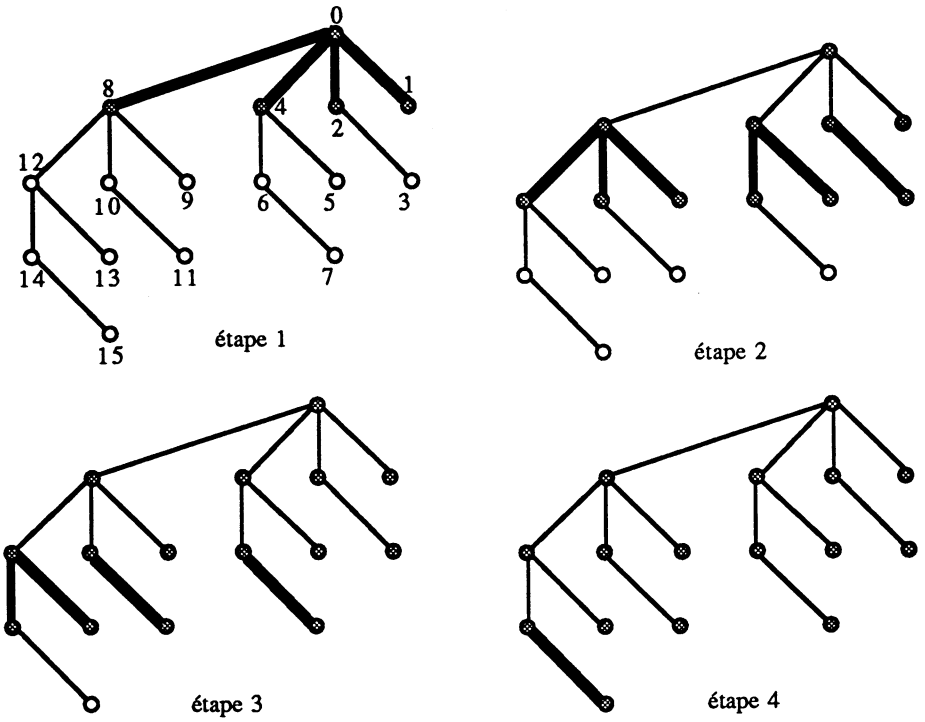


Figure 75 : stratégie de diffusion 1-paquet

Les processeurs deviennent disponibles pour les calculs locaux de la façon suivante :

- étape 1 : aucun (le processeur 0 émet)
- étape 2 : processeurs 0 et 1
- étape 3 : processeurs 2, 3, 4, 5, 8 et 9
- étape 4 : processeurs 6, 7, 10, 11, 12 et 13
- étape 5 : processeurs 14 et 15

Ainsi la matrice A_{i+15} (qui sera factorisée par Cholesky) peut être affectée soit au processeur 0, soit au processeur 1 ; afin de simplifier les communications (toutes portent donc sur $\frac{3n^2}{2}$ réels) le processeur 0 a été choisi. Les tâches restantes sont affectées de la façon suivante :

- processeurs 1 : A_{i+14}
- processeurs 2, 3, 4, 5, 8 et 9 : de A_{i+13} jusqu'à A_{i+8}
- processeurs 6, 7, 10, 11, 12 et 13 : de A_{i+7} jusqu'à A_{i+2}
- processeurs 14 et 15 : A_{i+1} et A_i

Pour l'étape i, l'algorithme peut donc s'écrire (en pseudo-code), en prenant les processeurs suivant leur profondeur dans l'arbre :

```
/* communications profondeur 1 */
si Proc=0
   pour i variant de 0 à 3
      OneToOne(O,2^i, L_{i-1}, (128^2/2)*8)
   GénèreD(A_{i+15})
   FactorisationCholesky(A_{i+15})
   RésolutionsTriangulaires(A_{i+15})
si Proc=1
   GradientConjuguéPréconditionné(L_{i-1},A_{i+14})
/* communications profondeur 2 */
si Proc=2
   OneToOne(2,3, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+13})
si Proc=4
   pour i variant de 1 à 2
      OneToOne(4,4+i, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+11})
si Proc=8
   pour i variant de 0 à 2
      OneToOne(8,8+2^i, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+9})
si Proc=(3,5,9)
   GradientConjuguéPréconditionné(L_{i-1},(A_{i+12},A_{i+10},A_{i+8}))
si proc=6
   OneToOne(6,7, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+6})
/* communications profondeur 3 */
si proc=10
   OneToOne(10,11, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+5})
si proc=12
   pour i variant de 1 à 2
      OneToOne(12,12+i, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+3})
si Proc=(7,11,13)
   GradientConjuguéPréconditionné(L_{i-1},(A_{i+6},A_{i+4},A_{i+2}))
/* communications profondeur 4 */
si proc=14
   OneToOne(14,15, L_{i-1}, (128^2/2)*8)
   GradientConjuguéPréconditionné(L_{i-1},A_{i+1})
si Proc=15
```

`GradientConjuguéPréconditionné(`L_{i-1}`,`A_i`)`

Un des points faibles de cette implantation réside dans l'initialisation : la factorisation du système 0 par la méthode de Cholesky est effectuée par le processeur 0, tous les autres restant inactifs (ils ont besoins de L_0 comme préconditionnement pour leurs résolutions).

Le temps d'exécution obtenu est de 2.2161 s pour la résolution parallèle de 16 systèmes linéaires. La figure ci-dessous donne le détail des temps d'exécution obtenus pour chaque processeur.

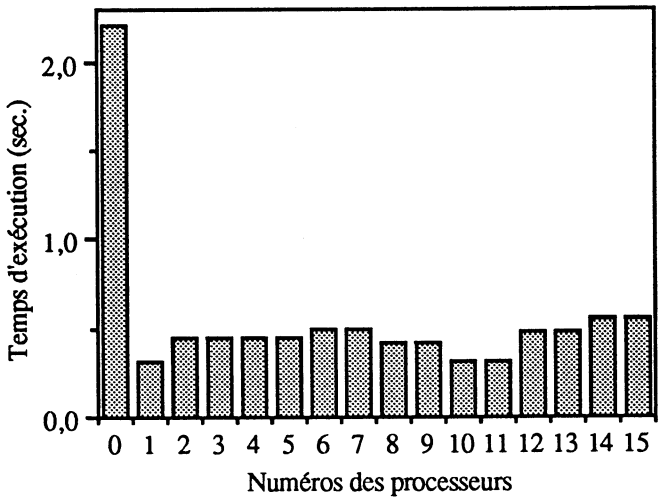


Figure 76 : temps d'exécution du gradient conjugué préconditionné pour systèmes linéaires successifs sur 16 processeurs avec 16 systèmes successifs

Il apparaît donc que cette implantation n'est pas bonne du tout : les réinitialisations (environ 4 fois plus coûteuses que les résolutions par le gradient conjugué préconditionné) sont beaucoup trop fréquentes et nuisent beaucoup à l'efficacité :

$$E_{16} = \frac{t_{seq}}{16 t_{16}} = 0.139$$

Il faut cependant remarquer que cette implantation a été écrite pour le cas de systèmes très mal conditionnés ; et que si l'on peut résoudre un nombre plus élevé de systèmes linéaires sans effectuer de réinitialisation, on sera alors plus rapide que par des résolutions classiques par la méthode Cholesky par exemple.
C'est pourquoi, une amélioration naturelle de cette implantation peut être obtenue en réinitialisant beaucoup moins souvent : tous les 32 ou tous les 64 systèmes par exemple. On obtient alors des temps d'exécution qui sont toujours de 2.2161 s , mais la figure 77 (dans le cas d'une réinitialisation tous les 32 systèmes linéaires) montre que l'efficacité obtenue n'est que légèrement meilleure :

$$E_{16} = \frac{t_{seq}}{16t_{16}} = 0.272$$

Les processeurs restent en effet inactifs environ la moitié du temps.

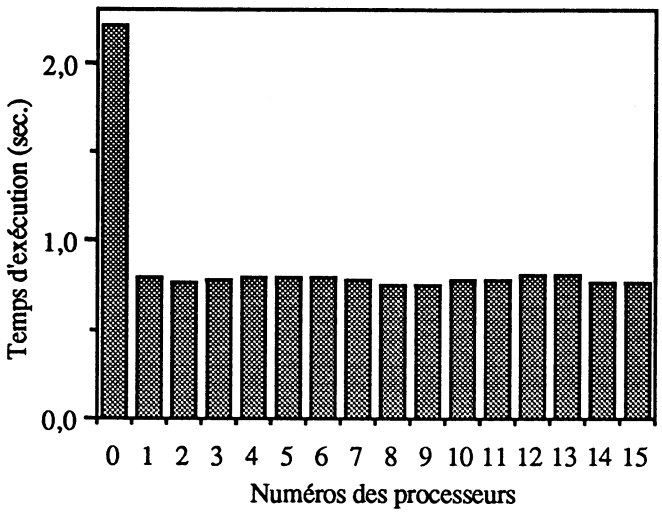
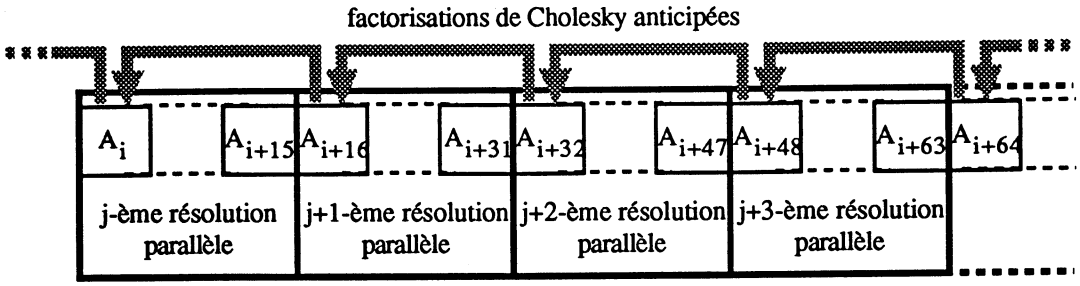


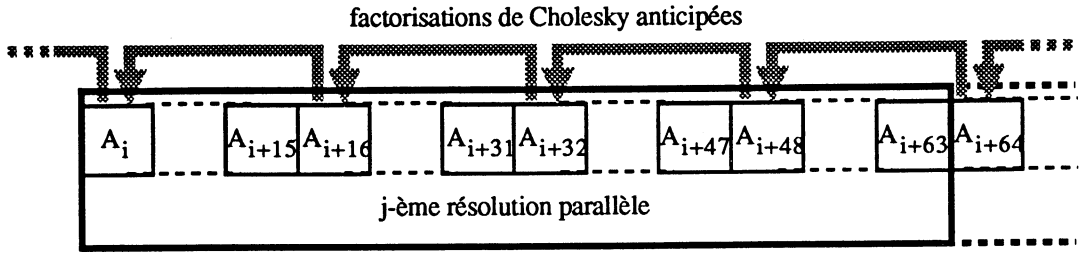
Figure 77 : temps d'exécution du gradient conjugué préconditionné pour systèmes linéaires successifs sur 16 processeurs avec 32 systèmes successifs

5.2.2.3. Implantation synchrone multi-arbres

Si l'on suppose toujours que les systèmes linéaires sont suffisamment mal conditionnés pour nécessiter une réinitialisation tous les 16 résolutions, il est encore possible d'améliorer l'algorithme asynchrone simple précédent en calculant simultanément la factorisation de Cholesky pour plusieurs groupes de 16 systèmes linéaires, et en résolvant en parallèle par gradients conjugués le nombre correspondant de systèmes linéaires.
L'algorithme asynchrone simple proposé précédemment laisse approximativement 15/16 des processeurs inactifs 1/4 du temps. Il semble donc à priori intéressant d'essayer de résoudre 4 fois plus de systèmes linéaires comme le montre la figure ci-dessous.



Résolution asynchrone simple



Résolution synchrone multi-arbres

Figure 78 : deux stratégies d'implantation de l'algorithme du gradient conjugué préconditionné pour systèmes consécutifs sur le T40

La phase de communications est modifiée : on diffuse cette fois 4 factorisations de Cholesky à tous les processeurs chargés de résolutions par le gradient conjugué. Il est cependant nécessaire de garder le coût des communications aussi faible que possible, faute de quoi tout le bénéfice de cette nouvelle implantation sera perdu.

Rappelons qu'il est possible de plonger 4 arbres binaires à arêtes disjointes dans un hypercube de degré 4, même si les sommets des arbres sont différents. Ainsi la stratégie adoptée pour la diffusion des 4 préconditionnement fait appel à 4 arbres binaires à arêtes disjointes, de façon à éviter des conflits sur les liens des Transputers lors des communications. Cependant, il n'est plus aussi intéressant qu'auparavant d'utiliser une stratégie asynchrone : les proceseurs communicant sur plus de liens à des instants différents, les gains réalisés par une stratégie asynchrone ne sont guère significatifs (surtout face au surcroit de difficultés de programmation).

C'est pourquoi l'implantation retenue est synchrone : les calculs débutent dès que les communications sont globalement terminées et la répartition des tâches sur les différents processeurs prend explicitement en compte ce facteur de synchronisme (au contraire de la précédente implantation où l'aspect asynchrone était spécifiquement utilisé).

On obtient un temps d'exécution qui est toujours de 2.2161 s, et la figure ci-dessous représente l'occupation des processeurs.

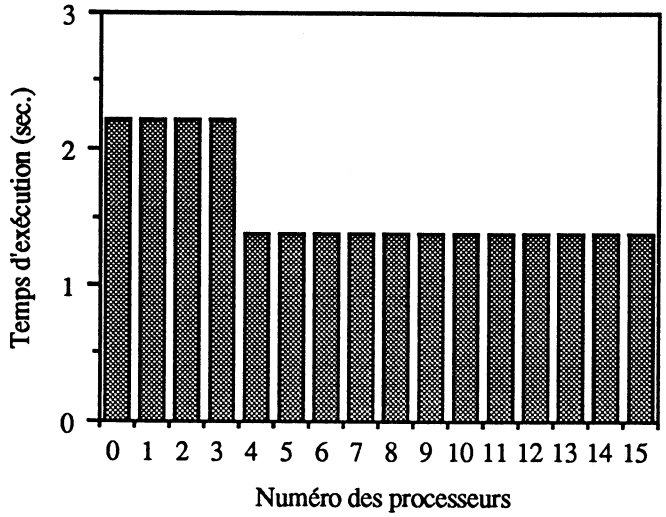


Figure 79 : occupation des processeurs pour la stratégie synchrone multi-arbres (16 syst. cons.)

L'efficacité obtenue est encore meilleure :

$$E_{16} = \frac{t_{seq}}{16 t_{16}} = 0.557$$

Cependant le meilleur algorithme que l'on puisse obtenir sous les hypothèses que l'on s'est fixé (en 5.2.2.1) est une stratégie multi-arbres avec des réinitialisations tous les 32 systèmes linéaires. En effet l'on dispose alors de suffisamment de résolutions par gradient conjugué préconditionné pour contrebalancer à la fois le poids des communications et des décompositions de Cholesky (très couûteuses). La figure ci-dessous fournit les temps d'occupations des processeurs.

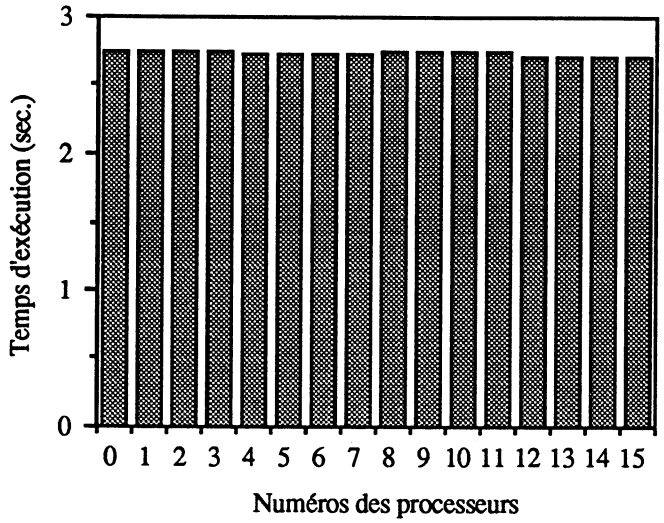


Figure 80 : occupation des processeurs pour la stratégie synchrone multi-arbres (32 syst. cons.)

L'efficacité obtenue dénote une parallélisation très efficace du problème :

$$E_{16} = \frac{t_{seq}}{16 t_{16}} = 0.908$$

Le tableau ci-dessous récapitule les résultats numériques obtenus :

		temps d'exécution	mflops	efficacité
asynchrone	16 syst. cons.	2.216 s	2.067	0.139
	32 syst. cons.	2.216 s	5.183	0.272
multi-arbres	16 syst. cons.	2.216 s	8.269	0.557
	32 syst. cons.	2.74 s	16.767	0.908

Rappelons pour finir que les unités vectorielles du T40 sont données par FPS comme ayant une puissance de crête de l'ordre de 10 Mflops ce qui donnerait une puissance maximale théorique de 160 Mflops pour 16 processeurs. Ce qui est très nettement supérieur aux 16 Mflops obtenus avec la meilleure implantation. Cependant cette différence s'explique : tout d'abord le problème étudié nécessite une importante phase de communications, ensuite les VPU atteignent des puissances de l'ordre de 10 Mflops pour des vecteurs de tr)s grande taille (supérieure à 10000) alors que les matrices actuelles ont une taille nettement inférieure (128x128).
Ce phénomène montre que sur un problème pratique, il est difficile de dépasser un dixième de la puissance théorique de la machine (en effet l'efficacité de l'implantation synchrone multi-arbres pour 32 systèmes linéaires consécutifs a une efficacité de 0.9).

Chapitre 6

Architecture SIMD massivement parallèle : la CM2

Où l'on décrit la Connection Machine, ordinateur SIMD massivement parallèle, en insistant sur sa spécificité et sur l'approche algorithmique nécessaire pour tirer pleinement partie des ressources de cette machine.

6.1. Présentation de la Connection Machine

La Connection Machine (CM2) est une machine parallèle SIMD créee par TMC (Thinking Machines Corporation) : la même instruction est effectuée de manière synchrone sur des données multiples. Cette approche permet d'atteindre un haut degré de parallèlisme, qui est la seconde caractéristique importante de la CM2 : elle peut contenir jusqu'à 65536 processeurs.
Cette présentation de la Connection Machine est basée sur les informations contenues dans [DTW1].

6.1.1. Principes de fonctionnement

Le principe de programmation de la CM2 consiste à associer à chaque élément d'un ensemble (qui va être traité par le programme) une unité de contrôle et de calcul qui est ici communément appelée processeur. Tous les éléments de cet ensemble sont alors traités simultanément. Par exemple, les éléments d'une matrice sont distribués un par processeur et un calcul global (typiquement l'addition de deux matrices) est effectué en parallèle sur chacun des éléments.
Lorsque le nombre d'éléments à répartir est supérieur au nombre de processeurs, ceux-ci partagent alors leur temps et leur mémoire de façon à pouvoir gérer plusieurs éléments. Ce phénomène est complétement transparent au programmeur qui ne se préoccupe que du nombre total d'éléments de son ensemble. Ce type de partage des ressources de la CM2 est désigné sous le terme de *processeurs virtuels*, et le nombre d'éléments gérés par un processeur physique est appelé *VP ratio*.

Les processeurs de la CM2 ont un fonctionnement SIMD mais il leur est néanmoins possible de ne pas exécuter l'instruction commune : sur chaque processeur virtuel un bit (ou *context flag*) permet de gérer cette possibilité.
Les processeurs de la CM2 ont chacun leur mémoire propre et par conséquent n'ont pas directement accès au contenu de la mémoire des autres processeurs. Il leur est alors nécessaire d'avoir recours à des communications permettant des échanges d'informations entre processeurs.

6.1.2. Organisation globale de la CM2

Une Connection Machine comporte au maximum 65536 processeurs 1-bit (soit 2^{16}). Ces processeurs 1-bit sont issus d'une technologie éprouvée et robuste mais assez lente (leur fréquence d'horloge est faible). Les performances élevées que l'on peut obtenir d'une CM2 sont issues de la grande régularité des flots d'instructions et de la grande quantité de données.

On accède à la CM2 par un ordinateur frontal classique, en mode multi-utilisateurs (depuis la version 6.0 du système) où il est alors possible de travailler sur un sous-cube de 8, 16, 32 ou 64 kprocesseurs (voir figure 81). En pratique les programmes sont développés sur l'hôte, où leur partie séquentielle est exécutée : seule la partie du code parallèle est exécutée sur la CM2. Le mode de fonctionnement SIMD garantit simultanément l'exécution de la même instruction sur tous les processeurs et sur des données ayant toutes les mêmes emplacements en mémoire. Une horloge globale permet de synchroniser les instructions.

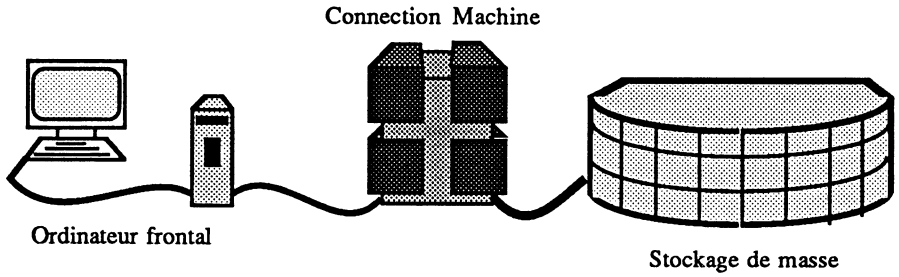


Figure 81 : organisation globale de la CM2

Les processeurs 1-bit sont regroupés par 16 modules de 32 processeurs sur une seule carte. Le module, constitué de 2 circuits de 16 processeurs avec leur unité de communication, d'une mémoire locale et d'une unité de calculs flottants est décrit sur la figure ci-dessous.

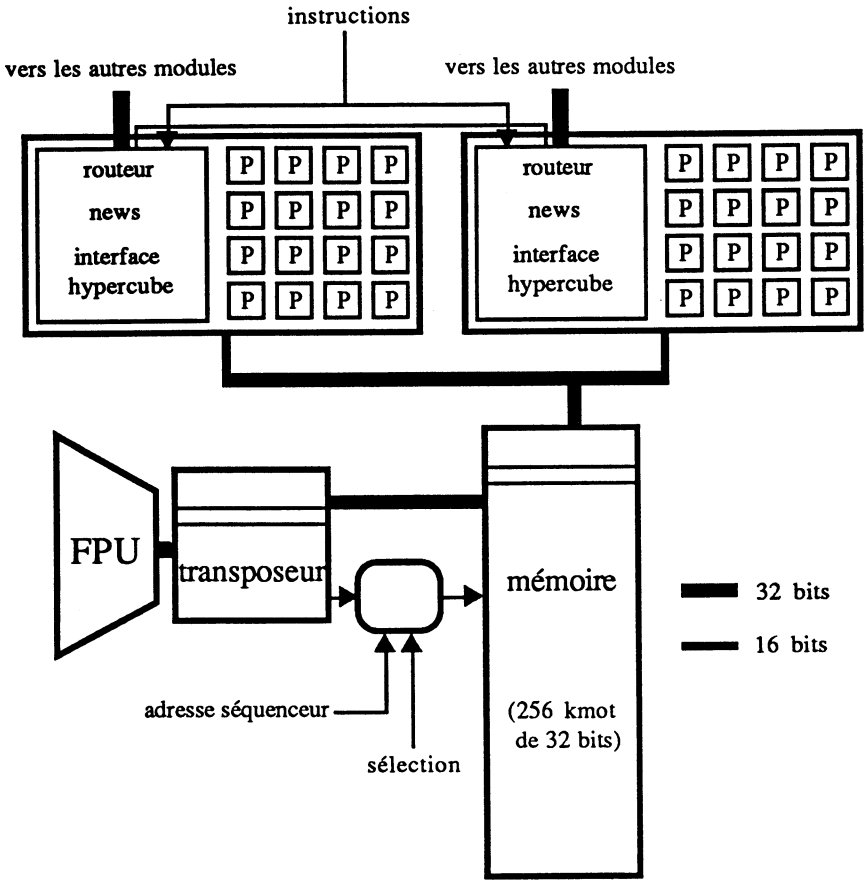


Figure 82 : organisation d'un module

Les modules sont reliés à l'ordinateur frontal via un séquenceur par deux bus (instructions et adresses). Les circuits de 16 processeurs 1-bit sont reliés entre eux par leurs unités de communications respectives suivant une topologie hypercube (de degré 12 pour la configuration maximale). Les 16 processeurs 1-bit d'un même circuit sont entièrement interconnectés par un réseau complet type butterfly (à 4 niveaux).

6.1.3. Les processeurs élémentaires

Il s'agit des processeurs 1 bit schématisés sur la figure ci-dessous.

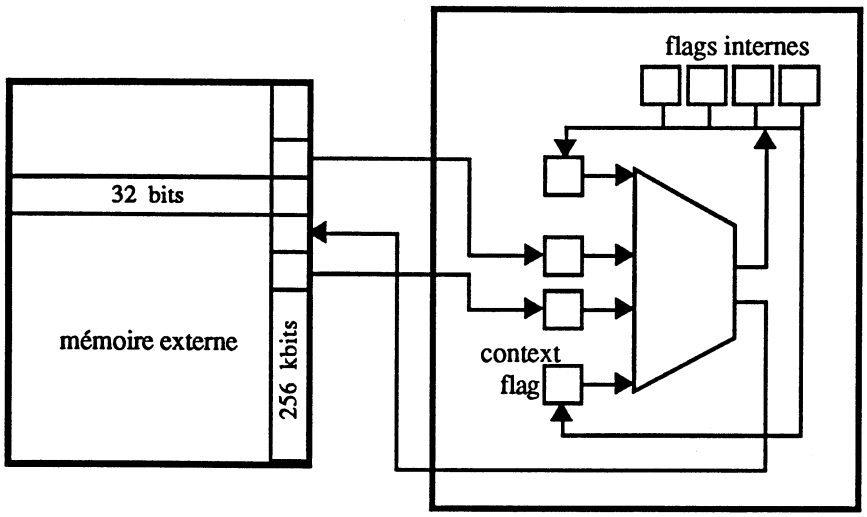


Figure 83 : processeur élémentaire de la CM2

Au cours d'un cycle d'horloge, ce processeur peut avoir accès :

- à deux bits de sa mémoire externe (celle-ci est décrite en 6.1.5) en lecture
- à un seul bit de sa mémoire externe en écriture
- à un de ses quatre flags internes en lecture
- à un de ses quatre flags internes en écriture

L'unité logique du processeur est capable d'effectuer toutes les opérations de $\{0, 1\}^3$ dans $\{0, 1\}^2$, soit 65536 opérations différentes. Dans la pratique, ces opérations sont tabulées dans une mémoire morte.
Les opérations en écriture sont conditionnées par un bit (*context flag*) : s'il est à 0, les instructions n'ont aucun effet (il existe cependant quelques instructions particulières qui passent outre).

6.1.4. Les routeurs

Chaque processeur de la CM2 peut échanger des informations avec n'importe quel autre, cependant ces deux processeurs ne sont pas forcément connectés par un lien physique ; aussi le message devra emprunter plusieurs canaux de communications pour atteindre son but. Il est donc nécessaire d'adjoindre à la CM2 un dispositif de routage pour chaque groupe de 16 processeurs.
Le routeur opère de la façon suivante : lorsqu'il reçoit un message en provenance d'un routeur voisin, il distribue les messages destinés aux 16 processeurs dont il a la charge et renvoie aux routeurs voisins les messages restant (y compris ses propres messages) en choisissant la direction appropriée à chacun de ces messages.

Si l'on considère le rôle des routeurs dans les communications entre processeurs virtuels

(qui sont seuls importants pour l'utilisateur), plusieurs cas peuvent se produire :

- les deux processeurs virtuels communiquant sont situés sur le même processeur élémentaire et la communication se limite à une opération de lecture-écriture en mémoire, donc pas d'intervention du routeur
- les deux processeurs virtuels sont implantés sur deux processeurs élémentaires différents mais appartenant à un même circuit de 16 processeurs, la communication se fait donc directement par le réseau butterfly, et là encore le routeur n'intervient pas
- enfin les deux processeurs virtuels appartiennent à deux proceseurs élémentaires situés sur deux circuits différents, le message emprunte, sous le contrôle des routeurs, le ou les liens reliant les deux cartes

Remarquons pour terminer que si l'on double le nombre de processeurs virtuels, le temps d'un schéma de communication générale donné augmente seulement d'un facteur inférieur à 2.

6.1.5. Les unités flottantes et les transposeurs

A chaque module de 32 processeurs est associé un FPU (au standard IEEE simple précision) qui accèlère les calculs flottants par un facteur 20. Les opérations sur les réels sont effectuées séquentiellement par mots de 32 bits issus d'un même processeur. Il est possible d'installer sur les derniers modèles de Connection Machine des FPU au format IEEE double précision (64 bits).

La mémoire ne permet d'accèder qu'à 1 bit sur chacun des 32 processeurs, au lieu d'un mot de 32 bits issu d'un seul processeur. C'est pourquoi un circuit spécial est interposé entre la mémoire et le FPU afin de transposer les données.
Il s'agit d'une matrice 32x32 qui permet de stocker et de réarranger les opérandes de l'unité flottante pour que ceux-ci entrent en série dans le FPU. En fait trois transposeurs sont physiquement présents à chaque entrée de l'unité flottante.

Notons pour finir qu'il est possible au plus bas niveau de programmation (en CMIS uniquement, cf 6.3.1.6) de stocker les nombres réels directement 1 bit par processeur élémentaire et de court-circuiter ainsi les transposeurs qui ne servent alors plus qu'à alimenter le FPU.

6.1.6. Les périphériques dédiés

La CM2 est une des rares machines parallèles à être équipée tout spécialement d'une

importante mémoire de masse à accès très performant : le *data vault*. Le transfert s'effectue avec un débit de 25 Mo/s. Le format des données est particulier au *data vault* et nécessite en général une coûteuse conversion. Ce dispositif est néanmoins très intéressant lorsque les programmes nécessitent des accès répétés à de gros volumes de données comme le traitement d'images par exemple.

La CM2 est aussi dotée d'une sortie graphique spécialisée très rapide : le contenu d'un champ (*frame buffer*) des processeurs est chargé directement dans un pixel à l'écran.

6.2. Les communications

6.2.1. Présentation des différents types de communications

Il est possible de classer les communications réalisables sur la CM2 en 4 classes :

- des communications générales
- des communications directes
- des communications spécialisées
- des communications entre les processeurs de la CM2 et l'ordinateur hôte

Il est bien entendu possible de mettre en communication n'importe quel processeur de la CM2 avec n'importe quel autre : les processeurs actifs peuvent recevoir un message du processeur de leur choix grâce à des instructions de type *get*, et dans le cas dual émettre des informations en direction du processeur de leur choix par des instructions de type *send*. Dans cette première classe de communications chaque processeur est désigné par une adresse qui lui est propre (*send-adress*).
Ces communications très générales sont coûteuses en temps car il y a de forts risques de conflits sur les canaux physiques de communication du fait de la grande complexité de ce schéma de communication et du volume d'informations échangées.

C'est pourquoi une seconde classe de communications directes et locales (et donc plus rapides) est disponible. Les processeurs sont disposés sur une grille et un processeur désigne son correspondant par son adresse relative dans la grille. Ces communications sont appelées *news*, par allusion avec les 4 points cardinaux (dans leurs appellations anglo-saxonnes).
Ainsi, ce mode de communication où la position relative d'un processeur vis à vis de son correspondant est uniforme sur toute la machine est plus rapide que les communications générales car on n'a ici qu'une translation de l'information, sans risque de conflit sur les canaux de communications.

La troisième classe de communications (spécialisées) disponible sur la CM2 permet de

combiner les informations d'un ensemble de processeurs puis de les redistribuer à un ensemble de processeurs (éventuellement différent). Notons que ce type de recombinaison peut se faire sur l'ensemble des processeurs, mais aussi simultanément sur des sous-ensembles de processeurs. Par exemple, supposons l'ensemble des processeurs disponibles organisé en grille ; on peut avec ce type de communication diffuser la somme d'une variable contenue sur toute une ligne de processeurs à cette ligne de processeurs, et cette diffusion peut s'effectuer en parallèle sur toutes les lignes de la grille.

Enfin la dernière classe de communications permet d'associer les processeurs de la CM2 avec l'ordinateur hôte, et réciproquement. Ainsi, on peut faire communiquer l'hôte avec un processeur particulier de la CM2, des informations calculées sur un groupe de processeurs de la CM2 peuvent être communiquées à l'hôte, et enfin l'hôte peut distribuer une même donnée à l'ensemble des processeurs de la CM2.

6.2.2. Communications générales

Deux instructions permettent les communications générales : *get* pour recevoir et *send* pour envoyer. Dans les deux cas, trois champs sont mis en jeu : *origine* contient le message à envoyer, *destination* le message à recevoir et *adresse* l'adresse du correspondant.

Un *get* permet à tous les processeurs actifs de recevoir un message d'un processeur de leur choix, quel que soit sont état (actif ou inactif). Lors d'un *get* chaque processeur actif *i* recoit donc dans son champ *destination* la valeur du champ *origine* du processeur dont il a l'adresse dans son champ *adresse*.

Il est possible lors d'une instrucion *send* (mais pas lors d'un *get*) qu'un processeur reçoive plusieurs messages à la même adresse mémoire. Il faut alors préciser la conduite à tenir sous peine d'avoir des résultats aléatoires. On peut distinguer trois solutions :

- si l'on est sûr qu'aucun conflit n'est possible, il est intéressant de le préciser car l'on peut alors gagner du temps en n'effectuant aucun contrôle de collision.

- il est aussi possible de concaténer tous les messages en un seul avec une opération de combinaison (comme l'addition ou un choix arbitraire), la figure ci-dessous illustre ce principe avec l'addition comme combinaison

- il sera possible (dans une version ultérieure) de recevoir les différents message dans un tableau, ce qui fera disparaître ce type de conflits

Notons qu'on peut passer outre la contrainte qui oblige tous les messages à avoir le même

champ *origine* dans tous les processeurs sources.

6.2.3. Communications de type news

Il existe deux instructions qui permettent les communications de type *news*, correspondant aux *get* et *send* précédent :

- *get-from-news* qui permet de recevoir sur la grille
- *send-from-news* qui permet d'émettre sur la grille

Ces deux instructions diffèrent de *get* et *send* par le fait que chaque processeur ne choisit plus son correspondant indépendamment des autres, mais qu'au contraire la direction de communication est maintenant globale.

Un *get-to-news* permet à tous les processeurs actifs de recevoir dans leur champ *destination* un message en provenance du champ *origine* du processeur voisin de position relative *déplacement* (même si ce processeur est inactif).

Les communications de type *news* font référence à une grille qui doit être créée et associée à un ensemble de processeurs avant toute communication de type *news*. Il est possible de changer la grille associée à un ensemble de processeurs virtuels (à condition que le nombre de processseurs du *vp-set* reste inchangé) ; cependant il n'existe alors pas de correspondance simple entre les positions d'un processeur avant et aprés le changement.

Contrairement à une communication générale *send*, l'instruction *send-to-news* ne permet à un processeur de recevoir qu'un seul message et rend donc inutile tout dispositif de gestion des conflits.

Remarquons pour terminer que si tous les processeurs d'une grille sont actifs, les deux instructions *send-to-news* et *get-to-news* sont équivalentes si leurs paramètres sont inversés.

6.2.4. Communications avec recombinaison

Ces communications, très utiles pour implanter des programmes d'algèbre linéaire font l'objet d'une présentation plus détaillée que précédément.

Il existe 3 instructions permettant d'effectuer des communications avec recombinaison : *spread*, *reduce* et *scan*. Les processeurs sont regroupés en ensembles ordonnés appelés "classes de recombinaison". Lors d'une communication, l'information est combinée à

l'intérieur de chaque classe par l'opération de recombinaison (associative) choisie, puis redistribuée.
Ces communications opèrent simultanément sur toutes les classes de recombinaison et il n'y a aucune communication entre deux processeurs s'ils n'appartiennent pas à la même classe.

L'instruction *spread* (ou diffusion) permet la combinaison de l'information de tous les processeurs d'une même classe de recombinaison, puis la diffusion à tous ces processeurs de l'information recombinée. La figure ci-dessous illustre cette instruction avec 6 processeurs disposés suivant la direction 1 d'une grille, et avec l'addition comme critère de recombinaison.

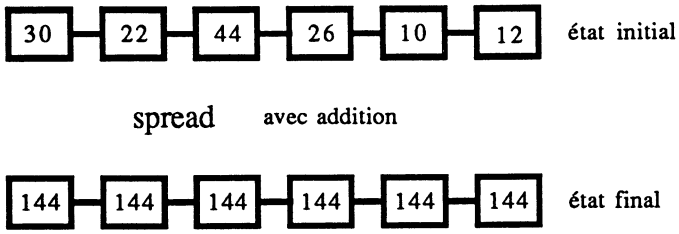


Figure 84 : l'instruction *spread*

L'instruction *reduce* permet à la combinaison de toutes les informations des processeurs d'une classe de recombinaison d'être retournée à un seul d'entre eux. La figure suivante illustre l'instruction *reduce* avec 6 processeurs disposés suivant la direction 1 d'une grille.

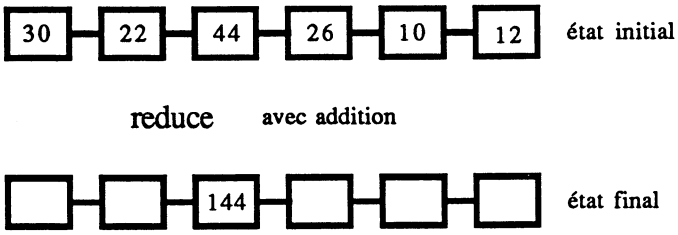


Figure 85 : l'instruction *reduce*

Une instruction *scan* permet à chaque processeur de la classe de recombinaison de recevoir la combinaison des informations des processeurs qui le précédent. Contrairement aux deux instructions précédentes (*spread* et *reduce*), l'instruction *scan* fait intervenir la notion d'ordre sur la classe de recombinaison.

6.2.5. Communications avec l'ordinateur hôte

Les deux instructions *read-from* et *write-to* permettent à l'ordinateur hôte d'échanger des informations avec un processeur virtuel (même si celui-ci est inactif) : lire la valeur d'un

champ du processeur virtuel et l'affecter à une variable de l'hôte, et affecter à un champ d'un processeur virtuel fixé la valeur d'une variable de l'hôte.

Les instructions *read-from-news-array* et *write-to-news-array* permettent d'échanger des informations entre un sous-ensemble d'une grille et l'ordinateur hôte. Ainsi, la valeur d'un champ sur le sous-ensemble de la grille peut être transféré vers un tableau de l'hôte ayant la même dimension que le sous-ensemble de la grille, et réciproquement. Les échanges d'informations ont lieu même si les processeurs virtuels sont inactifs.

L'instruction *global* permet (en association avec une opération de combinaison) de récupérer dans une variable de l'hôte la combinaison de toutes les valeurs d'un champ des processeurs virtuels actifs. Bien que cette instruction soit intéressante (on peut par exemple avec un *ou* comme opération de combinaison opérant sur le *context flag* déterminer s'il existe encore des processeurs actifs et donc effectuer un test de terminaison de programme), elle est en réalité peu efficace à cause du délai de communication entre le séquenceur et l'hôte.

6.2.6. Le communication compiler

Présentons pour terminer cette partie dédiée aux communications un outil qui devrait permettre de résoudre partiellement la lenteur des communications globales : le *communication compiler* (on peut consulter pour plus d'informations [Calv] ou [Dahl]).
En effet les communications globales conduisent à un schéma de communications sans stratégie globale où les messages transitent localement de routeur en routeur, ce qui conduit à un temps de communication important dû aux conflits qui peuvent se produire sur les canaux de communication.

Le *communication compiler* permet de créer, pour une communication générale donnée, un fichier contenant le graphe optimisé de cette communication. Lorsque la communication générale est exécutée, le programme lit le fichier contenant le graphe optimisé.
La lecture de ce fichier est pénalisante en temps, aussi cette méthode n'est-elle vraiment intéressante que pour des graphes de grande taille. Remarquons enfin que si cette communication générale est à l'intérieure d'une boucle, la lecture du fichier n'est effectuée qu'une seule fois, ce qui entraine un gain de temps certain.

6.3. Programmation de la CM2

6.3.1. Les langages de programmation

La Connection Machine peut être programmée dans un certain nombre de langages de haut niveau issus des langages usuels : *Lisp (la CM2 étant initialement destinée à l'intelligence artificielle, le *Lisp a été le premier langage développé), CM-Fortran et C*. Il existe aussi un niveau de programmation inférieur : ParIS (Parallel Instruction Set) qui peut être comparé à un assembleur très évolué. Enfin il est bien sur possible d'écrire un code de bas niveau avec CMIS (CM Instruction Set) qui est l'assembleur de la CM2.
Notons que pour les utilisateurs avertis, un bon compromis semble être l'emploi de C/ParIS qui est en fait un langage C séquentiel classique qui contient des primitives parallèles en ParIS.

6.3.1.1. Concepts parallèles

Les langages de programmation adaptés à des machines SIMD doivent pouvoir prendre en compte l'exécution d'une même instruction sur tout un ensemble de données. Ainsi, un programme écrit pour la CM2 est composé de deux types d'instructions : les instructions séquentielles classiques destinées à l'ordinateur hôte et les instructions parallèles destinées aux processeurs de la CM2.
Les instructions séquentielles sont celles de l'ordinateur hôte et elles opèrent sur des variables séquentielles, mises en œuvre par l'hôte. Les instructions parallèles permettent à tous les processeurs de la CM2 d'opérer simultanément sur la même portion de leur mémoire locale. Les opérateurs parallèles sont identiques à leurs homologues séquentiels et seul le fait de préciser qu'ils sont parallèles déclenche leur exécution sur la CM2 (et non pas sur l'hôte comme les opérateurs séquentiels).
Des opérations entre variables séquentielles et parallèles sont rendues possibles par la duplication de la variable scalaire sur les processeurs de la CM2 (ce mécanisme est transparent à l'utilisateur).

Les langages SIMD intégrent de plus des opérateurs de communication et de sélection d'un sous-ensemble de processeurs.
Les communications permettent d'échanger des informations stockées sur des processeurs différents.
L'opération de sélection s'opére en mettant à jour une variable booléenne parallèle (*context_flag*). Pour l'utilisateur tout se passe comme si seuls les processeurs actifs travaillent, alors qu'en fait tous les processeurs exécutent le même code mais seuls les processeurs actifs modifient leur mémoire locale.

6.3.1.2. Le *Lisp

Ce langage a été utilisé pour développer les expérimentations sur la CM2 (il est plus rapide que les autres langages de haut niveau d'environ 10% car les instructions *Lisp sont proches de la machine [DTW1]). C'est pourquoi ce langage est décrit ici plus en détails que ne vont l'être les autres.
Le *Lisp est une extension du Common Lisp à laquelle s'ajoute toutes les primitives propres à la Connection Machine (comme les communications ou les opérations de sélection de sous-ensembles de processeurs virtuels). Il reste bien entendu fidèle à la philosophie du Lisp qui veut que toute expression entre parenthèses soit évaluée et retourne un résultat (langage fonctionnel).

Le principal élément *Lisp exprimant le parallélisme est la *pvar* (ou *p*arallel *var*iable) : elle peut se concevoir comme un vecteur Lisp dont chaque élément serait affecté à un processeur virtuel. Une *pvar* est donc un objet Lisp constitué d'un champ de la mémoire locale de chacun des processeurs virtuels de la CM2. Ce champ peut contenir n'importe quel type d'élément Lisp. Cependant, fixer préalablement le format des *pvars* (par la fonction `pvar (type)`) permet de diminuer le temps d'exécution.
L'allocation mémoire des *pvars* est réalisée selon leur fonction : si une *pvar* est le résultat d'une fonction, son emplacement mémoire sera réservé dans la pile (et pourra être écrasé ultérieurement) ; mais si la *pvar* doit être conservée, elle sera affectée dans le tas à l'aide de la fonction `*set` (qui est l'extension parallèle du classique `set` du Lisp). Notons que l'on peut réserver de la place sur la pile à l'aide de la fonction duale `*let`.
La manipulation des *pvars* peut s'avérer périlleuse si l'utilisateur ne considère pas avec une certaine attention ce que retourne les fonctions. Une convention veut que si la fonction *Lisp commence par le symbole * le résultat retourné soit nil, et si l'opérateur est suivi du symbole *!!* le résultat est une *pvar*.

L'opération de sélection d'un ensemble de processeurs est basée en *Lisp sur le résultat de l'évaluation d'une *pvar* booléenne. Si le résultat est `nil`, le processeur est éliminé de l'ensemble courant des processeurs actifs.

*Lisp permet (comme la plupart des langages évolués) d'imbriquer les uns dans les autres plusieurs ordres de sélection différents, grâce à la sauvegarde du contexte. Il est aussi possible de sélectionner la totalité des processeurs, grâce à la fonction `*all` qui permet d'appliquer à tous les processeurs un bloc d'instructions sans perdre le contexte initial (restauré en fin d'exécution du bloc d'instructions).
Remarquons pour terminer ce paragraphe dévolu aux sélections, que lors d'un `*when`, `*if` ou `*cond`, l'ordre d'exécution des blocs conditionnels est envoyé à tous les processeurs, même s'ils ne sont pas actifs : il faut donc prendre garde lors de l'écriture de fonctions récursives.

Les quatre types de communications décrits en 6.2 sont utilisables en *Lisp. Remarquons que, toujours suivant le caractère * ou *!!* attaché à la fonction de communication, le résultat renvoyé peut être soit `nil` soit une *pvar*.
Précisons brièvement sur un exemple les paramètres d'une communication générale, dans le cas d'un *send* :

```
(*pset add value destination cube-adress notify collision-mode)
```

Cette fonction permet de tranférer une *pvar* `value` d'un processeur dans une *pvar* `destination` située sur un autre processeur (d'adresse `cube-adress`) en additionnant les entrées dans le processeur récepteur.

Les communications de type *news* nécessitent moins de paramètres (les problèmes de collision ne se posant pas), comme on peut le voir sur l'exemple suivant :

```
(*news source destination 0 1)
```

Cet exemple permet à la *pvar* `source` d'être transférée dans la *pvar* `destination` pour tous les processeurs actifs suivant un schéma de communication régulier : les processeurs actifs émettent à droite (*0*) sur la deuxième dimension (*1*) d'une 2-grille.

Les fonctions de communications avec recombinaison doivent préciser l'opération de recombinaison, une liste de processeurs (éventuellement segmentée) comme l'illustre l'exemple suivant :

```
(scan!! (self-adress!!) '+!! :direction :forward
        :segment segment :include-self t)
```

où `self-adress` est une *pvar* qui contient l'adresse du processeur courant, `'+!!` désigne l'addition comme opération de recombinaison, `:forward` le sens dans lequel la classe de recombinaison est parcourue, `segment` contient la classe de recombinaison et la segmentation, `include-self t` permet de choisir si le processeur émetteur est inclu dans la recombinaison.

Enfin, dans les communications avec l'hôte, on peut bien entendu charger ou décharger une *pvar* dans un tableau de l'ordinateur hôte. L'exemple ci-dessous permet de charger dans le tableau de l'hôte `destination` (dont la configuration est la même que celle des processeurs virtuels) la *pvar* `source` :

```
(pvar-to-array source destination)
```

Précisons pour terminer quelques aspects pratiques de l'exécution d'un programme *Lisp. Il est nécessaire d'initialiser un certain nombre de paramètres :

- mise à zéro des mémoires locales des processeurs de la CM2
- choix d'une topologie logique
- affectation d'un ensemble de processeurs virtuels (`vp-set`)

Ces opérations peuvent s'effectuer à l'aide de la fonction `(*cold-boot :initial-dimensions geometry)`.
Un programme *Lisp peut être soit interprété, soit compilé. L'influence de la compilation sur les performances n'est sensible qu'au niveau des instructions séquentielles exécutées sur la machine hôte qui autrement doivent être interprétées. Remarquons que les prises de temps ne sont pas intégrées au *Lisp et que l'on doit recourir à des instructions ParIS.
L'environnement de programmation *Lisp comprend un debogger et un système de gestion des erreurs d'exécution (qui permet de vérifier un programme et d'en suivre l'évolution). Le compilateur et l'interpréteur sont écrits en Common Lisp et sont donc facilement transportables sur toutes les machines possédant ce langage (Sun 4, stations Symbolics, Vax par exemple).

6.3.1.3. Le CM-Fortran

Le CM-Fortran est basé sur le Fortran 77 et inclus certains compléments offerts par le Fortran 8x. La caractéristique la plus marquante de ce langage de programmation est qu'il ne comporte pas à proprement parler de fonctions de communications : celles-ci sont incluses dans toutes les fonctions sophistiquées de manipulations de tableaux (qui sont briévement décrites ci-dessous).

Le Fortran 8x permet de traiter les tableaux en tant qu'entités (en quelque sorte équivalentes à des variables scalaires) et il en est de même pour le CM-Fortran. Sur la CM2, les tableaux sont stockés à raison d'un élément par processeurs virtuel, et la correspondance entre le tableau et la grille associée est naturelle : 2-grille pour une matrice par exemple.
La plupart des fonctions habituelles du Fortran permettant de traiter les tableaux sont accessibles en CM-Fortran. La principale restriction réside dans le fait que les tableaux doivent avoir les mêmes dimensions pour pouvoir y appliquer des opérateurs binaires comme l'addition ou la multiplication. Cette contrainte s'explique par le fait que le CM-Fortran alloue les mêmes processeurs virtuels aux éléments corespondant des deux tableaux, afin d'éliminer tout déplacement de données inutile.
Le CM-fortran dispose (en plus des fonctions algébriques précédentes) de nombreuses fonctions sophistiquées pour manipuler les tableaux :

- les fonctions caractéristiques qui renvoient exclusivement des informations sur les tableaux arguments, comme le rang ou le nombre d'éléments par sous- dimension

- les fonctions de réduction qui permettent de manipuler une seule dimension des tableaux
- les fonctions de constructions qui construisent, à partir de différents types d'éléments comme des vecteurs, des matrices ou des scalaires, de nouveaux tableaux (dont la taille peut être quelconque)
- les fonctions de manipulation qui permettent de manipuler les tableaux sans en changer cependant la forme (tranposition ou rotation des éléments suivant une direction fixe par exemple)

6.3.1.4. Le C*

C* est une adaptation du C ANSI pour la Connection machine. C'est à dire qu'un programme C classique peut être compilé et exécuté sur la CM2 (s'il ne comporte aucune utilisation erronnée des mots clés propres à C*). C* comprend plusieurs concepts propres à la Connection Machine.
Une méthode spécifique permet de définir la taille et la forme des variables parallèles ainsi que de créer ces variables parallèles. De nouveaux opérateurs destinés aux variales parallèles ont été ajoutés et les opérateurs classiques ont été adaptés à ce type de variables. Des méthodes de sélection des variables parallèles, ainsi que de spécifications d'éléments particuliers de ces variables ont été créées. Les pointeurs et les fonctions ont été adaptés aux données parallèles. Une bibliothèque de fonctions permettant de faire communiquer les variables parallèles est également disponible.

Un des principaux concepts de C* réside donc dans la notion de variables parallèles. La première étape lorsque l'on désire utiliser des variables parallèles consiste à déclarer leur *forme* (*shape*) qui est en fait celle d'une n-grille dont l'utilisateur définit toutes les tailles. En C* le concept de *forme* est étroitement lié avec celui plus hardware de processeur virtuel : on raisonne sur les *formes* exactement comme on avait l'habitude de le faire sur les processeurs virtuels (par exemple on peut sélectionner un sous ensemble de la *forme* ou même un seul élément). On définit ensuite autant de variables parallèles qu'on le désire, en les attachant aux différentes formes. Par exemple si l'on définit deux *formes*, l'une composée d'une grille 2x3 et l'autre d'une grille 3x3x3, et que l'on associe une variable parallèle de type *int* à la première *forme*, cette variable est en fait composée de 6 entiers disposés dans chacune des positions de cette *forme*.
Notons que les variables scalaires sont en C* directement stockées sur l'hôte.

On peut alors réaliser sur ces variables parallèles l'équivalent de la plupart des opérateurs scalaires usuels comme l'addition ou l'affectation. Il existe aussi plusieurs opérateurs propres aux variables parallèles qui comprennent intrinséquement des communications, par exemple la réduction (tel *valeur+=a* où *valeur* est une variable scalaire et *a* une variable parallèle, et l'on réalise dans *valeur* la somme de tous les éléments de *a*) qui intégre une communication de type *scan*.

6.3.1.5. ParIS

ParIS (Parallel Instruction Set) est un langage de niveau intermédiaire entre des langages de haut niveau comme *Lisp,CM-Fortran, C* et l'assembleur de la CM2, CMIS. Ce langage ne permet de décrire que les instructions parallèles du code, la partie séquentielle (qui va être exécutée sur l'ordinateur hôte) étant écrite en Lisp ou en C par exemple.
La syntaxe des instructions ParIS diffère suivant le langage support utilisé, mais dans tous les cas, le contrôle de type, le découpage des instructions mathématiques sont à la charge de l'utilisateur. Par contre, la gestion des mémoires locales ou des processeurs virtuels sont transparents à l'utilisateur. ParIS n'est donc ni un langage de haut niveau, ni un assembleur évolué, il s'agit plutôt d'une catégorie de langage propre aux machines SIMD type Connection Machine.

Un programme ParIS doit gérer explicitement le concept de processeur virtuel : création de la topologie des processeurs virtuels, allocation des processeurs virtuels sur les processeurs physiques et calcul du *VP-ratio*. Il est alors possible à l'utilisateur de changer de géométrie ou de *VP-ratio* en redéfinissant les paramètres précédents.
Plusieurs fonctions ParIS sont dévolues à la gestion de la mémoire locale des processeurs de la CM2. En effet l'utilisateur doit assurer lui-même la gestion mémoire des processeurs virtuels : il faut allouer des champs (c'est à dire une suite de bits consécutifs) de la mémoire et les affecter aux processeurs virtuels. Toute opération ParIS fera ensuite référence à ces champs qui contiennent les variables parallèles des processeurs virtuels. On peut ainsi allouer (ou récupérer) de la mémoire sur le tas ou la pile suivant la durée de vie de ces variables parallèles.

Les opérateurs ParIS comprennent l'ensemble des opérations arithmétiques classiques, mais aussi les communications (aussi bien entre processeurs de la CM2 qu'avec l'ordinateur hôte). Ces opérateurs portent sur des champs (dont la création a été décrite au paragraphe précédent), et il est possible d'utiliser des champs de type identique, mais de taille différente. Notons que le champ résultat n'existe pas dans le cas de communications avec l'ordinateur hôte puisque les résultats sont stockés dans des variables séquentielles de l'hôte.

La gestion de contexte est elle aussi explicite et il est nécessaire de faire appel à des fonctions spécialisées pour sauver (ou restaurer) des contextes dans la pile.
La gestion des processeurs actifs est aussi assurée par l'utilisateur qui doit pour ce faire tout d'abord mettre à jour un bit de test (en fonction de la condition retenue) et mettre ensuite à jour le *context-flag* en fonction de ce bit de test. Comme dans les autres langages, il existe des instructions qui s'appliquent d'autorité à tous les processeurs, actifs ou non.

6.3.1.6. CMIS

CMIS, ou Connection Machine Instruction Set, est l'assembleur de la CM2. Etant donné la nature SIMD de la Connection Machine, il existe une différence particulière entre CMIS et les langages précédents : le code CMIS est entièrement exécuté par le séquenceur de la CM2 (au lieu d'être en partie exécuté sur l'ordinateur hôte).
Un programme CMIS doit donc piloter les processeurs élémentaires de la CM2, éventuellement les FPU, assurer les communications entre processeurs élémentaires de la CM2, mais aussi entre les processeurs élémentaires et l'ordinateur hôte. Il doit donc être synchronisé avec le programme s'exécutant sur l'hôte.

La gestion des mémoires locales peut aussi être différente de celle rencontrée jusqu'à présent, elle est alors directement liée à l'architecture des modules de la CM2 : un mot de 32 bits appartient à la mémoire associée à un module (sur lequel se trouvent les 32 processeurs élémentaires associés à cette mémoire locale), et non plus à un processeur (élémentaire ou virtuel) comme auparavant.
Par exemple, dans le cas d'une instruction destinée aux processeurs élémentaires, le mot de 32 bits est réparti à raison d'un bit par processeur (accès *slice-wise*).

Les communications se font en CMIS en accédant directement aux liens physiques de l'hypercube qui est constitué par les modules de 32 processeurs élémentaires. Ainsi les communications s'effectuent par échanges de mots de 32 bits entre modules via les liens correspondants de l'hypercube. Notons pour terminer qu'un des grands avantages de CMIS semble être la possibilité d'écrire des routines de communications très performantes, voire même dans certains cas de pouvoir utiliser l'hypercube de manière optimale [DTW2].

6.3.2. Exemple de programmation : le produit matrice-vecteur

La spécificité de programmation de la Connection Machine, son caractère moins "intuitif" que des machines MIMD programmables en C parallèle, font que l'exemple dont cette section fait l'objet va être plus détaillé que ne l'ont été ceux du FPS T40 par exemple.
Cet exemple est programmé en *Lisp, les raisons de ce choix sont expliqués en 7.1.

6.3.2.1. Remarques préliminaires

Le choix d'un langage de haut niveau permet de s'abstraire au maximum des contraintes matérielles de la CM2, tout en conservant la spécificité de programmation de la CM2.
Le *Lisp a été le langage choisi pour développer les expérimentations sur les résolutions de systèmes successifs. C'est pourquoi il a paru naturel que cet exemple soit aussi

programmé en *Lisp.

Le choix du produit matrice-vecteur n'est pas innocent : outre le fait que cette opération algébrique s'accorde naturellement avec les concepts de programmation de la CM2, c'est aussi un des éléments clés de la méthode à base de gradients conjugués successifs qui va être expérimentée (voir le chapitre 7) sur cette machine.

Cette multiplication peut être réalisée de plusieurs façons (ces diverses méthodes ont été examinées dans le cas des machines MIMD en 4.2) : formes produit scalaire, saxpy, blocs ... Parmi toutes ces méthodes, choisissons la méthode "naturelle" à base de produits scalaires (schématisée sur la figure ci-après). Elle est rappelée ci-dessous en séquentiel (on effectue x = A .b, l'algorithme est décrit en pseudo-code) :

```
pour i variant de 1 à n
   x[i]=0
   pour j variant de 1 à n
      x[i]=x[i]+A[i,j]*b[j]
```

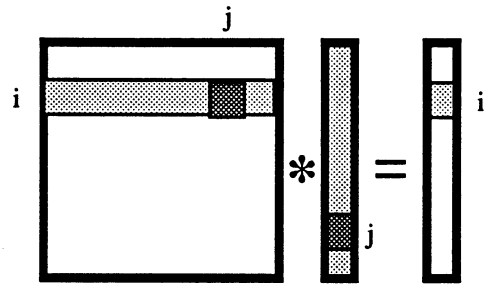


Figure 86: version dot du produit matrice vecteur

Dans toute la suite, le produit matrice-vecteur sera illustré dans le cas d'un problème de taille 4.

6.3.2.2. Description de l'implantation

Les données (à savoir la matrice A, les vecteur b et x) sont répartis sur les processeurs, configurés en grille 2-D, à raison d'un élément par processeur virtuel. Les vecteurs sont alors rangés par ligne, à raison d'un élement par colonne, et sont simultanément présents dans toutes les lignes. Le vecteur résultat est, lui, rangé en colonne. La figure ci-dessous illustre cette répartition des données.

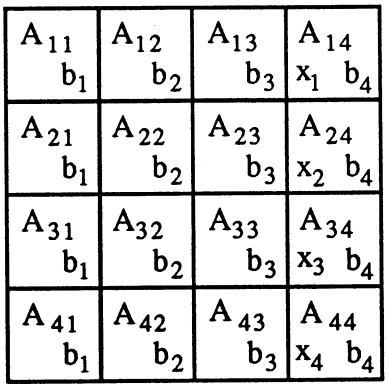


Figure 87 : répartition des données sur une 2-grille

L'intérêt d'une telle répartition des données est qu'elle permet en une seule instruction de réaliser le produit matrice-vecteur.
En effet l'instruction *spread* (une des fonctions de communication avec recombinaison décrites en 6.2.4) permet d'effectuer un produit scalaire. Comme la Connection Machine est un ordinateur SIMD, cette instruction fait exécuter à tous les proceseurs virtuels actifs un produit scalaire, et l'on obtient par conséquent un produit matrice-vecteur forme *dot*.

6.3.2.3. Le programme *Lisp

Le programme *Lisp interprété correspondant à un produit matrice-vecteur ($x = A.b$) forme dot est donné ci-dessous dans son intégralité pour une grille 2048x2048 (pour une utilisation réelle sur la CM2, le nombre de processeurs virtuels doit être au moins égal au nombre de processeurs physiques alloués). Les chiffres entre crochets renvoient aux commentaires ci-aprés.

```
(in-package '*lisp)                                              [1]

(defun init ()                                                   [2]
;;;; ------------------------------------------------------------
;;;;                initialisation de la matrice A
;;;; ------------------------------------------------------------
(*defvar x (!! 0.0))                                             [3]
(*defvar A (!! 0.0))
(*defvar b (!! 1.0))
(*defvar ligne (self-address-grid!! (!! 0)))                     [4]
(*defvar colonne (self-address-grid!!(!! 1)))                    [5]
(*defvar add (cube-from-grid-address!! colonne ligne))           [6]
(*if (=!! colonne ligne)                                         [7]
     (*set A (!! 10.0)))                                         [8]
 (*if (=!! colonne (+!! ligne (!! 1)))
```

```
      (*set A (-!! 2.0)))
  (*if (=!! colonne (-!! ligne (!! 1)))
      (*set A (-!! 2.0)))
  (*if (=!! colonne (+!! ligne (!! 2)))
      (*set A (-!! 1.0)))
  (*if (=!! colonne (-!! ligne (!! 2)))
      (*set A (-!! 1.0)))
)



(defun matvec ()
;;;; ------------------------------------------------------------------
;;;; calcul du produit matrice-vecteur x = A.b
;;;; ------------------------------------------------------------------
    (*set x (reduce-and-spread!! (*!! A b) '+!! 1))        [9]
    (*pset :overwrite x x add)                             [10]
)

(setq N 2048)                                              [11]
(*cold-boot :initial-dimensions (list N N))                [12]
(init)                                                     [13]
(cm:time (matvec))                                         [14]
```

Commentons tout d'abord les initialisations nécessaires au programme. En [1], le chargement du "package" des instructions *Lisp est effectué, cette instruction est obligatoire si l'on veut que la CM2 puisse interpréter les ordres *Lisp. [2] est l'instruction Lisp classique de définition d'une fonction. L'initialisation des variables parallèles (ou *pvar*, contenant un élément de la matrice ou des vecteurs par processeur virtuel, voir figure 87) *x*, *A* et *b* est réalisée en [3]. L'instruction [4] assure la création d'une *pvar* qui contient le numéro de ligne des processeurs virtuels (dans la 2-grille) ; l'instruction [5] fait de même pour les colonnes. En [6] est créée une *pvar* contenant l'adresse du processeur virtuel symétrique (pv_{ji}) du processeur virtuel courant (pv_{ij}) par rapport à leur position dans la 2-grille, cette variable est nécessaire à la communication [10] décrite ci-aprés. La séquence d'instructions [7] et [8] permet de sélectionner un sous-ensemble des processeurs virtuels et de réaliser sur ces processeurs actifs l'affectation d'une *pvar*.

Le produit matrice vecteur lui-même est exécuté en une seule instruction [9] qui effectue un produit scalaire simultanément sur toutes les lignes de processeurs. On affecte au vecteur *x* le résultat du *reduce-and-spread!!* qui effectue sur chaque processeur virtuel la multiplication des éléments de *A* par ceux de *b*, puis réalise la réduction par ligne en additionnant chaque produit (voir figure 87 l'exemple dans le cas d'une matrice 4x4). Le

résultat x est alors disponible en colonne puisque chaque élément x_i est le résultat du produit scalaire sur la ligne i. La fonction [15] permet l'affichage des résultats.

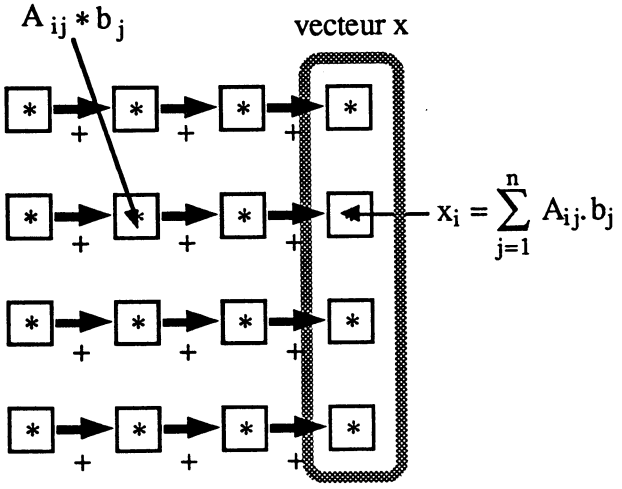


Figure 88 : réalisation du produit matrice-vecteur par un *reduce-and-spread!!*

L'instruction suivante ([10]) permet de transposer le résultat pour que x soit à nouveau stocké par ligne, cette communication n'est nécessaire que si l'on désire effectuer d'autres opérations sur x (comme l'utiliser à nouveau dans le cas d'une itération du gradient conjugué par exemple).

Les deux fonctions étant définies, il reste maintenant à expliciter le programme principal. Il faut tout d'abord préciser la taille de la grille [11]. L'instruction [12] permet d'initialiser la CM2 et de choisir la topologie, à savoir une grille à 2 dimensions (l'instruction *list* a en effet 2 arguments) de taille N par N. L'appel de la fonction *init* est réalisé par l'instruction [13]. Le temps d'exécution du produit matrice-vecteur (appel de la fonction *matvec*) est mesuré par la fonction *cm:time* (instruction [14]) qui fournit à la fois le temps d'occupation de l'hôte plus celui de la CM2, et le temps d'occupation de la CM2 uniquement , le tout en secondes.

Chapitre 7

Expérimentations sur la CM2

Où l'on présente l'implantation de l'algorithme du gradient conjugué préconditionné pour systèmes consécutifs sur la CM2, qui ne confirme pas les résultats théoriques que l'on espérait. Une réponse efficace (mais partielle) à la résolution de systèmes linéaires successifs est donnée dans le cas de modifications de rang 1 avec les expérimentations réalisées (en collaboration avec L. Colombet) sur la méthode de Woodbury.

7.1. Préambule

Ce qui nous intéresse ici n'est pas tant d'obtenir des codes très performants sur la CM2 : le chapitre 6 montre qu'il suffit alors de tout écrire en CMIS pour voir une spectaculaire augmentation des performances, même avec des algorithmes peu adaptés au parallélisme SIMD. Le but recherché est plutôt de montrer l'influence que peu avoir ce type de machine sur les algorithmes décrits au chapitre 2, et par là même plus généralement sur d'autres algorithmes de calculs.

Toutes les expérimentations menées sur la CM2 ont été effectuées en *Lisp. Ce choix a été guidé par plusieurs raisons.
Tout d'abord, afin de rester cohérent avec les expérimentations réalisées sur les autres machines (où le langage C a été utilisé). En effet, dans des programmes de calculs intensifs, l'utilisation de langages parallèles de bas niveau est plutôt réservée à de courtes portions de code permettant de diminuer fortement le temps d'exécution : par exemple, dans le cas de l'implantation de l'algorithme du gradient conjugué préconditionné pour systèmes linéaires successifs sur le FPS T40, la réécriture des calculs de la boucle de convergence du gradient conjugué avec des instructons vectorielles de bas niveau a permis de diminuer le temps d'exécution de plus de 40% par rapport à une version utilisant des fonctions vectorielles classiques écrites en langage C.
D'autre part, parmi tous les langages de haut niveau disponibles sur la CM2, le *Lisp a été choisi parce qu'il permet un gain de performances de l'ordre de 10% car il est (pour des raisons historiques) plus proche de la machine que ne le sont les autres langages de haut niveau.

7.2. L'algorithme du gradient conjugué

7.2.1. Implantation SIMD classique du gradient conjugué

La CM2 est une machine SIMD massivement parallèle à grain fin. Il faut donc que l'algorithme utilisé puisse permettre un parallélisme massif à chacune des étapes du calcul. C'est le cas du gradient conjugué qui peut se décomposer à base de produits scalaires (comprenant le produit matrice-vecteur) et de saxpy. Ces opérations peuvent toutes être réalisées sur la Connection Machine à base de communications avec recombinaison ou de communications de type *news*.
Les problèmes liés aux calculs dûs au préconditionnement seront abordés ultérieurement (en 7.4 et 7.5.1). L'implantation décrite ici a un préconditionnement diagonal. Ce type de préconditionnement n'est pas très utilisé dans la pratique, mais il permet de discuter de l'implantation du gradient conjugué préconditionné dans toute sa généralité.

La topologie utilisée est tout ce qu'il y a de plus classique sur la CM2 : les processeurs virtuels sont organisés suivant une grille à deux dimensions de la même taille que celle du système à résoudre. Les données sont donc stockées à raison d'un élément par processeur virtuel : l'élément i-j de la matrice *A* est stocké sur le processeur virtuel de coordonnées (i,j) dans la 2-grille, les vecteurs sont stockés par ligne (sur toutes les lignes de processeurs afin de diminuer les communications). La figure ci-dessous montre cette structure de données pour la matrice *A* et le vecteur gradient *g* (par exemple) dans le cas d'une grille 5x5.

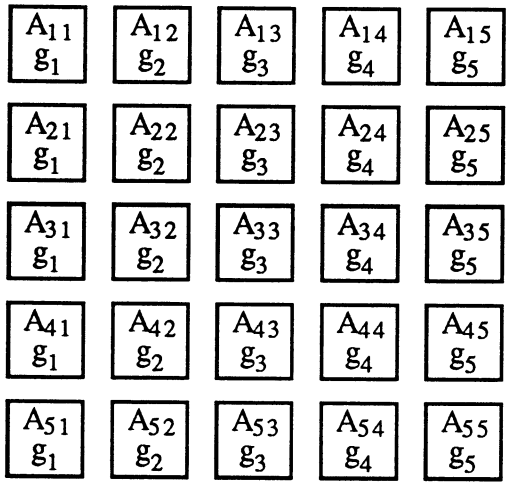


Figure 89 : stockage des données sur les processeurs virtuels pour le gradient conjugué

L'algorithme séquentiel implanté est rappelé ci-dessous en pseudo code (les références entre accolades désignent les noyaux d'exécution qui font l'objet d'explications particulières ci-aprés) :

```
/* initialisations */
pour i variant de 1 jusqu'à n
   g[i]=-b[i]                          {1}
   z[i]=g[i]/A[i,i]                    {2}
   d[i]=z[i]
/* itérations du gradient conjugué préconditionné */
répéter                                {3}
   /* produit matrice-vecteur */
   pour i=1 jusqu'à n
      v[i]=0
      pour j=1 jusqu'à n
         v[i]=v[i]+A[i,j]*d[j]         {4}
   /* mise à jour de x et g */
   olderror=0
   psgd=0
```

```
psvd=0
pour i=1 jusqu'à n
   olderror=olderror+g[i]*z[i]     {5}
   psgd=psgd+g[i]*d[i]
   psvd=psvd+g[i]*v[i]
rho=psgd/psvd                      {6}
pour i=1 jusqu'à n
   x[i]=x[i]-rho*d[i]              {7}
   g[i]=g[i]-rho*v[i]
/* résolution du système */
pour i=1 jusqu'à n
   z[i]=g[i]/A[i,i]                {8}
/* mise à jour de d */
newerror=0
pour i=1 jusqu'à n
   newerror=newerror+g[i]*z[i]
beta=newerror/olderror
pour i=1 jusqu'à n
   d[i-=z[i]-beta*d[i]
/* critère de fin d'itération */
ggcc=0
pour i=1 jusqu'à n
      ggcc=ggcc+g[i]*g[i]
ggcc=sqrt(ggcc)                    {9}
jusqu'à (ggcc<=precision)
```

Le gradient conjugué préconditionné a été implanté tel quel : les noyaux d'exécution (produits scalaires et saxpy) sont effectués dans cet ordre, séquentiellement. Nous allons maintenant discuter de la parallélisation de ces noyaux, en les prenant dans leur ordre d'exécution dans l'algorithme.

Le premier noyau à s'exécuter en parallèle est l'affectation vectorielle {1} : chaque élément *g[i]* prend la valeur opposée de l'élément correspondant *b[i]*. Comme ces deux éléments sont stockés sur le même processeur virtuel, aucune communication n'a lieu.
L'opération {2} est une division vectorielle, et ici aussi chaque élément *z[i]* prend pour valeur le résultat du quotient de *g[i]* par *A[i,i]*. Afin de ne pas effectuer une communication générale inutile (afin de communiquer à chaque processeur la valeur correspondante de *A[i,i]*), un vecteur auxiliaire a été créé qui contient la diagonale de la matrice *A* : ainsi *aux[i]=A[i,i]*. De cette façon les calculs restent encore une fois locaux et ne mettent en jeu aucune communication.
La boucle de convergence du gradient conjugué (notée {3}) est effectivement implantée séquentiellement, pour plusieurs raisons. Tout d'abord, la structure de donnée adoptée ne permet pas facilement d'effectuer en parallèle les calculs de plusieurs occurences de la

boucle : il faudrait alors dupliquer les données, ce qui diminuerait fortement la taille maximale du problème pouvant être traité sur la CM2 (chaque processeur élémentaire de la CM2 ne dispose en effet que de 8 ko de mémoire, à répartir entre les différents processeurs virtuels qu'il doit émuler). D'autre part l'algorithme lui même ne se prête pas à la parallélisation de cette boucle : c'est ce caractère séquentiel (qui permet à l'étape *k* de disposer des résultats de l'étape *k-1*) qui assure la convergence du gradient conjugué.
L'étape {4} est constituée du produit matrice-vecteur, il est implanté avec n produits scalaires (les différentes formes du produit matrice-vecteur sont décrites en 4.2) qui sont calculés en parallèle. Ces produits scalaires sont implantés sous forme de communications avec recombinaison (un *reduce-and-spread* en fait, voir 6.2.4) comme dans l'exemple de programmation de la section 6.3.2. Toutes ces communications sont effectuées en parallèles, chaque processeur virtuel faisant les calculs sur ses propres données. Par conséquent le vecteur résultat est obtenu sur la dernière colonne de la 2-grille (dans les faits il s'agit d'une légère simplification qui ne change rien au principe). La figure ci-dessous illustre ce fait sur un produit matrice-vecteur $v=A.d$ de taille 5x5.

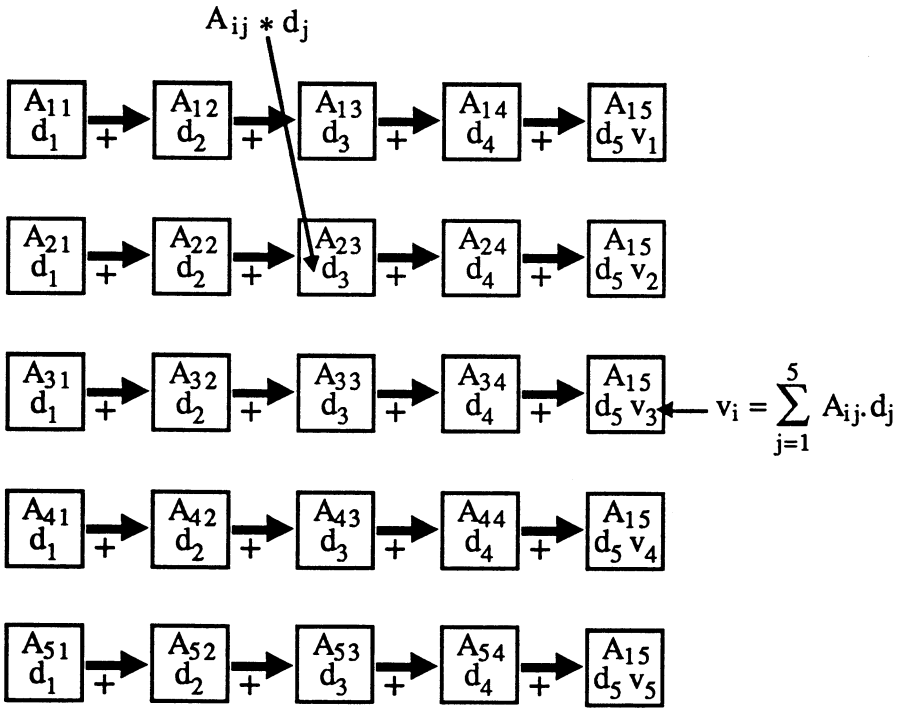


Figure 90 : déroulement du produit matrice-vecteur

Le résultat *v* du produit matrice-vecteur est ensuite réutilisé comme donnée dans un produit scalaire (pour le calcul de *psvd*). Le vecteur *v* doit par conséquent lui aussi être rangé par ligne. C'est pourquoi il faut utiliser une communication générale qui transpose ce vecteur.
En {5}, un produit scalaire est toujours effectué par une communication avec recombinaison, mais cette fois-ci, le résultat est le même scalaire sur tous les processeurs (chaque ligne de processeurs ayant les vecteurs opérandes).

Les opérations purement scalaires comme la division {6} sont tout de même effectuées sur tous les processeurs virtuels (ils ont tous les données) afin d'éviter les instructions supplémentaires et la perte de temps due à la sélection d'un sous-ensemble de processeurs actifs.
Le second type de noyau d'exécution à la base du gradient conjugué est le saxpy (dont le premier est effectué en {7}). La structure de données adoptée permet d'effectuer ce type d'opération sans aucune communication : chaque processeur effectue la multiplication des deux éléments des deux vecteurs opérandes, ainsi que l'accumulation dans l'élément correspondant du vecteur résultat.
La résolution du système linéaire due au préconditionnement du gradient conjugué est ici réduite à une division vectorielle {8}, le préconditionnement étant ici choisi diagonal.
Le critère de convergence comporte tout d'abord un produit scalaire (effectué de façon classique à l'aide d'une fonction de communication avec recombinaison), suivi de la racine carrée du résultat précédent. Ce calcul numérique (qui n'est pas directement accessible sur les FPU associés aux processeurs élémentaires de la CM2) est effectué sur la machine hôte via une variable scalaire.

7.2.2. Le gradient conjugué : résultats expérimentaux

Commençons par discuter des méthodes de mesure du temps d'exécution sur la Connection Machine.
Dans tous les langages de haut niveau, les prises de temps se font via l'équivalent *Lisp de la fonction *cmtime*. Cette fonction prend pour paramètre la fonction dont on veut mesurer le temps d'exécution, et elle renvoie trois paramètres :

- le temps d'exécution sur la CM2
- le temps d'exécution sur l'hôte (qui comprend le temps d'exécution sur la CM2)
- le pourcentage d'utilisation de la CM2 (rapport des deux temps précédents)

Rappelons que les instructions de la Connection Machine sont fournies par l'hôte en "temps réel". Par conséquent pour avoir une exécution efficace du programme, il faut que le flot d'instructions en provenance de l'ordinateur hôte soit continu.
Malheureusement les ordinateurs hôte de la CM2 sont la plupart du temps des machines multi-utilisateurs et donc à temps partagé. Par conséquent, lorsque l'hôte traite les processus des autres utilisateurs, il se peut que le flot d'instructions de la CM2 soit interrompu. Cela n'est en effet pas obligatoire car il se peut très bien (à cause d'une machine très performante, d'un petit nombre d'utilisateurs ou d'un temps d'exécution des instructions de la CM2 élevé) que la Connection Machine n'ait pas terminée l'exécution des instructions qui lui ont été envoyées lorsque l'hôte recommence à traiter son processus.
Il y a donc là une source d'erreur sur les prises de temps non négligeable. Par exemple, en prenant des circonstances particulièrement défavorables, le temps d'exécution d'un

même programme peut varier du simple au double suivant que l'hôte est un MicroVax ou une grosse configuration d'un Sun 4 !
Cependant, il est possible, de remédier au moins partiellement à ce phénomène. Tout d'abord en s'assurant que le *VP ratio* de la CM2 est élevé : chaque processeur physique doit alors gérer plusieurs processeurs virtuels et donc exécuter plusieurs fois chaque instruction.
Dans ces conditions, il est raisonnable de tabler sur une variation maximum des temps d'exécution de l'ordre de 30%. C'est dans cette hypothèse que sont menées les expérimentations ; les temps donnés sont les temps moyens obtenus sur 3 à 5 exécutions successives sur des hôtes moyennement chargés.

Les temps d'exécution obtenus sur la CM2 avec 8192 processeurs pour l'implantation de l'algorithme du gradient conjugué préconditionné décrite ci-dessus sont les suivants (pour différentes tailles de systèmes) :

Taille	Temps	% CM2
128x128	0.325 s	17.5
256x256	1.077 s	46.3
512x512	3.55 s	72.7
1024x1024	13.16 s	83.7

Notons que les processeurs élémentaires ne possèdent pas assez de mémoire pour résoudre des systèmes de taille supérieure à 1024x1024.

On obtient donc les performances suivantes (sachant que le gradient conjugué nécessite $0(\alpha n^2)$ opérations arithmétiques, si α est le nombre d'itérations) :

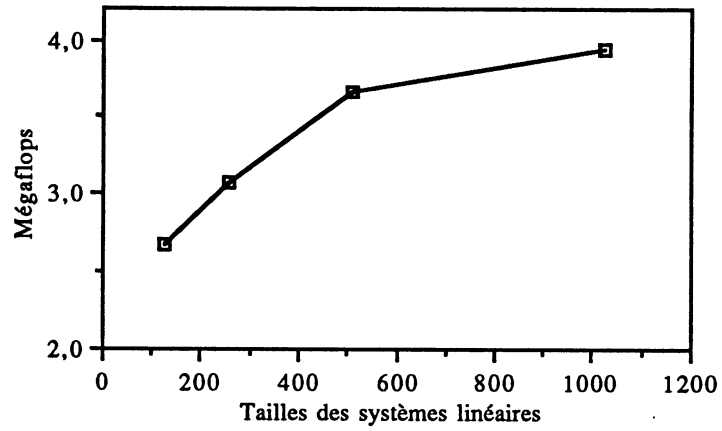


Figure 91 : performances du gradient conjugué préconditionné sur la CM2

L'amélioration des performances constatée peut s'expliquer par l'augmentation du *VP ratio* qui passe de 2 pour des systèmes 128x128 à 128 pour les systèmes 1024x1024 ; ce

qui permet un meilleur taux d'utilisation de la CM2 (les mêmes instructions étant de ce fait exécutées plusieurs fois).

7.3. La factorisation de Cholesky

La factorisation de Cholesky a été implantée sur la Connection Machine à la fois pour avoir une référence (c'est une méthode de résolution classique et qui s'adapte assez facilement au parallélisme SIMD) et parce qu'il s'agit du préconditionnement particulier de l'algorithme du gradient conjugué préconditionné pour systèmes linéaires consécutifs.

7.3.1. Implantation de la factorisation de Cholesky

La version ligne de l'algorithme de factorisation de Cholesky utilisé pour implantation sur la Connection Machine est donnée ci-dessous en pseudo-code (les nombres entre accolades sont les repères pour la description de la parallélisation de l'algorithme) :

```
pour k=0 jusqu'à n-1
   pour i=0 jusqu'à k-1
      aux[i]=d[i]*L[k,i]              {1}
   d[k]=A[k,k]
   pour i=0 jusqu'à k-1
      d[k]=d[k]-L[k,i]*aux[i]         {2}
   pour i=k+1 jusqu'à n-1
      L[i,k]=A[i,k]
      pour j=0 jusqu'à k-1
         L[i,k]=L[i,k]-L[i,j]*aux[j] {3}
      L[i,k]=L[i,k]/d[k]              {4}
```

Ici encore la topologie choisie est une 2-grille. La structure de données adoptée pour implanter la décomposition de Cholesky est celle classique qui consiste à placer sur chacun des processeurs virtuels un élément de chaque matrice *A* et *L*. Le vecteur *d* et les vecteurs auxillaires *inter* et *aux* (utilisés lors de la décomposition) sont répartis par ligne : par exemple le processeur virtuel de coordonnée (i,j) dans la 2-grille contient le $j^{ème}$ élément du vecteur *d*. De plus, afin d'économiser des communications, les vecteurs sont dupliqués sur toutes les lignes. La figure ci-dessous illustre le placement choisi sur une grille 5x5.

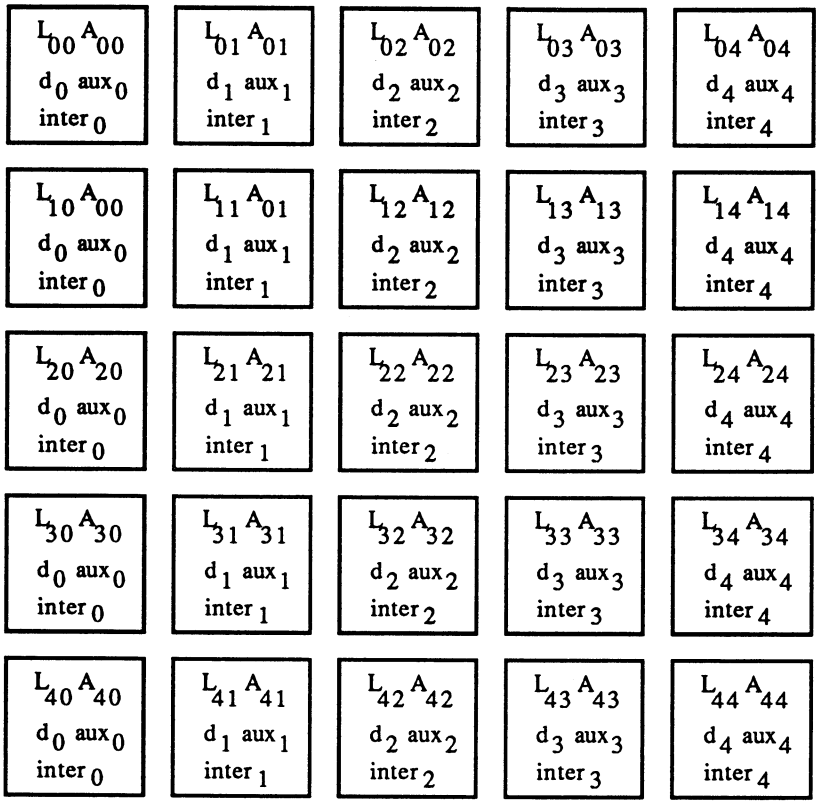


Figure 92 : placement des données pour la décomposition de Cholesky

Remarquons tout d'abord que si l'on peut inverser les boucles de la factorisation de Cholesky (pour en obtenir les versions lignes et colonnes), l'algorithme est néanmoins très séquentiel : le calcul du vecteur *d* doit précéder ceux de la mise à jour de la matrice *L*. Afin de décomposer l'algorithme en noyaux de calcul de granularité n, on peut le formuler ainsi (les chiffres entre accolades désignent les calculs correspondants dans l'algorithme précédent) :

```
pour k=0 jusqu'à n-1
   calcul du vecteur aux  ({1})
   calcul de d[k]         ({2})
   pour i=k+1 jusqu'à n-1
      calcul de L[i,k]    ({3},{4})
```

La parallélisation de la factorisation de Cholesky met en jeu (par rapport à celle du gradient conjugué) un nouvel aspect de la programmation de la CM2 : la gestion de masques sur les processeurs virtuels. En effet tout au long de l'algorithme de Cholesky les calculs sont effectués sur des portions de la matrice ou des vecteurs (alors que le gradient conjugué met en œuvre des calculs "complets" : produits matrice-vecteur, produits scalaires ou saxpy).
Cette contrainte ne peut que diminuer les performances de la factorisation de Cholesky sur la CM2 : seule une partie des processeurs va effectuer des calculs, or il apparaît clairement

(la puissance de calcul d'un processeur élémentaire étant très faible) que sur la CM2 on ne peut obtenir des performances très élevées que si tous les processeurs travaillent.

Détaillons maintenant la parallélisation des différents noyaux de calculs de la factorisation de Cholesky (qui correspondent aux repères entre accolades dans l'algorithme séquentiel ci-dessus).
La boucle sur k est obligatoirement conservée, et cette itération reste donc séquentielle.
Dans un premier temps (ce qui correspond à {1}), le calcul de *aux* est effectué sur tous les processeurs virtuels dont l'ordonnée est égale à k et l'abscisse est comprise entre 0 et $k-1$: on calcule localement (donc sans communication) sur chaque processeur virtuel i $aux[i]=d[i]*L[k,i]$.
Le calcul de $d[k]$ (correspondant à l'étape {2}) se fait en deux étapes. Tout d'abord on effectue sur les processeurs virtuels i de la ligne k (i variant comme précédément de 0 à $k-1$) l'accumulation $inter[i] = \sum_{j=0}^{i} L[k,i].aux[i]$. Ce calcul se fait par une opération de communication avec recombinaison (un *scan* sur la ligne k en fait). Ensuite on propage les résultats obtenus dans toutes les colonnes (avec une diffusion suivant les ordonnées) et l'on calcule $d[k]=d[k]-Inter[i]$. De cette façon le vecteur d est bien mis à jour simultanément sur toutes les lignes de la grille.
La mise à jour de la matrice L (opérations {3} et {4}) n'intervient que pour les processeurs virtuels dont l'abscisse est compris entre 0 et $k-1$ et l'ordonnée entre $k+1$ et $n-1$. Ici encore le calcul est effectué en deux étapes : on accumule tout d'abord sur chaque processeur virtuel i actif dans $inter[i]$ la somme $\sum_{j=0}^{k-1} L[i,j].aux[j]$, toujours au moyen d'une communication avec recombinaison (ici un *reduce-and-scan* qui diffuse le résultat sur la ligne opérande). Enfin, le calcul de $L[i,k]$ est effectué sur place : $L[i,k]=(A[i,k]-Inter[i])/d[k]$, sans communications.

7.3.2. Factorisation de Cholesky : résultats expérimentaux

Les temps d'exécution obtenues sur la CM2 avec 8192 processeurs pour l'implantation de la factorisation de Cholesky décrite ci-dessus sont les suivants (pour différentes tailles de systèmes) :

Taille	Temps
128x128	1.42 s
256x256	5.94 s
512x512	28.77 s
1024x1024	174.37 s

Notons que (comme précédément) les processeurs élémentaires ne possèdent pas assez de

mémoire pour résoudre des systèmes de taille supérieure à 1024x1024.

On obtient donc les performances suivantes (sachant que le gradient conjugué nécessite $0(\frac{n^3}{3})$ opérations arithmétiques) :

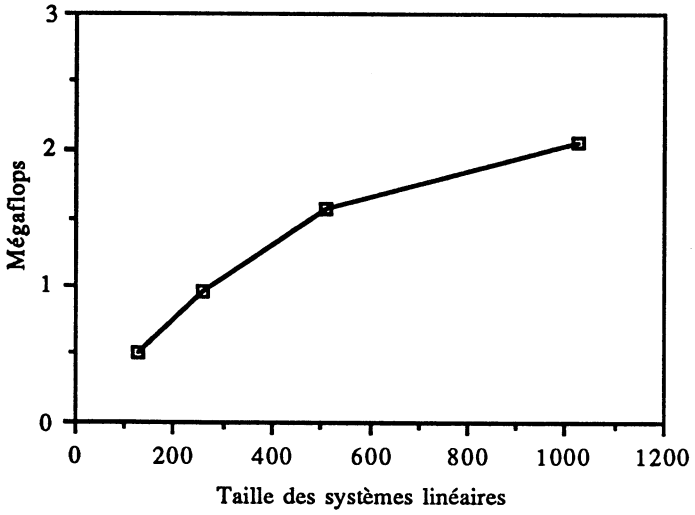


Figure 93 : performances de la factorisation de Cholesky

L'amélioration des performances s'explique ici aussi par l'augmentation du *VP ratio*. L'augmentation est cependant plus sensible que dans le cas du gradient conjugué : pour les petites tailles de matrice, on est davantage pénalisé par les masquages des processeurs virtuels : lorsque la taille du système est élevée, le volume de processeurs virtuels actifs augmente plus vite.
Remarquons néanmoins que les performances obtenues sont plus faibles que dans le cas du gradient conjugué (cela est bien entendu le fait de la mauvaise efficacité due au masquage des processeurs virtuels).
On a donc comme prévu intérêt à privilégier sur la CM2 des algorithmes effectuant des calculs très réguliers.

7.4. Résolutions de systèmes triangulaires

Ces méthodes de résolution de systèmes linéaires dont les matrices sont triangulaires (inférieures ou supérieures) sont utilisées pour résoudre les systèmes issus de la factorisation de Cholesky, soit pour une résolution directement par la méthode de Cholesky, soit pour la résolution du système issu du préconditionnement du gradient conjugué.

7.4.1 Une première implantation

Rappelons brièvement l'algorithme séquentiel de résolution d'un système triangulaire supérieur (qui est donné ci-dessous en pseudo-code pour un système $Ax=b$) :

```
pour i=n jusqu'à 1
   xi=bi
   pour j=n jusqu'à i+1
      xi=xi-Aij xj
   xi=xi/Aii
```

La Connection Machine est configurée suivant une grille à deux dimensions de la taille du problème à résoudre, les données sont réparties géométriquement sur la grille à raison d'un élément par processeur virtuel (il s'agit bien entendu du même placement de données que pour les expérimentations précédentes).
Une première version de cet algorithme peut être implanté sur la CM2 : l'idée clé consiste à propager à l'étape k, le long de la colonne k, l'élément x_k qui vient juste d'être calculé. La mise à jour de cette colonne est ensuite effectuée, puis les valeurs obtenues sont ensuite communiquées à la colonne $k-1$, ce qui permet ainsi d'initialiser l'étape suivante (qui va permettre le calcul de x_{k-1}). On réitère donc n fois ce processus pour obtenir l'ensemble $x_1, \ldots, x_n$ des solutions.
La figure ci-dessous illustre l'étape k de cet algorithme.

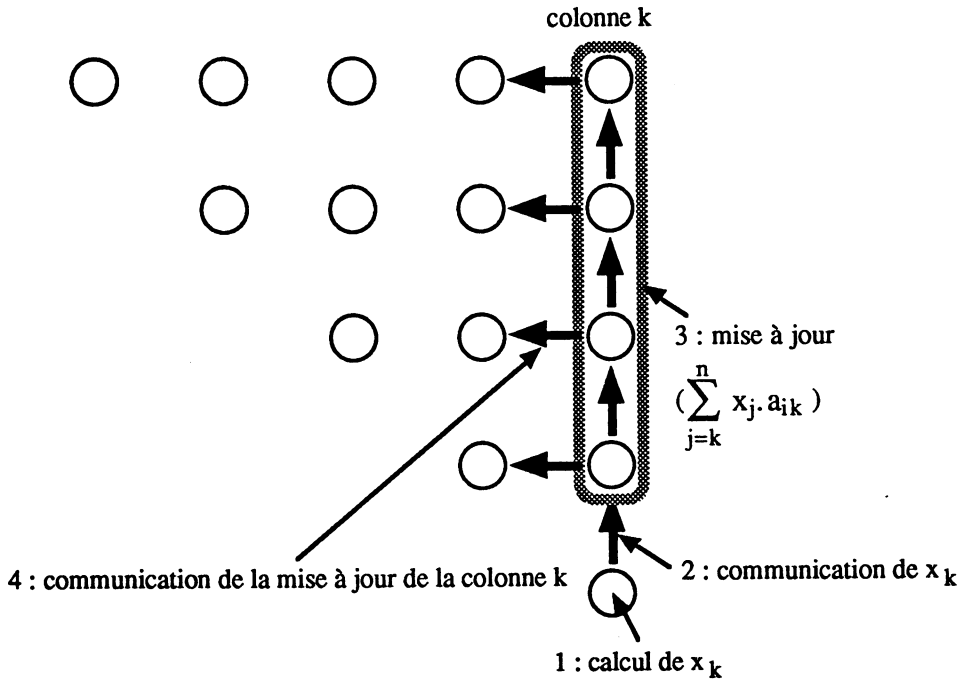


Figure 94 : une première implantation de la résolution triangulaire supérieure

Les deux phases de communications sont respectivement implantées sous forme d'un *spread* (qui permet de communiquer d'un processeur avec tous les autres processeurs de la même colonne) et un *news* qui fait communiquer chaque processeur avec un de ses voisins.

On obtient alors les performances suivantes :

Taille du système	Temps d'exécution (secondes)
8	0,049
16	0,105
32	0,218
64	0,442
128	0,889
256	5,661
512	42,376
1024	324,780

La résolution d'un système triangulaire coutant n^2 opérations arithmétiques, on obtient les performances suivantes.

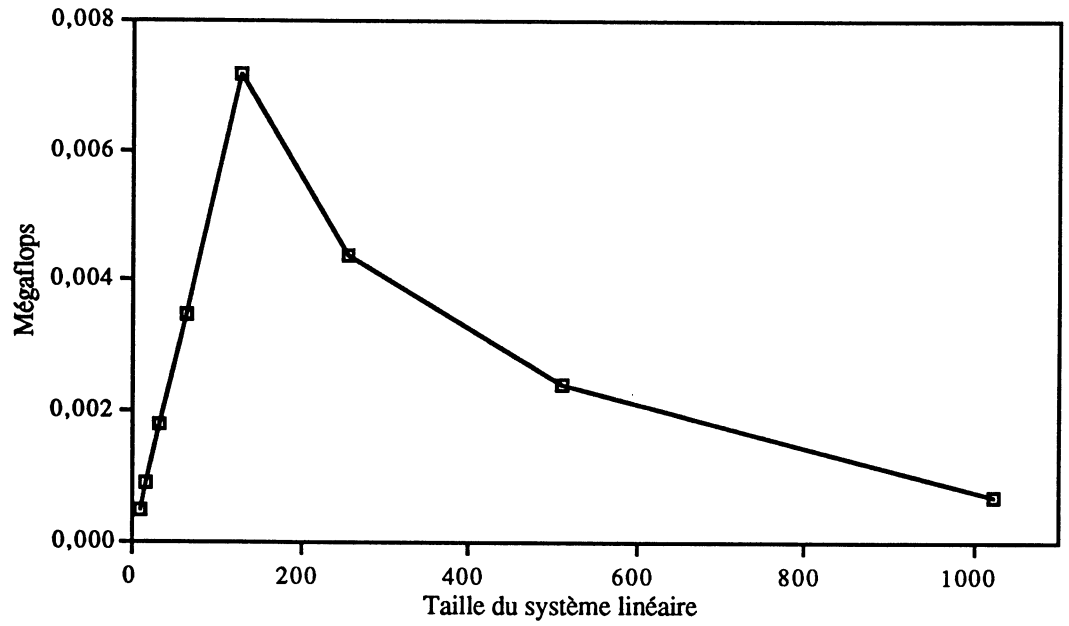


Figure 95 : résolution d'un système triangulaire (implantation classique)

Il apparaît donc nettement que cet algorithme ne convient pas du tout à une machine SIMD massivement parallèle comme la Connection Machine : une résolution par le gradient conjugué étant au moins vingt fois plus rapide (13 secondes contre plus de 300).
Ces piètres performances sont en partie dues au fait que seule une partie des processeurs travaille. En effet, la matrice étant distribuée géométriquement à raison d'un élément par

processeur virtuel, il apparaît déjà que la moitié inférieure des processeurs virtuels restera inactive pendant toute la durée de l'algorithme. De plus, parmi les processeurs qui vont avoir une tâche à exécuter, à un instant donné seule une colonne de processeurs virtuels est active. Et encore, il ne s'agit pas du moment le plus défavorable : lors du calcul d'un nouvel x_i, seul le processeur virtuel concerné effectue le calcul.
La chute des performances après un seuil donné peut être imputée au mécanisme de gestion des processeurs virtuels. Le programme a été exécuté sur une CM2 dotée de 8192 processeurs physiques, donc jusqu'à des problèmes de taille 64x64 la CM2 travaillait avec un seul processeur virtuel par processeur physique. Ensuite, pour des problèmes de taille supérieure, chaque processeur physique gérait séquentiellement plusieurs processeurs virtuels. Le maximum des performances correspond à un *VP ratio* de 2, il semble donc que pour ce programme on atteigne le meilleur compromis entre une charge de travail élevée des processeurs physique, et une perte de performances due au traitement séquentiel des processeurs virtuels sur les processeurs physiques.
Ce phénomène est particulièrement aïgu dans le cas de la résolution de systèmes linéaires triangulaires où un faible nombre de processeurs virtuels sont actifs simultanément : comme l'allocation des processeurs virtuels sur les processeurs physiques est transparente à l'utilisateur, il se peut que cette allocation soit particulièrement défavorable, à savoir par exemple qu'une ligne de la matrice soit regroupée sur un petit nombre de processeurs physiques.

Essayons maintenant d'évaluer le coût théorique de cet algorithme sur la Connection Machine. L'unité de mesure adoptéee sur la CM2 pour ces calculs de coût correspond au temps d'exécution d'une instruction élémentaire : par exemple une addition parallèle. On va de plus poser l'hypothèse (certes réductrice) que tous les éléments de la matrice sont stockés sur des processeurs physiques différents.
Avec ces hypothèses qui ne permettent guère d'évaluer finement le temps d'exécution théorique, il n'est possible que de donner un ordre de grandeur du coût de l'algorithme. La partie de l'algorithme la plus coûteuse est le *spread* (en effet les calculs et le *news* s'exécutent en parallèle en chacun une seule instruction) qui bien qu'utilisant la structure hypercube sous-jacente de la CM2 nécessite l'équivalent de $Logn$ instructions élémentaires pour s'exécuter. Comme la résolution triangulaire nécessite n itérations, on a donc un algorithme en $nLogn$.

7.4.2. Une version plus sophistiquée

L'analyse de la complexité parallèle ci-dessus a permis de trouver un algorithme en temps linéaire pour la résolution de systèmes triangulaires sur la CM2. Cette méthode s'inspire des algorithmes systoliques de résolution des systèmes triangulaires décrits dans [QuRo].
L'idée clé consiste à remplacer la communication de type *spread* qui coûte $Logn$ instructions par des communications locales de type *news* d'un coût théorique de 1 instruction. L'algorithme obtenu est sensiblement plus compliqué, mais son coût

théorique est linéaire puisqu'il s'exécute en $4n$ instructions.
Détaillons le principe de cet algorithme sur les deux premières itérations : on calcule x_n, x_{n-1} et x_{n-2}. La figure ci-dessous schématise ces étapes du calcul dans le cas de la résolution d'un système triagnulaire supérieur..

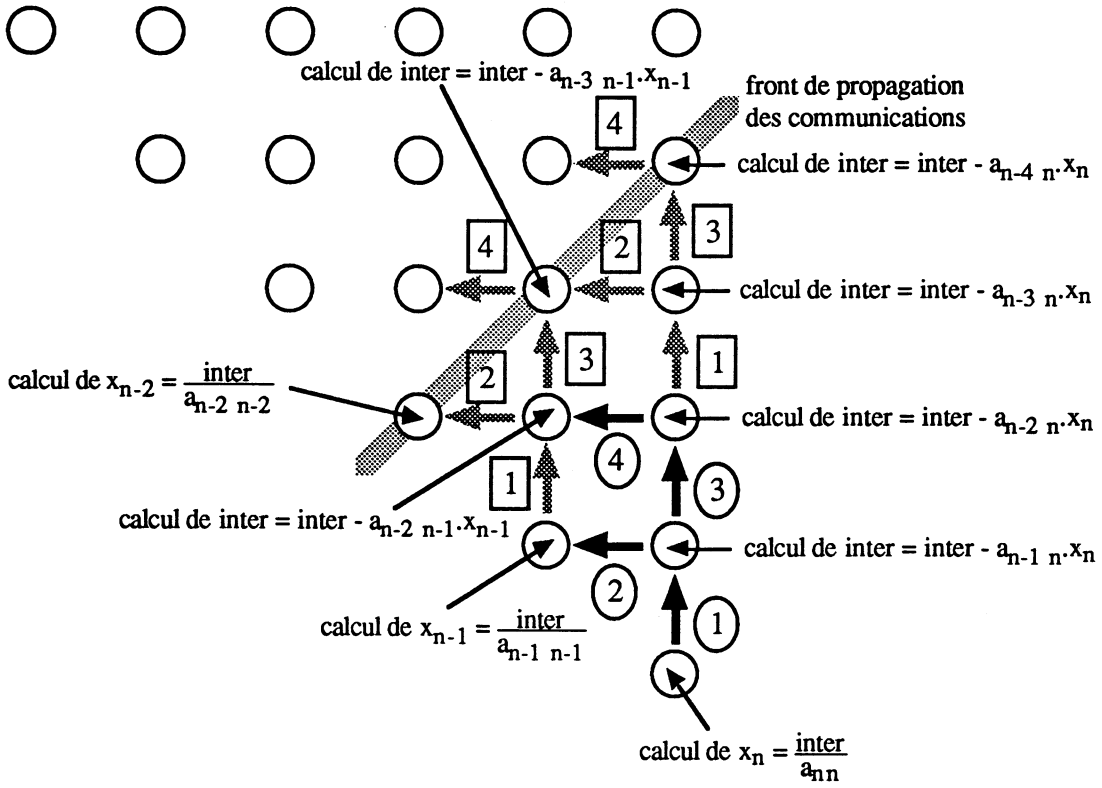


Figure 96 : implantation "systolique" de la résolution triangulaire supérieure

Les processeurs virtuels sont désignés ci-aprés par leurs coordonnées dans la 2-grille (ce qui est équivalent à l'indice de l'élément de la matrice A qu'ils stockent). Cet algorithme nécessite d'initialiser (avant de commencer les itérations) une variable *inter* stockée sur chacun des processeurs virtuels avec b_i.
Le calcul de x_n étant immédiat, la première itération de l'algorithme va permettre d'obtenir x_{n-1}. A l'étape 1, le processeur virtuel (n,n) calcule x_n(égal à b_n/a_{nn}) puis en communique la valeur au processeur virtuel (n-1,n), il s'agit de la communication notée {1} sur la figure ci-dessus. Ensuite lors de l'étape 2, le processeur virtuel (n-1,n) calcule (dans une variable *inter*) b_{n-1}-$a_{n-1n}x_n$, valeur qui est transmise au processeur virtuel (n-1,n-1), communication {2}. La troisième étape consiste à communiquer x_n du processeur (n-1,n) au processeur (n-2,n), communication {3}, qui effectue alors le calcul de b_{n-2}-$a_{n-2n}x_n$ et stocke à son tour le résultat dans la variable *inter*. La quatrième et dernière étape consiste à communiquer *inter* du processeur (n-2,n) au processeur (n-2,n-1), afin de préparer l'itération suivante de l'algorithme.
La deuxième itération de l'algorithme va permettre de calculer x_{n-1}, la troisième x_{n-2} et ainsi de suite jusqu'à obtention de x_0.

Calculons mainteant le coût de cet algorithme : chacune des 4 étapes s'exécutent en l'équivalent d'une instruction : il s'agit d'un calcul et d'une communication de type *news* qui prend (théoriquement) l'équivalent d'une instructions élémentaire pour s'exécuter. On effectue n itérations, mais la dernière itération ne nécessite que les deux premières étapes, le coût de cet algorithme est donc $4n-2$ instructions élémentaires.

Les temps d'exécution obtenus sont donnés ci-dessous :

Taille du système	Temps d'exécution (secondes)
8	0,128
16	0,275
32	0,560
64	1,156
128	2,263
256	14,770
512	110,146
1024	1485,561

Le coût d'une résolution d'un système linéaire triangulaire étant de n^2 opérations arithmétiques, on obtient les performances suivantes.

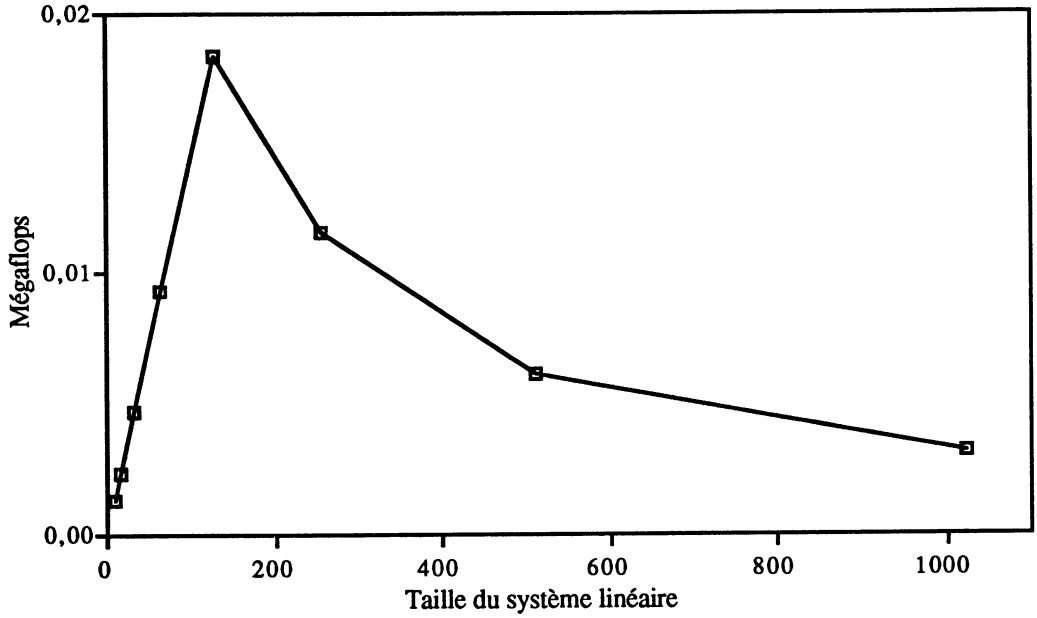


Figure 97 : résolution d'un système triangulaire supérieur (algorithme systolique)

Les performances sont encore plus mauvaises que pour l'algorithme précédent ! Et pourtant l'analyse de la complexité parallèle (voir paragraphe ci-dessus) montre que cet algorithme devrait permettre d'atteindre des performances nettement supérieures à celles du précédent : le temps est linéaire $4n-2$ au lieu de $nLogn$.

La différence importante entre les deux algorithmes réside bien sûr dans le type de communication employée : il s'agit ici de communications de type *news*. Ces communications sont implantées en utilisant le routeur : en général, il n'y a pas de conflit d'accès au niveau des liens du fait de la géométrie de ce type de communication, mais cela peut néanmoins se produire. Les performances sont alors beaucoup plus mauvaises que celles théoriquement attendues, et l'on aboutit a une très importante dégradation du temps d'exécution (d'autant plus forte que les conflits sont nombreux). De plus les communications de type *spread* sont très bien programmées et utilisent au mieux le routeur.

7.5. Résolution de systèmes successifs sur la CM2

7.5.1. Remarques préliminaires

Rappelons tout d'abord deux résultats expérimentaux que l'on vient d'obtenir sur la CM2 : résoudre un système 1024x1024 par le gradient conjugué avec un préconditionnement diagonal nécessite environ 13 secondes, tandis que la résolution d'un système triangulaire de même taille nécessite plus de 320 secondes ! Comme l'algorithme SPCG est basé sur des résolutions par le gradient conjugué préconditionné avec une décomposition de Cholesky (couteuse mais réalisée peu fréquemment), chaque itération du gradient conjugué nécessite donc la résolution de deux systèmes triangulaires.
Par conséquent, il apparaît au vu des expérimentations réalisées en 7.2 et 7.4 que cette méthode de résolution ne peut être implantée efficacement sur la CM2 dans les conditions que nous nous sommes fixés au début du chapitre 8. A savoir, programmer en *Lisp les algorithmes issus du chapitre 2 et n'utiliser sur la CM2 que des placements de données généraux susceptibles d'être adaptables à un cadre plus général.

On peut maintenant s'interroger sur les conditions à modifier pour réaliser une implantation efficace de cet algorithme sur la CM2. Les problèmes se posent tout d'abord au niveau des résolutions de systèmes triangulaires, et dans une moindre mesure pour les factorisations par la méthode de Cholesky. Nous allons dans un premier temps exposer le raisonnement théorique qui nous a fait choisir la Connection Machine pour réaliser ces expérimentations et ensuite décrire quelles peuvent être les solutions "techniques" à apporter au niveau de la programmation.

On veut tout d'abord estimer le temps d'exécution sur la Connection Machine de la résolution d'un système linéaire de taille nxn par la méthode du gradient conjugué (en supposant qu'il y ait convergence au bout de α itérations). Pour ce faire, décomposons cet algorithme (ici avec un préconditionnement diagonal) en noyaux d'exécution algébriques :

```
2 affectations vectorielles
1 division vectorielle
répéter jusqu'à convergence
   1 produit matrice-vecteur
   1 produit scalaire
   2 saxpy
   1 division scalaire
   2 saxpy
   1 division vectorielle
   1 produit scalaire
   1 division scalaire
   1 saxpy
   1 produit scalaire
```

Détaillons maintenant le temps d'exécution sur la Connection Machine de chacunes de ces primitives. Les affectations vectorielles et les divisions vectorielles prennent 1 instruction élémentaire : les éléments des matrices et des vecteurs étant placés à raison d'un par processeur virtuel, il suffit de faire exécuter soit une affectation, soit une division en parallèle sur tous les processeurs virtuels.
Le cas du produit matrice vecteur est plus délicat : si le *reduce-and-spread* prend $Log n$ instructions élémentaires (il utilise les liens de l'hypercube), par contre le temps d'exécution de la communication générale qui permet de transposer le vecteur résultat est beaucoup plus dur à estimer, bien que cette communication utilise elle aussi les liens de l'hypercube, elle fait appel au routeur. Compte tenu de la quasi absence de conflits lors cette communication particulière, on peut estimer son temps d'exécution à n instructions élémentaires. Le coût du produit matrice-vecteur peut donc être évalué à $n+Log n$ instructions élémentaires.
Les produits scalaires sont réalisés par des *reduce-and-spread* qui effectuent à la fois les multiplications sur place et la réduction du résultat par addition. Cette communication est implantée sur la CM2 de façon à utiliser les liens de l'hypercube, elle parcourt donc un vecteur de taille n en un temps $Log n$.
Les saxpy sont implantés sous forme d'une multiplication et d'une addition vectorielle (chaque processeur virtuel gère un élément de matrice ou de vecteur), le coût d'un saxpy est donc de 2 instructions élémentaires.
Les opérations scalaires prenent bien entendu 1 instruction élémentaire.
On aboutit finalement à un coût global de $\alpha(n+4Log n+13)+3$ instructions élémentaires pour l'algorithme du gradient conjugué avec préconditionnement diagonal.

Pour que l'implantation de l'algorithme du gradient conjugué préconditionné pour systèmes linéaires consécutifs puisse se justifier théoriquement sur la CM2, il faut que le préconditionnement particulier de la méthode (qui consiste à résoudre deux systèmes

linéaires triangulaires) apporte un surcoût qui soit au maximum en n. Rappelons donc le coût théorique des deux algorithmes de résolutions de systèmes linéaires triangulaires qui ont été implantés sur la CM2 (ils sont décrits en 7.4).
Le premier, qui fait appel a des communications le long des colonnes de la matrices, a un coût en $nLogn$ instructions élémentaires (cf 7.4.1). Cet algorithme n'est donc pas intéressant puisqu'il a un coût théorique supérieur à celui de l'implantation du gradient conjugué sur la CM2 qui est en αn.
Le second algorithme, inspiré des méthodes systoliques, a un coût en $4n$ instructions élémentaires (cf 7.4.2). Ainsi le surcoût causé par les résolutions triangulaires donne un algorithme en $4\alpha n$ au lieu de αn. Cependant il faut remarquer que l'algorithme du gradient conjugué préconditionné pour systèmes linéaires successifs permet de diminuer α (le nombre d'itrations du gradient conjugué) de façon sensible, et par conséquent on aboutit bien au gain de temps théorique espéré et qui a motivé ces expérimentations sur la CM2.

Le cas de la factorisation de Cholesky est moins critique. Parce que d'une part les performances obtenues sont meilleures (aussi bien le coût théorique que les résultats expérimentaux, voir 7.3) que celles des résolutions triangulaires, et d'autre part à cause de leur faible fréquence dans l'algorithme du gradient conjugué préconditionné pour systèmes linéaires successifs.

Examinons maintenant les modifications qu'il faudrait apporter pour obtenir une implantation très efficace de cet algorithme sur la CM2. Si l'on désire améliorer les performances du gradient conjugué préconditionné pour systèmes linéaires successifs sans modifier ni l'algorithme ni la répartition de données (ce qui est souhaitable), la seule solution consiste à abandonner *Lisp pour un langage de bas niveau qui permette de contrôler exactement les opérations que l'on effectue sur la Connection Machine. ParIS ne fournit qu'un accès incomplet à toutes les ressources de la CM2. Si l'on doit exploiter à fond les possibilités offertes : placement des processeurs virtuels sur les processeurs physiques, gestion optimale des mémoires locales et des unités de calculs flottants, communications optimisées pour cet algorithme, il faut alors programmer en CMIS. Remarquons que l'investissement nécessaire serait alors très important : un programme CMIS doit gérer explicitement tous les composants de la CM2 (cf 6.3.1.6) !
Il serait alors possible à l'utilisateur de placer explicitement les processeurs virtuels sur les processeurs physiques, ce qui permettrait d'avoir un *VP ratio* constant sur toute la machine et même de choisir les processeurs physiques pour les étapes délicates de résolution triangulaire.
Programmer en CMIS permet de stocker les données de façon répartie entre les 32 processeurs physiques d'un module de la CM2 et d'accélérer notablement les calculs flottants (le transposeur ne sert alors plus que de buffer pour les communications). Notons enfin qu'une version optimale du produit matrice-vecteur sur la CM2 a été écrite en CMIS [DTW2] : on obtient 1.5 Gflops sur une CM2 à 32768 processeurs.

Néanmoins, le secteur où les gains seraient les plus importants sont les communications : CMIS permet en effet d'utiliser explicitement les canaux de l'hypercube entre modules. Ainsi il serait par exemple possible d'implanter les *news* sous forme de *cube swap* (ou échange entre voisins, qui est la communication de bas niveau la plus efficace).
Le performances que l'on pourrait espérer atteindre serait alors de l'ordre de plusieurs centaines de Mflops à 1 Gflops suivant le nombre de processeurs élémentaires dont disposerait la CM2.

8.5.2. Résultats expérimentaux : la méthode de Woodbury

Nous proposons ici une solution, partielle certes, au problème posé par l'inefficacité sur la Connection Machine d'algorithmes présentant des calculs peu réguliers.

Il faut donc éviter toutes les méthodes de résolution nécessitant des résolutions triangulaires, ou même des calculs peu réguliers. Parmi les méthodes spécifiques étudiées au chapitre 2, la modification des facteurs de Cholesky et la mise à jour de QR entrent dans cette catégorie de méthodes peu appropriées à la CM2.
Tout d'abord, ces deux méthodes nécessitent la résolution de systèmes triangulaires. Ensuite, les calculs portent sur de petites portions de vecteurs ou de matrices et ne sont donc guère favorable à une implantation efficace sur la CM2. On pourra consulter à ce sujet [Colo] qui donne des résultats pour des implantations en *Lisp de ces méthodes : par exemple une implantation sophistiquée de l'algorithme de Bennett pour des modifications de rang 1 sur une matrice 512x512 donne seulement des performances de 0.076 Mflops.

Par contre, la méthode de Woodbury pour des transformations de rang 1 n'utilise que des calculs très réguliers, et l'on peut donc espérer obtenir des performances intéressantes sur la Connection Machine. Cependant toutes les perturbations ne peuvent être obtenues à partir de perturbations de rang 1, par conséquent la réponse apportée ici au problème de résolution de systèmes linéaires successifs n'est que partielle.
Remarquons que l'intérêt n'est plus aussi grand dans le cas de transformations de rang quelconque qui nécessitent le calcul de produits matrice-matrice de tailles différentes de celle de la matrice du système linéaire. Par conséquent il faudra réaliser le masquage sur la 2-grille des processeurs virtuels afin que seuls les processeurs contenant les éléments des matrices des perturbations effectuent les calculs. On retrouve ici le même problème que pour la factorisation de Cholesky : les performances deviennent inférieures à celles que l'on peut obtenir avec un gradient conjugué (qui ne préjuge d'ailleur pas de la façon dont les matrices sont modifiées).

La méthode de Woodbury pour calculer l'inverse d'une matrice perturbée par une transformation de rang 1 a été présentée au chapitre 2. Nous rappelons simplement ci-dessous la forme de l'algorithme séquentiel qui a été utilisé pour implanter cette méthode

sur la CM2. On veut donc calculer l'inverse de la matrice $A+czz^t$, où A est une matrice nxn dont on connaît déjà l'inverse A^{-1}, c un scalaire et z un vecteur de taille n. L'algorithme est donné ci-dessous en pseudo-code (avec $x = z^tA^{-1}$ et $y = A^{-1}z$), l'inverse de la matrice $A+czz^t$ étant stocké dans la matrice *Amoins1*.

```
pour i et j variant de 0 à n-1
   x[i]=x[i]+Amoins1[j,i]*z[j]
   y[i]=y[i]+Amoins1[i,j]*z[j]
pour i variant de 0 à n-1
   s=s+x[i]*z[i]
s=(1+s*c)/c
pour i variant de 0 à n-1
   y[i]=c*y[i]
   pour j variant de 0 à n-1
      Amoins1[i,j]=Amoins1[i,j]-s*x[j]*y[i]
```

Cet algorithme peut se décomposer en trois tâches principales : le calcul des deux vecteurs x et y, le calcul du scalaire s et le calcul final de l'inverse perturbée *Amoins1*. Ces trois tâches sont liées par des contraintes de précédence : on doit les exécuter séquentiellement dans cet ordre.
Par conséquent, l'implantation de cet algorithme sur la Connection Machine a été réalisée en parallélisant chacune de ces trois tâches qui sont ensuite exécutées séquentiellement.
La CM2 est classiquement configurée en 2-grille. La matrice A^{-1} est placée sur la grille à raison d'un élément par processeur virtuel : $A^{-1}[i,j]$ est placé sur le processeur virtuel de coordonnées (i,j) dans la grille. Le vecteur z est distribué suivant les lignes et les colonnes (dans 2 *pvar* différentes *z1* et *z2*) afin d'éviter des communications entre processeurs lors du calcul de x et y. La figure ci-dessous illustre ce placement sur une 5-grille.

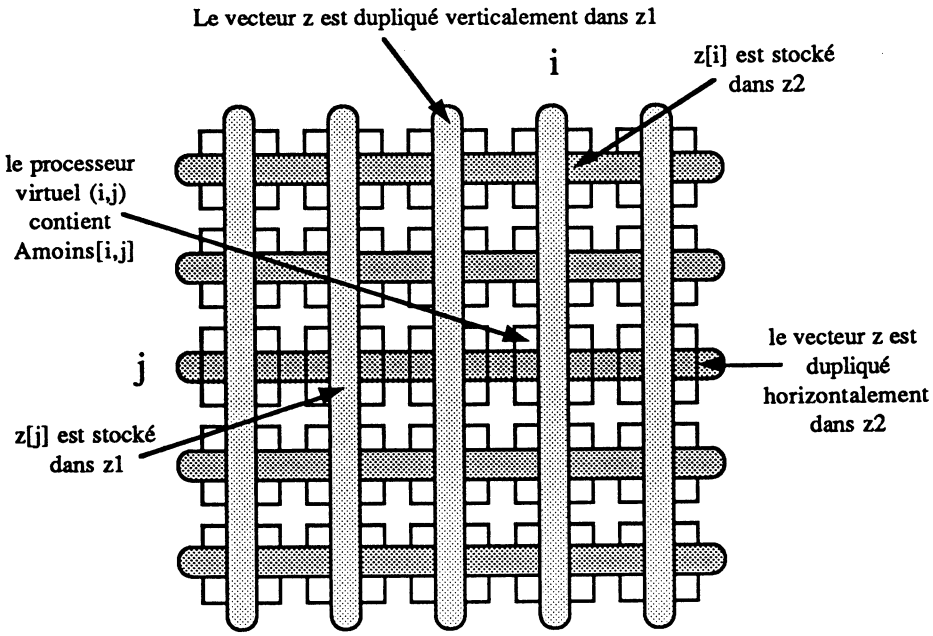


Figure 98 : placement des données pour l'algorithme de Woodbury (rang1)

Le calcul de *x[i]* se fait en effectuant le produit scalaire de *A[.,i]* (stocké dans la i^{ème} colonne de la grille) par *z* (aussi stocké sur cette colonne dans *z1*). On calcule donc *x* en effectuant en parallèle un *reduce-and-spread* (qui est une communication avec recombinaison) sur toutes les colonnes. Cette communication diffuse également le résultat *x* qui se retrouve donc stocké sur toutes les lignes de la grille. Le vecteur *y* s'obtient de manière symètrique en opérant simultanément des produits scalaires sur les lignes, le résultat du second *reduce-and-spread* étant stocké sur toutes les colonnes.
Le calcul du scalaire *s* consiste en un produit scalaire de *x* par *z* et de trois opérations scalaires (*s=(1+s*c)/c*). Les vecteurs *x* et *z* étant stocké sur toutes les lignes de la grille, il faut effectuer un *reduce-and-spread* le long des lignes de la grille, cette communication avec recombinaison effectue le produit scalaire en parallèle sur toutes les lignes de la grille (le scalaire *s* est alors stocké sur la dernière colonne de la grille) et assure la diffusion du résultat le long des lignes, s est donc finalement stocké sur tous les processeurs virtuels.
Le calcul de l'inverse perturbée *Amoins1* en est donc grandement facilité : toutes les données se trouvent sur place. En effet *y* est stocké sur toutes les colonnes (et on y accède bien par colonne lors de la boucle sur *i*), *x* sur toutes les lignes (et l'on y accède bien par ligne lors de la boucle sur *j*) et *s* sur tous les processeurs virtuels. Il n'y a donc aucune communication lors de la dernière phase de l'algorithme.

Les temps d'exécution obtenus par cette implantation sur la Connection Machine sont donnés ci-dessous (en secondes).

Taille de la matrice	Temps d'exécution
128x128	0.0130
256x256	0.0232

512x512	0.0861
1024x1024	0.03308

Le nombre d'opérations arithmétiques nécessaires à la méthode de Woodbury pour des transformations de rang 1 étant de $O(6n^2)$, on obtient les performances suivantes.

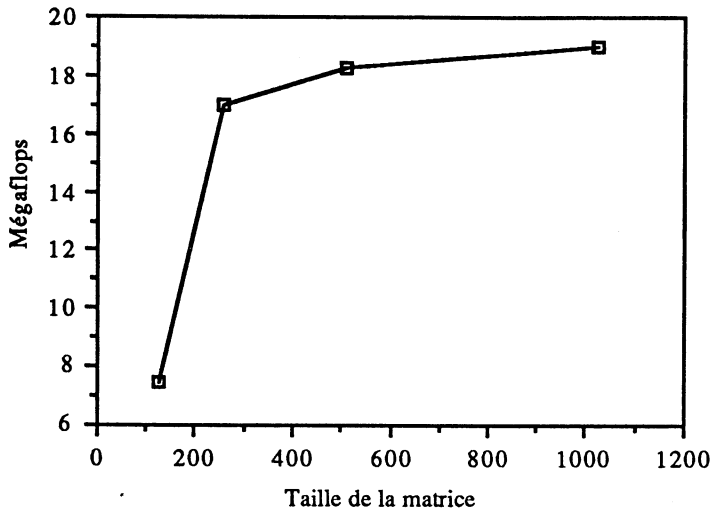


Figure 99 : performances de la méthode de Woodbury (rang1)

Ici encore, l'amélioration des performances avec la taille des matrices est imputable au *VP-ratio* qui augmente et permet donc une meilleure utilisation de la CM2. On voit ici nettement que la courbe s'écrase pour les plus grandes tailles et que par conséquent un *VP-ratio* élevé n'est pas suffisant pour masquer complétement le temps d'accès à la machine hôte qui envoie les instructions au séquenceur de la CM2.

Chapitre 8

Architecture MIMD à topologie reconfigurable : le MégaNode

Où l'on présente un second ordinateur MIMD à mémoire distribuée : le MégaNode ; en insistant particulièrement sur son caractère spécifique : la topologie de ses processeurs est entièrement reconfigurable. Les expérimentations réalisées sur cette machine ont pour but de montrer qu'en reéquilibrant calculs et communications, on obtient des conclusions assez différentes de celles obtenues dans le cas du FPS T40 (qui est une machine de même nature).

8.1. Présentation du MégaNode

Nous allons dans un premier temps décrire les ordinateurs de la famille du MégaNode, puis nous intéresser à l'architecture globale de cette machine, pour ensuite traiter séparément chacune des composantes spécifiques qui font son originalité.

8.1.1. Les machines SuperNode

Les machines SuperNode sont des architectures MIMD reconfigurables issues du projet Esprit P1085, comprenant des universités du Royaume-Uni, des entreprises privées allemandes, britanniques et françaises, ainsi que l'Institut de Mathématiques Appliquées de Grenoble. Ce programme européen (aujourd'hui mené à bien) avait pour but de développer une architecture multiprocesseurs à hautes performances avec un coût faible. Ce projet d'une durée de trois ans (qui a débuté en 1985) a conduit à la réalisation d'un microprocesseur : le Transputer, des architectures de la famille SuperNode (T-Node, TandemNode et MégaNode), ainsi qu'aux logiciels systèmes associés [Nico].
Ces machines sont commercialisées par une société française : Telmat. Il s'agit de machines à base de Transputers dont la caractéristique principale est la reconfigurabilité dynamique de la topologie des processeurs suivant une structure hiérarchique. Une machine SuperNode posséde entre 16 et 1024 processeurs qui lui permettent d'atteindre des performances de crête variant de 24 à 1500 Mflops.

Ces ordinateurs sont constitués de plusieurs unités modulaires, appelées *nodes*. Elles regroupent sur une même carte 32 Transputers T800 de travail, plus quelques autres Transputers chargés de tâches internes (il s'agit de T414 et leur nombre varie suivant les configurations). Ce système permet la constitution de toute une gamme de machines :

- le T-Node 16/32 qui implémente un seul module de base (qui dans le cas d'un T-Node 16 n'est pas complet et contient seulement 16 processeurs)
- le TandemNode qui comprend 64 processeurs de travail sous la forme de 2 *nodes* de base
- le MégaNode est la véritable machine multi-*nodes* et comprend un nombre variable de *tandems* permettant de disposer de 128 à 1024 processeurs de travail

Seul le MégaNode à 128 processeurs (machine dont dispose le LMC) va être décrite en détail ci-aprés. Le lecteur pourra trouver une présentation des T-Node et TandemNode dans [Telm].

8.1.2. Architecture du MégaNode/128

La principale différence entre le MégaNode et les autres ordinateurs de la famille des SuperNode réside dans la façon dont les différents modules sont interconnectés. Les T-Nodes ne comportent en effet qu'un seul module et n'ont donc pas de système de connexion, le TandemNode connecte ses deux *nodes* directement switch à switch. Le MégaNode possède quant à lui tout un système hiérarchisé de connexions entre *tandems* (décrit plus en détails en 8.1.3).

L'architecture du MégaNode est illustrée par la figure ci-dessous.

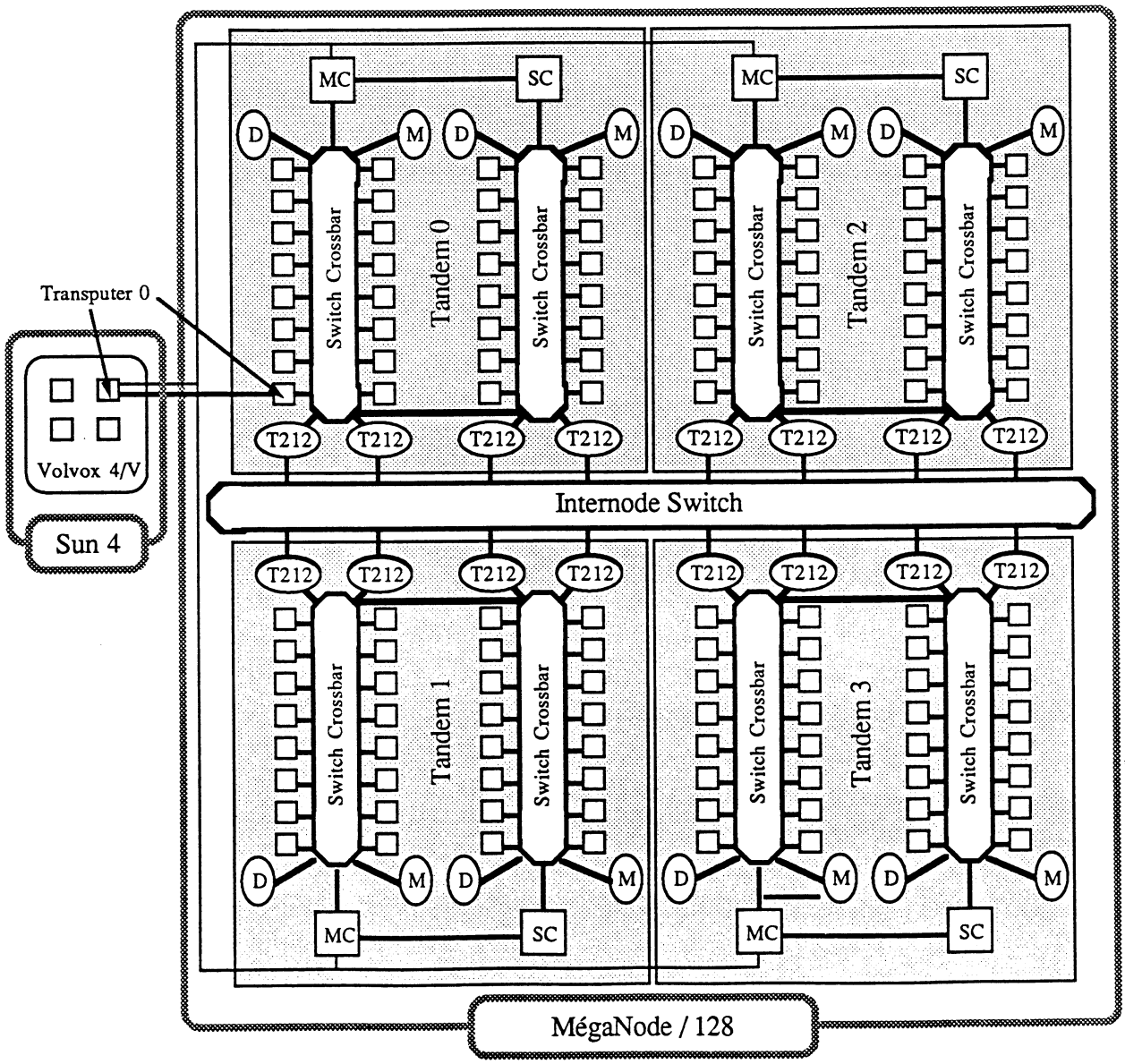


Figure 100 : architecture du MégaNode

Le MégaNode est donc une machine mono-utilisateur reliée à un ordinateur hôte (un Sun 4 par exemple) qui contient une carte mulitprocesseurs à base de Transputers (au LMC une carte Volvox 4/V, fabriquée par Archipel, avec 4 T800 est utilisée). Cette carte assure toutes les entrées / sorties (y compris le chargement des programmes) avec le MégaNode et c'est aussi sur ces Transputers que sont compilés les programmes destinés au MégaNode.
Un MégaNode/128 est donc constitué de 4 *tandems* contenant chacun 32 Transputers T800 de travail disposant chacun de 1Mo de mémoire vive. Ces Transputers sont reliés entre eux par deux niveaux de connexions : au niveau des tandems par un crossbar RSRE 72x72 entièrement configurable par son Transputer de contrôle, et pour les communications entre tandems par un *internode switch* constitué de 4 crossbar C004 32x32. D'autre part, tous les Transputers de contrôle sont reliés par un bus qui permet par exemple de véhiculer les requêtes de synchronisation des Transputers. Les Transputeurs de contrôle ont des fonctions diverses : certains assurent la gestion des disques, d'autres les communications avec l'*internode switch* et enfin le rôle de contrôleur pour les *nodes* de 8 Transputers de travail.
Les différents composants du MégaNode sont décrits plus en détails ci-après.

8.1.3. Les différents composants du MégaNode

Le MégaNode intégre dans son architecture 3 types de Transputers : des T800 pour les processeurs de travail, des T414 pour le contrôle et des T212 pour la gestion des communications avec l'*internode switch*. Le lecteur peut se reporter en 3.2 ou à[ViTr], [DeSm] et [Inmo] pour plus de détails concernant les Transputers.
Le Transputer T800 est identique au T414 décrit au chapitre 3 mais il intégre en plus une unité de calculs flottants. Ses performances de crête sont de 10 Mips et 1.5 Mflops pour une horloge interne à 20 Mhz. Le T212 est un microprocesseur spécialement dévolu à des tâches systèmes, et qui sur le MégaNode n'est pas accessible à l'utilisateur.

Les communications au sein d'un tandem sont assurées par un commutateur RSRE qui est un crossbar 72x72, implanté sous la forme de deux composants NEC qui sont fonctionnellement équivalents à deux crossbars 72x36. Rappelons qu'un réseau crossbar (ou matrice de croisement) est le type de réseau d'interconnexion conceptuellement le plus simple : il permet de relier n'importe quelle entrée à n'importe quelle sortie. Le commutateur RSRE est piloté par un Transputer de contrôle afin que l'on puisse reconfigurer les Transputers de travail suivant toute topologie au choix de l'utilisateur. Ce connecteur est implanté au sein d'un tandem parce qu'il ne recalibre pas les signaux qu'il reçoit et n'introduit donc que peu de délai dans les communications.
Les communications entre les tandems d'un MégaNode sont assurés par l'*internode switch*. Ce composant est réalisé de façon modulaire avec des C004 suivant la taille de la machine, assurant à ce niveau aussi une parfaite modularité au MégaNode. Le C004 est

un crossbar 32x32 qui intégre en sortie un dispositif de recalibrage des signaux. Ce dispositif permet aux messages de parcourir une plus gande distance sur les fils et de traverser un second niveau de commutation. Ceci augmente de façon importante le temps de traversée du C004 : environ une fois et demi la durée de traversée d'un bit.
Pour plus de détails sur l'implantation des niveaux de commutation, le lecteur peut se reporter à [MuWa].

Un système de bus de contrôle relie tous les Transputers du MégaNode : chaque *node* posséde un Transputer de contrôle, et ceux-ci sont reliés entre eux par un bus de contrôle. De plus à l'intérieur de chaque *node*, un bus relie tous les Transputers (de travail ou non) au Transputer de contrôle. Ce bus fonctionne suivant le mode maître-esclave, le maître étant le Transputer de contrôle, et il permet une synchronisation rapide des Transputers. De plus, d'autres fonctionnalités sont supportées par ce système de bus comme un *reset* sélectif, l'envoi de messages, un blocage de tous les Transputers pour remontée d'informations de déboggage (sans altérer l'état des liens des Transputers).
Tous ces services sont transparents à l'utilisateur, et la plupart ne sont même pas implantés dans les environnements de programmation actuellement disponibles (en particulier le système d'aide au déboggage).

Toutes les entrées / sorties avec l'ordinateur hôte se font par l'intermédiaire de la carte Volvox 4/V d'Archipel. Cette carte est équipée de 4 Transputers T800 avec 4Mo de mémoire pour chacun. Plus précisément les communications avec le MégaNode se font par l'intermédiaire du processeur 0 de la carte dont le lien 0 est en communication avec le lien 0 du Transputer de travail 0 du tandem 0 du MégaNode.

8.2. Environnement logiciel

Un Supernode peut être programmé avec divers environnement de programmation : en multi-utilisateurs sous Helios, puis en mode mono-utilisateur tout d'abord avec Occam (le langage natif du Transputer) sous TDT et ensuite en C parallèle avec les environnements C3L ou Logical C.
L'environnement de développement le plus communément utilisé avec le MégaNode/128 dont dispose le LMC est le langage C de 3L, qui va donc être décrit plus en détails que les autres environnements.

8.2.1. Les environnements de programmation usuels

Helios est un système d'exploitation qui permet à plusieurs utilisateurs d'accéder au MégaNode dans un environnement de style Unix ([Peri]). La machine est divisée en un

certain nombre de sites, chaque utilisateur configurant la topologie de son site.
Un programme est alors vu comme une collection de tâches qui sont réparties de façon à optimiser la charge du réseau, les communications et l'accès à certaines ressources physiques. Le placement d'une tâche sur un processeur peut soit être confié au système, soit être explicitement décrit par l'utilisateur.
Le système d'exploitation est constitué de serveurs répartis dans le réseau dont le rôle est de contrôler la localisation et les communications des objets Hélios.
Les communications entre deux tâches d'un programme s'effectuent à travers des canaux bidirectionnels, lors d'une communication le système engendre un processus assurant la communication entre les deux tâches. Le mécanisme de communication diffère suivant qu'il s'agit de communications au sein d'un tandem ou de communication passant par l'*internode switch*, mais cela est transparent à l'utilisateur. Tout processeur peut communiquer avec l'ordinateur hôte (même s'il n'est pas en communication directe avec l'hôte), le routage vers l'hôte est pris en compte de manière transparente par le système d'exploitation.
Il existe trois bibliothèques principales pour les communications : les bibliothèques *Message* et *System* sont en principe destinées à la programmation système et la bibliothèque *Posix* au développement d'applications.

Le langage natif du Transputer est Occam qui est disponible sur le MégaNode sous TDT. Le MégaNode se programme en Occam 2 qui est une évolution du langage Occam qui inclut entre autre des définitions de type. Un programme Occam est constitué d'une suite de processus qui s'exécutent en général sur plusieurs processeurs. Aucune gestion système des entrées / sorties n'est effectuée : seul le processeur 0 du tandem 0 peut communiquer avec le Transputer 0 de la carte Volvox 4/V qui permet les communications avec l'hôte.
Le placement des processus et des canaux de communications sur les Transputers sont à la charge de l'utilisateur qui doit éditer un fichier descripteur (qui est ensuite compilé par un utilitaire qui génère le fichier de configuration physique des Transputers du MégaNode).
Il existe sous TDT plusieurs bibliothèques de fonctions mathématiques et d'entrées / sorties.

Le C de Logical System est assez semblable au C de 3L décrit ci-dessous : les programmes sont décrits sous forme de modules tournant chacun sur des processeurs différents. Là aussi, seul le processeur 0 de la carte Volvox 4/V est autorisé à faire des entrées / sorties.
Il faut également créer soi-même le placement des modules sur les processeurs, le chemin de chargement des Transputers et l'affectation des canaux sur les différents Transputers. Plusieurs utilitaires facilitent ce travail (citons par exemple "genenif" et "TéNOR_RITA").

8.2.2. Le langage C de 3L

Ce langage est fondé sur le modèle de processus séquentiels communicants : un programme écrit en C3L est constitué d'un ensemble de tâches s'exécutant en parallèle (en général placées sur des Transputers différents, mais la possibilité d'affecter deux tâches au même processeur existe).
Les communications sont bloquantes et se font via des canaux mono-directionnels. Une communication entre deux tâches placées sur deux processeurs différents n'est possible que s'il existe un canal physique entre ces deux processeurs. De plus, il n'est pas possible de définir plusieurs canaux logiques sur un seul canal physique.
La configuration du réseau, le placement des tâches sur les processeurs et l'association entre canaux logiques et liens physiques doivent être explicitement décrits par l'utilisateur. Mais restent heureusement (contrairement au C de Logical System) à un niveau logique : l'utilisateur n'a pas à se préoccuper de l'affectation des canaux logiques de tâches aux liens physiques des Transputers, ni de la position physique des Transputers dans la machine (seule une numérotation logique des Transputers est utilisée) ; la transcription au niveau physique est réalisée par l'utilitaire TéNOR_RITA. Seules les tâches placées sur les Transputers de la carte Volvox 4/V peuvent effectuer des entrées / sorties avec l'hôte : pour toutes les autres tâches, une entrée / sortie avec l'hôte passe par un routage explicite à la charge de l'utilisateur.
La bibliothèque *Ouf* [BFAl] a été utilisée pour l'envoi et la réception de messages en parallèle sur un même Transputer. Cette bibliothèque gère la notion de canaux, les communications et les processus. Les communications se font selon le principe du rendez-vous, la tâche émettrice émettant vers un canal, comme en Occam. Les canaux sont unidirectionnels. Il n'existe aucune primitive permettant directement d'effectuer des communications sur plusieurs canaux en parallèle, il faut pour cela créer des processus (suivant le mode *fork and join*) qui effectuent chacun une communication, puis se synchronisent avant d'être détruit par le processus père. Signalons une contrainte importante : la création de processus nécessite la création d'un espace de travail assez important (de l'ordre 50 à 100 ko) au sein du Transputer où le processus est créé.

Signalons pour terminer l'existence d'un debogger (réalisé au sein du LMC [BFAl]). Il s'agit d'un programme parallèle qui tourne sur chaque Transputer et qui permet d'effectuer du déboggage "post-mortem" en rapatriant un certain nombre d'informations (en particulier au niveau des communications) dans un fichier sur la machine hôte.

8.2.3. Les contraintes de programmation

Contrairement à ce qui a été annoncé précédément, avec l'environnement dont on dispose au LMC, il n'est pas actuellement possible de reconfigurer dynamiquement la topologie du MégaNode. Cette contrainte oblige non seulement à choisir soigneusement son réseau

de processeurs, mais aussi parfois à faire de délicates concessions si le programme nécessite plusieurs topologies s'excluant mutuellement.
Par exemple, pour la diffusion un excellent compromis entre l'efficacité et la difficulté de mise en œuvre est l'arbre ternaire (cf 8.3.1). Supposons qu'à un stade donné du programme l'on veuille diffuser une donnée à partir d'un processeur *a*, on décide de configurer le MégaNode en arbre ternaire de sommet *a*. Plus avant dans le programme, une autre donnée doit être diffusée à partir d'un processeur *b* situé à la base de l'arbre ternaire précédent ...

Hélios n'étant pas disponible sur le MégaNode/128 du laboratoire, dans tous les autres langages seul le processeur 0 de la carte Volvox 4/V peut effectuer des entrées / sorties entre le MégaNode et le Sun 4 hôte. Tous les autres processeurs doivent remonter leurs informations jusqu'à ce processeur de façon explicite ; par conséquent, il est absolument nécessaire d'éviter d'effectuer des entrées / sorties sur ces processeurs si l'on désire conserver des performances élevées.

Toujours à propos des communications, il faut garder à l'esprit le fait que le MégaNode/128 est constitué de 4 tandems qui comuniquent entre eux par l'*internode switch* (environ deux fois plus lent que les *switchs RSRE* équipant les *nodes*). Par conséquent il semble raisonnable de s'en tenir à deux stratégies : soit utiliser 32 processeurs sur un seul tandem, soit passer directement à 128 processeurs sur les 4 tandems. En effet, pour peu qu'un programme s'exécute sur 33 processeurs, toutes les communications émises ou reçues par le dernier Transputer passent par l'*internode switch* et introduisent un temps de communication double pour toutes les communications ; et ce pour un gain en parallélisme d'un processeur par rapport à une configuration à 32 Transputers avec des communications deux fois plus rapides.

Enfin, le placement des tâches sur les Transputers de travail est effectué de façon transparente à l'utilisateur par le système via un arbre de chargement mais aucune synchronisation globale de tous les processeurs n'est effectuée : les processeurs commencent à exécuter leur tâche dès que celle-ci a été chargée en mémoire. Cela peut occasionner des problèmes si le programme est par exemple basé sur une stratégie asynchrone telle celle décrite en 4.3 : l'utilisateur doit en effet s'assurer lui-même que tous les Transputers de son réseau sont synchronisés lors du lancement du programme (et c'est un problème délicat avec les deux niveaux de switch qui induisent des vitesses de communications hétérogènes).

8.3. Expérimentations sur le MégaNode

Toutes les expérimentations ont été réalisées sur le MégaNode/128 du laboratoire en C3L, en utilisant la bibliothèque *Ouf* pour la gestion des communications en parallèle. La topologie des processeurs est discutée ci-après.
Les expérimentations réalisées sur le MégaNode ont pour but de mettre en œuvre la méthode du gradient conjugué pour les systèmes consécutifs (décrite en 2.4) en utilisant une vraie stratégie distribuée, ce qui n'avait pas été possible sur le FPS T40 où calculs et communications n'étaient pas équilibrés.

8.3.1. Choix de la topologie

Le choix de la topologie a été guidé par la nature des expérimentations réalisées sur le MégaNode, ainsi que par la possibilité de mettre en œuvre efficacement (au vu des particularités de cette machine) ces algorithmes.

8.3.1.1. Choix d'une stratégie de parallélisation

La méthode du gradient conjugué préconditionné pour systèmes consécutifs utilise comme préconditionnement une factorisation de Cholesky pour les résolutions par le gradient conjugué.
Le choix entre une méthode entiérement distribuée et une méthode type "maître-esclave", similaire à celle utilisée pour le gradient conjugué sur le FPS T40, a été guidé par les caractéristiques propres du MégaNode. En effet une méthode où tous les calculs sont entièrement distribués nécessite périodiquement (pour la réalisation de produits scalaires par exemple) que tous les processeurs communiquent les valeurs qu'ils viennent de calculer à tous les autres.
Cela met en œuvre des communications globales du type *personalized all-to-all* selon la terminologie de [JoHo] : chaque processeur envoie des données différentes à chacun des autres. Pour rester relativement efficace (car c'est tout de même la forme de communication la plus coûteuse) ce type de communication met en œuvre des algorithmes complexes quelle que soit la topologie du réseau de processeurs. Ces algorithmes ne peuvent être implantés si tous les processeurs ne sont pas synchronisés : le cas d'une simple stratégie pipeline est alors déjà très délicat. Or on a vu en 8.2.3 qu'on ne pouvait en pratique assurer la synchronisation de tous les processeurs en début de programme.
Pour cette raison, on a préféré choisir une stratégie de type "maître-esclave" où les calculs nécessitant des informations globales (comme des produits scalaires) sont effectués sur le processeur maître et tous les autres calculs sont distribués aux processeurs esclaves (qui renvoient ensuite leurs résulats au processeur maître).

8.3.1.2. Choix de l'arbre ternaire

Une topologie en arbre est particulièrement adaptée à ce type de stratégie "maître-esclave". Les Transputers ayant 4 canaux physiques de communication, l'arbre ternaire a donc naturellement été adopté comme topologie du MégaNode pour ces expérimentations. Avec les 128 Transputers de travail disponibles sur le MégaNode, on peut réaliser un arbre ternaire de profondeur 4 qui regroupe 121 processeurs (voir figure ci-dessous). Notons que l'on peut implanter un arbre ternaire de degré 2 seulement (regroupant donc 13 processeurs) sur un tandem de 32 Transputers.

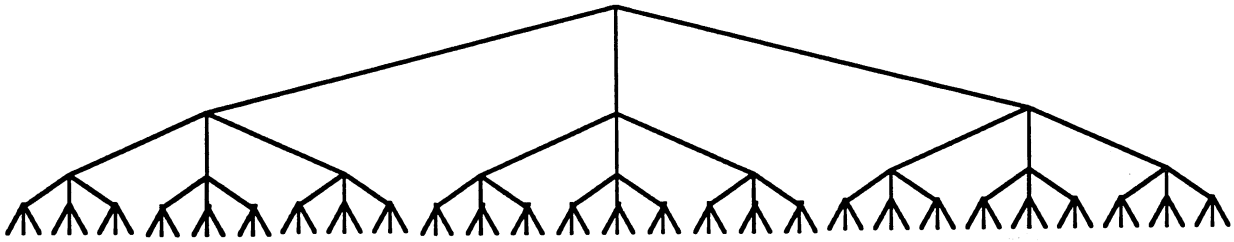


Figure 101 : arbre ternaire de degré 4

Une autre raison en faveur du choix de l'arbre ternaire est la mise au point d'un algorithme performant de diffusion et regroupement personnalisés bien adapté au MégaNode.

8.3.1.3. Diffusion et regroupement personnalisés dans un arbre ternaire

La stratégie de parallélisation type "maître-esclave" adoptée nécessite principalement deux types de communication : une diffusion et un regroupement personnalisés. Ces deux types de communications sont utilisés par le maître pour envoyer des données aux processeurs esclaves et à ceux-ci pour rapatrier leurs résultats sur le processeur maître.
Un problème spécifique se pose cependant dans le cas des résolutions de systèmes triangulaires, mais il est discuté en 8.3.3.2.
Rappelons brièvement ce que signifie le terme diffusion personnalisée : une diffusion (ou *broadcast* en anglais, et *one-to-all* dans la terminologie de [JoHo]) consiste pour un processeur à envoyer le même message à tous les autres processeurs. Une diffusion personnalisée est une diffusion où le processeur émetteur envoie des messages différents à chacun des autres processeurs (*dispatch* en anglais).
Un regroupement personnalisé est la communication duale : tous les processeurs envoient chacun un message (qui tous sont différents) à un même processeur.

L'algorithme présenté ci-aprés présente plusieurs avantages :

- il est étudié pour des communications entre le sommet de l'arbre et le reste de celui-ci (ce qui est en majorité le cas qui nous intéresse ici)
- il est bien adapté à la programmation du MégaNode
- les modifications entre la diffusion personnalisée et le regroupement personnalisé sont minimes
- pour des messages de grande taille, les communications peuvent être pipelinés

L'algorithme est tout d'abord décrit dans le cas de la diffusion personnalisée. On suppose que les messages à envoyer aux autres processeurs sont rangés en mémoire de façon continue (ce qui est le cas pour des vecteurs par exemple). L'idée de base est très simple et peut être vue récursivement : les processeurs reçoivent un message de leur père, garde la partie qui leur est destinée et en renvoie le tiers à chacun de leurs fils.
L'algorithme est donné ci-dessous (en pseudo-code) dans le cas de la diffusion personnalisée d'un message a de longueur n dans un arbre ternaire de profondeur p, pour un processeur quelconque. Les primitives *Envoi(i, adresse, longueur)* et *Réception(i, adresse, longueur)* désignent une émission (respectivement une réception) sur le lien i d'un message dont le début est stocké en mémoire à l'adresse *adresse* et de longueur *longueur*. On convient que le lien 0 permet de communiquer avec le père, le lien 1 avec le fils gauche, le lien 2 avec le fils du milieu et le lien 3 avec le fils droit. *Niveau* désigne le niveau dans l'arbre (0 correspond à la racine et p aux feuille terminales).

$$\text{NbProc} = \sum_{i=0}^{p} 3^i$$

```
Longueur=n/NbProc
si Niveau=0
   Taille=(n-Longueur)/3
   pour i=1 jusqu'à 3
      Envoi(i,Adresse-(i-1)*Taille,Taille)
si Niveau>0 et Niveau<p
```

$$\text{Taille} = \frac{\text{NbProc}}{3^{\text{Niveau}}} - \sum_{i=1}^{\text{Niveau}} \frac{1}{3^i}$$

```
   Réception(0,Adresse,Taille)
   Taille=(Taille-Longueur)/3
   pour i=1 jusqu'à 3
      Envoi(i,Adresse+(i-1)*Taille,Taille)
si Niveau=p
   Réception(0,Adresse,Taille)
```

La figure ci-dessous illustre le déroulement de l'algorithme sur un arbre ternaire de profondeur 2.

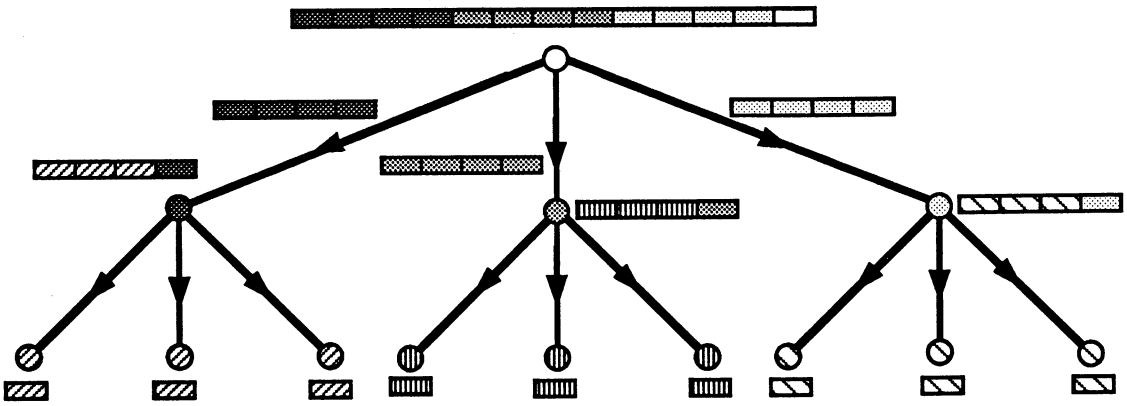


Figure 102 : déroulement de la diffusion personnalisée sur un arbre ternaire de profondeur 2

La programmation de cet algorithme sur le MégaNode en C 3L est immédiate car un programme est composé de tâches qui sont ensuite placées sur les Transputers. Il suffit d'expliciter les trois *si* de l'algorithme : une tâche pour la racine (qui émet les messages à leurs fils), la même tâche sur tous les processeurs du milieu de l'arbre ternaire (qui reçoivent les messages de leur père et renvoient les messages à leurs fils) et la même tâche sur toutes les feuilles terminales de l'arbre (qui reçoivent les messages de leur père).

L'algorithme de regroupement personnalisé consiste en les mêmes étapes à dérouler dans l'autre sens : les feuilles terminales de l'arbre commencent par émettre vers leur père, et ainsi de suite jusquà que la racine reçoivent les trois messages de ses fils. Avec les mêmes notations que précédemment, l'algorithme devient :

$$NbProc = \sum_{i=0}^{p} 3^i$$

```
Longueur=n/NbProc
si Niveau=0
   Taille=(n-Longueur)/3
   pour i=1 jusqu'à 3
      Réception(i,Adresse-(i-1)*Taille,Taille)
si Niveau>0 et Niveau<p
```

$$Taille = \frac{NbProc}{3^{Niveau}} - \sum_{i=1}^{Niveau} \frac{1}{3^i}$$

```
   pour i=1 jusqu'à 3
      Réception(i,Adresse-(i-1)*Taille,Taille)
   Taille=Taille*3+Longueur
   Envoi(0,Adresse,Taille)
si Niveau=p
   Envoi(0,Adresse,Longueur)
```

Sur la figure 102, seul le sens des communications est à inverser.

La version pipeline de cet algorithme n'a pas été implantée à cause de la faible profondeur du plus grand arbre (p=4, soit 121 processeurs) et de la taille relativement faible des vecteurs mis en jeu lors de la résolution des systèmes linéaires consécutifs (en fait 2057, voir à ce propos 8.3.3).

8.3.2. Détermination de la granularité du MégaNode

Pour cette machine, une approche plus pragmatique a été utilisée que la classique modélisation informatique présentée en 4.1 dans le cas du FPS T40. La question qui se posait dans le cas du MégaNode est en fait très simple : avec des communications et des calculs équilibrés, le MégaNode avait-il une granularité de n ? Ou encore, est-il plus intéressant d'effectuer des calculs d'un coût en n (soit la taille du problème à résoudre) localement ou de façon distribuée ?
Les expérimentations qui ont été ménées ici ont été réalisées en collaboration avec P. Michallon.
Les expérimentations ci-aprés ont été réalisées dans cette optique et dans chacun des trois cas, produit scalaire, saxpy et produit matrice-vecteur, nous justifions les transferts de données réalisés dans l'optique distribuée de type "maître-esclave".

8.3.2.1. Influence du parallélisme sur les communications

Nous allons étudier ici l'influence de l'échange (émission ou réception) des données en parallèle sur plusieurs liens des Transputers. Rappelons que dans le cas du langage C 3L, cet échange est réalisé via la bibliothèque *Ouf* qui permet de créer des processus fils au sein de chaque Transputer afin de réaliser l'échange de ces données. Ces processus sont alors créés selon le mode "fork and join" : ils sont créés, échangent leurs données puis disparaissent après synchronisation.
Cette étude a été menée directement dans le cas du produit scalaire (décrit ci-après en 8.3.2.2). Les résultats sont les suivants : la courbe n°1 est celle où toutes les communications se font en séquentiels (émission et réception), la courbe n°2 est celle ou les émissions de la racine sont parallèles et la courbe n°3 est celle où toutes les émissions se font en parallèle.

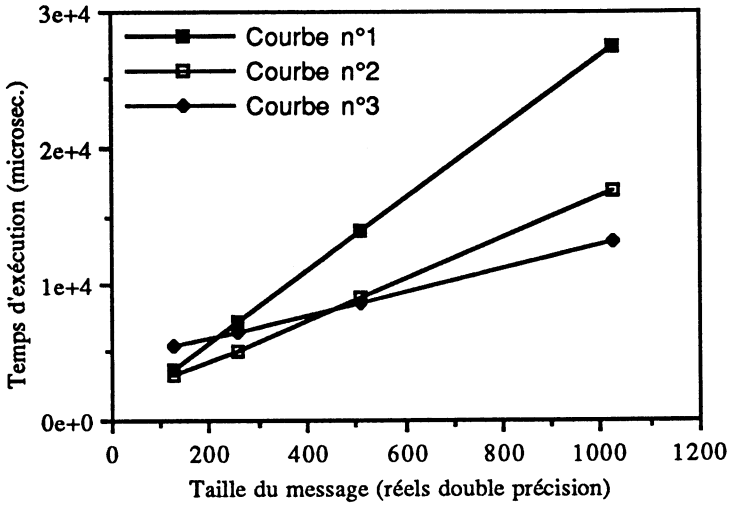


Figure 103 : influence du parallélisme au niveau des canaux de communications

On voit donc que les communications en mode séquentiel ne sont intéressantes que pour de petites tailles (jusqu'à 256 réels) et qu'ensuite les communications en parallèle sur les liens d'un même Transputer sont beaucoup plus intéressantes.
Cette inversion de la tendance s'explique facilement : lorsqu'on réalise des communications sur les liens d'un même Transputer, il faut créer autant de processus concurrents qu'il y a de communications (moins une, on peut en effet très bien émettre sur le processus père). Cette création prend alors un certain temps ainsi que sa gestion sur le Transputer. Mais lorsque les messages sont de grande taille, le gain dû au parallélisme surclasse alors la perte occasionnée par le création et la gestion de processus multiples au sein d'un même Transputer.

Par conséquent toutes les communications réalisées par la suite utilisent l'envoi et la réception en parallèle sur chaque Transputer.

8.3.2.2. Le produit scalaire

Dans la stratégie "maître-esclave" qui est choisie pour implanter les méthodes de résolutions de systèmes linéaires successifs, la réalisation d'un produit scalaire distribué nécessite la diffusion personnalisée des deux vecteurs et le regroupement personnalisé (avec accumulation) du résultat scalaire.
Par conséquent les résultats obtenus ci-dessous comprennent la diffusion personnalisée de deux vecteurs, les calculs de produits scalaires locaux et le regroupement personnalisé avec accumulation du scalaire résultat.

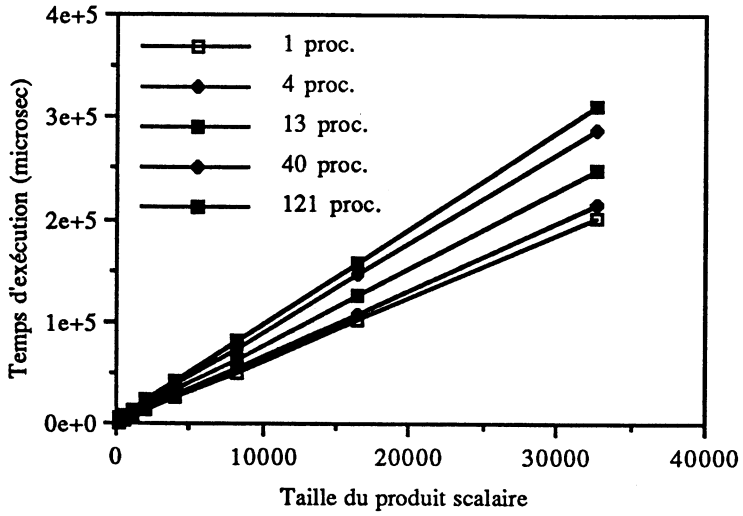


Figure 104 : produit scalaire distribué sur un arbre ternaire

Par conséquent, quelle que soit la taille du vecteur et le nombre de processeurs, il est plus intéressant d'effectuer un produit scalaire sur place plutôt que de le distribuer. Il semble donc que sur l'exemple du produit scalaire, la granularité du MégaNode ne soit pas de *n*.

8.3.2.3. Le saxpy

Avec la stratégie distribuée type "maître-esclave adoptée, un saxpy nécessite la diffusion personnalisée du vecteur résultat (pour que la stratégie soit véritablement distribuée, il ne doit avoir aucune donnée supposée en place) et du vecteur opérande ainsi que du scalaire opérande, un regroupement personnalisé du vecteur est nécessaire, pour rapatrier le vecteur résultat sur le processeur maître.

Par conséquent les résultats ci-dessous comprennent la diffusion personnalisée de deux vecteurs et d'un scalaire, les saxpy locaux et le regroupement personnalisé d'un vecteur.

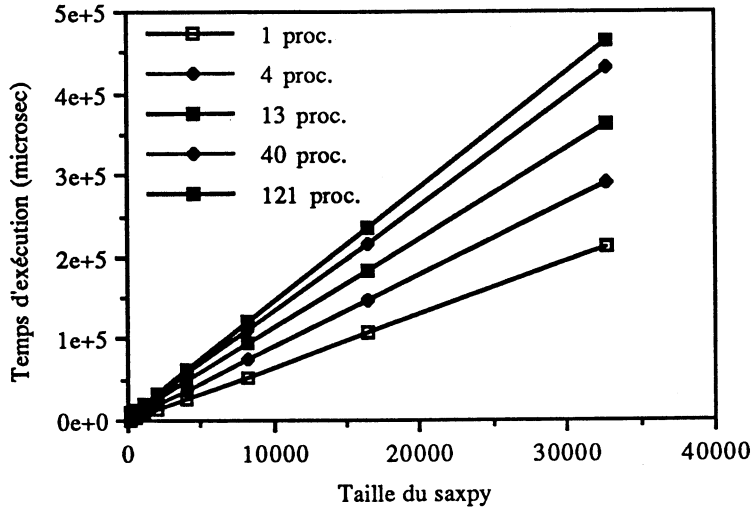


Figure 105 : saxpy distribué sur un arbre ternaire

Donc dans le cas du saxpy également, il est plus avantageux d'effectuer les calculs localement plutôt que de les distribuer, quelle que soit la taille du vecteur et du nombre de processeurs.
De plus le coût d'un saxpy distribué est plus élevé que celui d'un produit scalaire distribué, ce qui est dû au surcroit de communications occasionné par la diffusion personnalisée du scalaire opérande.

Il apparaît donc clairement que sous les hypothèses adoptées, la granularité du MégaNode n'est pas de n. Nous allons maintenant voir si elle est de n^2 en étudiant le cas du produit matrice-vecteur.

8.3.2.4. Le produit matrice-vecteur

Sous l'hypothèse de calcul distribué type "maître esclave" adoptée, l'exécution distribuée d'un produit matrice-vecteur nécessite la diffusion personnalisée du vecteur opérande et le regroupement personnalisé du vecteur résultat, la matrice étant distribuée sur les processeurs.

La même étude que précédément ne peut être menée : il faudrait que la taille de la matrice varie suivant des diviseurs du nombre de processeurs (afin de ne pas fausser les résultats avec des calculs inégalement répartis, qui pouvaient eux être tolérés dans le cas d'un coût en n). Par conséquent l'expérimentation a été menée sur la plus grande taille de matrice qu'il soit possible de faire tenir sur un MégaNode/128 configuré en arbre ternaire de profondeur 4 : 2057x2057, soit 17 lignes de la matrice par processeur.
Parmi les différentes formes possibles du produit matrice-vecteur (voir 4.2) la forme produit scalaire a été naturellement choisie car elle est moins coûteuse que la forme saxpy

au niveau de la remontée des résultats. En effet, la forme produit scalaire nécessite le regroupement personnalisé du vecteur résultat : au niveau des feuilles terminales les communications sont de longueur $n/121$. Tandis que la forme saxpy nécessite le regroupement avec accumulation des contributions des divers processeurs : au niveau des feuilles terminales les communications sont de longueur n, avec à chaque étape la somme de deux vecteurs de longueurs n.

Le résultats obtenus sont les suivants :

1 processeur	26.4134 s	0.32 Mflops
121 processeurs	0.4396 s	19.25 Mflops

Le résultat obtenu sur 1 processeur est une extrapolation de celui du produit scalaire sur 1 processeur : stocker une matrice 2057x2057 nécessiterait en effet près de 33 Mo de mémoire, alors que l'on ne dispose au maximum que de 4 Mo (sur les Transputers de la carte Volvox 4/V).
L'augmentation des performances obtenues est de deux ordres : tout d'abord le volume de calculs locaux a augmenté d'un facteur 16 (le nombre de lignes de la matrice stockées sur chaque Transputer) et les communications mises en œuvres sont restées inchangées.

Par conséquent, sous les hypothèses adoptées, la granularités du MégaNode est de n^2.

A titre de comparaison, une implantation sophistiquée, réalisée par H. Frydlender en Occam (les données sont supposées en place et les calculs sont distribués par blocs, les processeurs échangeant des données sur un tore 2D par des schémas de type *all-to-all personnalisés*), sur 120 processeurs donnent des performances de 27.9 Mflops [Fryd].

8.3.3. Implantation de la méthode du gradient conjugué préconditionné pour systèmes linéaires consécutifs

Notre but ici est d'expérimenter la méthode basée sur le gradient conjugué pour les systèmes linéaires successifs décrite en 2.4 sur des matrices de taille ne tenant pas sur un seul Transputer (ce cas ayant déjà été étudié dans le cas du FPS T40). La stratégie distribuée de type "maître-esclave" décrite précédément est utilisée dans les trois implantations suivantes : la factorisation de Cholesky (8.3.3.1), la résolution de systèmes triangulaires (8.3.3.2) et le gradient conjugué préconditionné (8.3.3.3).

8.3.3.1. Implantation distribuée de la factorisation de Cholesky

La factorisation de Cholesky permet de factoriser une matrice A (de taille nxn) en

$L.D.L^t$ où la matrice L est triangulaire inférieure et la matrice D diagonale (en pratique stockée dans un vecteur). Rappelons tout d'abord la version ligne algorithme (qui est donné ci-après en pseudo-code), les chiffres entre accolades sont les opérations distribuées sur les processeurs (voir ci-après).

```
pour k=0 jusqu'à n-1
        pour i=0 jusqu'à k-1
           aux[i]=d[i]*L[k,i]              {1}
        d[k]=A[k,k]
        pour i=0 jusqu'à k-1
           d[k]=d[k]-L[k,i]*aux[i]         {2}
        pour i=k+1 jusqu'à n-1
           L[i,k]=A[i,k]
           pour j=0 jusqu'à k-1
              L[i,k]=L[i,k]-L[i,j]*aux[j] {3}
           L[i,k]=L[i,k]/d[k]              {4}
```

Dans le cas de la factorisation de Cholesky, la taille du problème retenu implique un traitement distribué de l'ensemble des calculs (le stockage d'une matrice 2057x2057 nécessite 33Mo), bien que l'algorithme n'utilise que des noyaux calculs de coût n (une multiplication vecteur-vecteur et des produits scalaires dans la version ligne). La version ligne de la factorisation de Cholesky a été préférée parce qu'étant à base de produits scalaires qui sont plus efficaces que les saxpy lorsqu'ils sont distribués (cf 8.3.2.2 et 8.3.2.3).

On a vu en 7.3.1 qu'il n'est pas possible de paralléliser les calculs de la boucle sur k. Aussi, avec la méthode de parallélisation adoptée (de type "maître-esclave"), l'implantation de l'algorithme consiste à exécuter le corps de l'algorithme séquentiellement et à distribuer les noyaux de calculs de grain n sur l'arbre ternaire (qui correspondent à {1}, {2} et {3}). On obtient alors l'algorithme (entre parenthèses figurent les tailles des opérandes) :

```
pour k variant de 0 jusqu'à n-1
   produit vecteur-vecteur distribué de d par L[k,.] (taille k)
   affectation scalaire
   produit sclaire distribué de L[k,.] par aux (taille k)
   pour i variant de k+1 jusqu'à n-1
      affectation scalaire
      produit sclaire distribué de L[i,.] par aux (taille k)
      division scalaire
```

Le temps d'exécution de cet algorithme est nécessairement très élevé : les performances du produit scalaire et du produit vecteur-vecteur sont trés faibles (0.18 Mflops pour le produit scalaire et 0.12 pour le saxpy).
On obtient un temps d'exécution de 5 heures 52 minutes, ce qui correspond à des performances de 0.138 Mflops. A titre de comparaison, la factorisation d'une matrice 512x512 sur un seul Transputer (de la carte Volvox 4/V pour pouvoir disposer de 2 Mo de mémoire) prend 141.77 secondes, ce qui donne des performances de 0.37 Mflops. Le recours au parallélisme dans des conditions défavorables (les communications sont très nombreuses et mal équilibrées avec les calculs locaux) diminue donc les performances de la machine de moitié.

8.3.3.2. Les résolutions de systèmes triangulaires

Nous examinons une parallélisation de la résolution de systèmes triangulaires. Considérons pour cela un système triangulaire (supérieur par exemple) de taille nxn réparti à raison de m lignes par processeur. L'implantation choisie pour la parallélisation est l'algorithme classique qui consiste à l'étape i à calculer les m nouvelles solutions stockées sur le processeur i, puis celui-ci diffuse ces solutions à tous les autres processeurs qui en parallèle mettent à jour leurs solutions, puis l'on passe à l'étape i+1. Cet algorithme est illustré sur la figure ci-dessous.

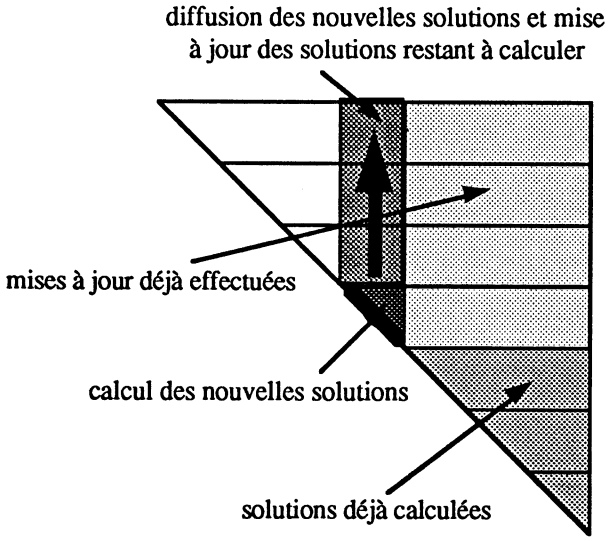


Figure 106 : résolution triangulaire en parallèle

Le point faible de cet algorithme, avec la topologie choisie pour le MégaNode, réside dans les phases de communications : seuls les solutions x_n à x_{n-17} sont stockées sur la racine de l'arbre. Dans tous les autres cas, il faut tout d'abord remonter les nouvelles solutions qui viennent d'être calculées jusqu'à la racine avant de les diffuser.
Notons toutefois que si l'on pouvait s'assurer un synchronisme de départ de tous les

processeurs (ce qui n'est pas le cas sur le MégaNode), une version asynchrone de cet algorithme pourrait être écrite : les solutions nouvellement calculées seraient simultanément diffusées dans la branche de l'arbre d'où elles sont issues tandis qu'elles seraient remontées jusqu'à la racine pour diffusion dans le reste de l'arbre. Cet algorithme présente toutefois un inconvénient : la plupart des processeurs sont des feuilles terminales de l'arbre, donc le gain dû à la diffusion immédiate des solutions dans la branche de l'arbre d'où elles sont issues serait assez faible (il faudrait quand même une phase de remontée partielle) pour tous ces processeurs.

L'algorithme de résolution triangulaire est donné ci-dessous pour un processeur quelconque. Au niveau de la remontée des solutions nouvellement calculées, tous les processeurs envoient une pseudo-solution (pour éviter un blocage des communications en réception).

```
/* cas special : racine commence les calculs */
si proc=racine et début_résolution
   calcul des nouvelles solutions
sinon
   répéter
      /* diffusion des nouvelles solutions */
      si proc=racine
         stockage des nouvelles solutions
         envoi aux fils des nouvelles solutions
      sinon si proc=milieu
         réception du père des nouvelles solutions
         envoi aux fils des nouvelles solutions
      sinon
         réception du père des nouvelles solutions
      /* mise à jour des solutions */
      si solutions(proc) restent à calculer
         mise à jour de ses solutions
      si solutions(proc) doivent être calculées
         calculs des nouvelles solutions
      /* remontée des nouvelles solutions */
      si proc=feuille terminale
         envoi au père de la nouvelle solution
      sinon si proc≠feuille terminale et proc≠racine
         réception des fils des nouvelles solutions
         si 4 pseudo-solutions
            envoi au père d'une pseudo-solution
         sinon
            envoi au père de la nouvelle solution
      sinon
```

```
        réception des fils de la nouvelle solution
        stockage de la nouvelle solution
    jusqu'à fin_résolution
```

Le temps d'exécution de cet algorithme est de 4.21 secondes pour un système 2057x2057. Le coût d'une résolution triangulaire étant de $n^2+O(n)$ opérations arithmétiques, on obtient des performances de 1.001 Mflops.
Ces performances sont supérieures à celles de la factorisation de Cholesky malgré le nombre élevé de communications parce que les envois de messages sont de taille très inférieure (17 réels) et que les calculs sont de l'ordre du carré par rapport à la taille des messages.

8.3.3.3. Le gradient conjugué distribué

L'algorithme du gradient conjugué a été implanté avec un préconditionnement diagonal (pour des raisons de simplicité, le préconditionnement définitif de la méthode du gradient conjugué préconditionné pour sytèmes linéaires consécutifs est exposé en 2.1). L'algorithme est donné ci-dessous en terme de noyaux d'exécution (les commentaires renvoient à l'algorithme complet, décrit en 5.1.2) :

```
/* initialisations */
affectation vectorielle
affectation vectorielle
/* résolution du système Mz=g */
division vectorielle
/* initialisations : fin */
affectation vectorielle
répéter
   produit matrice-vecteur
   /* mise à jour de x et g */
   produit scalaire
   produit scalaire
   produit scalaire
   division scalaire
   saxpy-like
   saxpy-like
   /* résolution du système Mz=g */
   division vectorielle
   /* mise à jour de d*/
   affectation scalaire
   division scalaire
```

```
    saxpy-like
jusqu'à convergence
```

Les sections 8.3.2.2 et 8.3.2.3 ont montré que la granularité du MégaNode était de n^2 et que par conséquent l'on n'avait pas intérêt à distribuer des produits scalaires et des saxpy. Par conséquent l'algorithme du gradient conjugué préconditionné a été implanté en séquentiel sur le processeur "maître" (qui est la racine de l'arbre ternaire) et seul le produit matrice-vecteur a été distribué aux processeurs.

Le temps d'exécution obtenu pour un système de taille 2057x2057 est de 0.51 secondes par itération, le gradient conjugué ayant un coût de $2\alpha n^2 + O(n)$ (α étant le nombre d'itérations), on obtient donc des performances de 8.10 Mflops.

8.3.3.4. Remarques

Le temps d'exécution obtenu pour la factorisation de la méthode de Cholesky amoindrit énormément l'intérêt de la méthode du gradient conjugué préconditionné pour systèmes linéaires successifs sur cette machine. En effet à moins d'avoir un pas de réinitialisation très élevé (ce qui n'est pas en général le cas), le coût lié à la factorisation de Cholesky sera beaucoup plus élevé que celui d'une série de résolutions par le gradient conjugué préconditionné (même avec un préconditionnement moins performant).
D'autres part, la méthode du gradient conjugué préconditionné pour systèmes linéaires successifs nécessite deux résolutions de systèmes triangulaires au sein de chaque itération du gradient conjugué préconditionné. Les performances des résolutions triangulaires étant beaucoup plus faibles que celles du gradient conjugué préconditionné, les résolutions par le gradient conjugué préconditionné par la factorisation de Cholesky (utilisées dans l'algorithme du gradient conjugué préconditionné pour systèmes linéaires successifs) seraient fortement diminuées.

Le grain du MégaNode n'étant que de n^2, on se trouve confronté aux mêmes problèmes que dans le cas du FPS T40 lorsqu'on veut implanter les méthodes directes présentées au chapitre 2. A savoir, un excès de communications devant les calculs distribués qui conduit à des performances très faibles (analogues à celles obtenues pour Cholesky).

Conclusion

Nous nous sommes intéressés tout au long de cette thèse à l'apport potentiel du parallélisme à la résolution d'un problème de répartitions de charges dans les réseaux électriques. Ce problème mathématique se ramène à la résolution d'une série de systèmes linéaires successifs dont les matrices (qui modélisent les configurations du réseau électrique et dont on cherche la "meilleure") sont symétriques mais n'ont aucune structure particulière ; les modifications permettant de passer d'une matrice à la suivante sont à priori quelconques.

Nous avons tout d'abord présenté les méthodes directes classiques de résolution de systèmes linéaires successifs . Nous avons ensuite proposé une méthode originale basée sur des résolutions par le gradient conjugué avec un préconditionnement spécifique : une factorisation de Cholesky, certes coûteuse, mais qui n'est pas effectuée à chaque résolution. Un aspect intéressant de cette méthode est qu'elle s'applique à des matrices modifiées de façon quelconque et non pas par des transformations de rang donné. De plus, les expérimentations réalisées sur une machine séquentielle classique montrent qu'elle est plus intéressante que des résolutions par la méthode Bennett (et d'autant plus que le rang des transformations est important). Il est en outre possible de jouer sur le pas des réinitialisations, ce qui permet d'adapter l'algorithme à l'architecture parallèle utilisée.

Avant d'étudier l'impact des architectures parallèles sur ces différents algorithmes, nous avons réalisé une modélisation informatique des machines MIMD à mémoire distribuée (décrite au chapitre 4 dans le cas particulier du FPS T40). Ce travail permet à l'utilisateur de se poser les questions nécessaires à la réalisation d'une implantation efficace sur ce type de machine : quelle est l'influence d'éventuelles unités de calculs vectoriel, quelle est la granularité de la machine ... La modélisation des différents types de communications utilisées lors de l'implantation d'algorithmes d'algèbre linéaire nous a conduit à proposer un algorithme de diffusion original basé sur une stratégie partiellement asynchrone.

Les expérimentations effectuées sur une première architecture MIMD à mémoire distribuée, le FPS T40, mettent en évidence le grain important de cette machine (il n'est intéressant de distribuer des noyaux de calcul que si leur "taille" est de n^2 au moins, si n est la taille du problème), ainsi que la disproportion existant sur cette machine entre communications et calculs (en partie due aux unités de calcul vectoriel).
La stratégie distribuée du gradient conjugué préconditionné mise en œuvre sur cette machine ne permet pas d'atteindre des performances élevées : une grande partie du code est exécutée séquentiellement (en fait, seul le produit matrice-vecteur est distribué sur tous

les processeurs). C'est pourquoi plusieurs implantations locales où chaque processeur résout un système ont été expérimentées. Le point essentiel est alors d'équilibrer la charge des processeurs : une factorisation par la méthode de Cholesky est beaucoup plus coûteuse que toute résolution par la méthode du gradient conjugué préconditionné. L'équilibrage de la charge de travail des différents processeurs est rendu possible par la souplesse de la méthode du gradient conjugué pour les systèmes successifs qui permet de jouer sur le pas des réinitialisations. Il est possible d'obtenir des performances élevées avec ce type de méthode, à condition d'utiliser des méthodes sophistiquées de communication (par exemple en diffusant simultanénement plusieurs préconditionnements à l'aide d'arbres de recouvrement disjoints).

Des expérimentations ont ensuite été menées sur une architecture SIMD massivement parallèle : la Connection Machine. Le caractère synchrone et la faible puissance de calcul des processeurs élémentaires de cette machine influence profondément le choix des algorithmes. Les méthodes de résolution de systèmes successifs implantées sur la Connection Machine font clairement apparaître que les algorithmes qui permettent un usage intensif de tous les processeurs virtuels sont les seuls qui permettent d'obtenir de réelles performances. Par exemple, les algorithmes de résolution de systèmes triangulaires (même les versions les plus sophistiquées) qui n'ont qu'un faible taux d'emploi des processeurs élémentaires, malgré des communications peu pénalisantes, donnent de très faibles performances.
Parmi les méthodes directes décrites au chapitre 2, toutes celles qui ne permettent pas une utilisation massive des processeurs élémentaires sont donc mal adaptées à ce type de machine. Une réponse efficace à la résolution de systèmes linéaires successifs a été proposée : la méthode de Woodbury permet l'utilisation de tous les processeurs élémentaires (dans le cas de modifications de rang 1 uniquement) et l'implantation réalisée permet de plus d'effectuer les calculs les plus coûteux de façon purement locale (sans communication).

La dernière machine sur laquelle ont été effectuées des expérimentations est le MégaNode : un ordinateur MIMD à mémoire distribuée et à architecture entièrement reconfigurable. Deux raisons ont motivé ce choix : cette machine est plus équilibrée que le FPS T40 (les calculs sont plus lents : il n'y a pas d'unité de calcul vectoriel, les communications sont plus rapides) et la topologie des processeurs est reconfigurable (ce qui permet d'utiliser un réseau mieux adapté aux applications).
Le choix de stratégies type "maître-esclave" (des échanges complets de données entre tous les processeurs s'avérant trop coûteux) et les spécifications techniques du Transputer (doté de 4 canaux physiques de communication) ont motivé le choix d'une configuration des processeurs en arbre ternaire. La détermination expérimentale de la granularité du MégaNode dans cette configuration a permis de conclure que seuls les noyaux de calcul d'un coût en n^2 ou plus devaient être distribués sur tous les processeurs (le grain du MégaNode est donc comparable à celui du FPS T40).

Le nombre élevé de processeurs (121 Transputers T800) a permis l'expérimentation d'une stratégie distribuée de la méthode du gradient conjugué pour les systèmes successifs, sur des problèmes de taille élevée (de l'ordre de 2000x2000). Les résolutions par le gradient conjugué préconditionné permettent d'atteindre des performances élevées pour ce type de machine car les noyaux de calcul proportionnels à n peuvent être effectués localement et le produit matrice-vecteur est distribué sur tous les processeurs. Cependant les réinitialisations (qui sont des factorisations de Cholesky) ne permettent d'atteindre que de très faibles performances : les noyaux d'exécution de l'ordre de n (produits scalaires ou saxpy suivant la version de l'algorithme) utilisent les éléments de la matrice du système, et cette matrice est nécessairement distribuée sur les processeurs, par conséquent ces noyaux de calcul d'un grain trop fin doivent être distribués à tous les processeurs.

Les différentes méthodes de résolution de systèmes linéaires successifs expérimentées sur les différentes machines font apparaître l'influence très forte des différents types d'architectures sur les algorithmes. Des méthodes efficaces en séquentiel peuvent ne pas être intéressantes sur certains types de machines parallèles : citons par exemple dans le cas de la Connection Machine les méthodes directes qui nécessitent des résolutions de systèmes triangulaires (comme la méthode de Bennett pour la mise à jour des facteurs de Cholesky). Par contre certains algorithmes, peut-être moins efficaces en séquentiel, s'avèrent intéressants sur des machines parallèles.
Cependant, pour chaque expérimentation réalisée, nous avons montré qu'il est possible d'implanter des méthodes adaptées à l'architecture permettant une résolution efficace du problème de répartition de charge dans les réseaux électriques.

Toutes les expérimentations réalisées au long de cette thèse montrent que l'utilisation de machines parallèles nécessite encore actuellement un grand investissement en temps de la part de l'utilisateur. Il faut tout d'abord acquérir des outils théoriques propres au parallélisme (tout au moins dans leur utilisation) comme les algorithmes de communication sophistiqués ou la notion de précédence de tâche qui est sous-jacente à toute implantation sur machine parallèle. Il est ensuite nécessaire d'assimiler des notions plus techniques comme l'utilisation des langages de bas niveau ou la programmation explicite des processeurs vectoriels qui sont tous deux difficiles à mettre en œuvre sur des ordinateurs parallèles, mais qui sont encore les seuls à offrir des performances élevées.
Implanter un algorithme sur une machine parallèle nécessite encore d'adapter celui-ci à l'architecture cible et pour cette raison la plupart des implantations sont réalisées sous formes d'applications rigides qui se contentent souvent d'appeler des bibliothèques numériques adaptées à telle ou telle machine (c'est l'approche qui semble prévaloire aujourd'hui).
Il est cependant raisonnable d'espérer pour bientôt une utilisation plus souple et plus générale des machines parallèles, on assiste en effet actuellement à des progrès intéressants comme les études réalisées sur la notion de mémoire virtuelle distribuée ou encore l'arrivée de machines reconfigurables.

Références bibliographiques

[Agra] D. P.Agrawal : *Advanced Computer Architectures*, Tutorial IEEE, North Holland, 1986.

[Alma] G. S. Almassi : *Overview of Parallel Processing*, Parallel Computing n°2, 1985.

[Anon] Anonyme : *Etude de répartition à "courant continu", méthode matricielle de résolution*, note technique EDF-DER HR 4.366 JA-PA/SR, 1960.

[AsWi] K. J. Astrom & B. Wittenmark : *Computer Controlled System-Theory and Design*, Prentice Hall, 1984.

[AxBa] O. Axelsson & Barker : *Finite Element Solution of Boundary Value Problems*, Academic Press, 1984.

[BFAl] P. Bouvry, H. Frydlender, J. Prevost, J.-L. Roch, A. Touzene & G. Villard : *Rapport technique, manuel du méganode*, rapport de recherche LMC (Grenoble), 1991.

[Blan] G. Blanchon : *Calculs de répartition*, note technique EDF-DER HR 33.0815 GB/AC, 1986.

[BlCT] J.-Y. Blanc, P. Comon & D. Trystram : *Using Preconditioned Conjugate Gradient for Solving Consecutive Linear Systems*, Communication in Numerical Methods, vol. 6, 1990.

[BlTr] J.-Y. Blanc & D. Trystram : *Implementation of Parallel Numerical Routines using Broadcast Communication Schemes*, CONPAR 90, Zürich, 1990.

[BlTV] J.-Y. Blanc, D. Trystram et G. Villard : *Desynchronized Communication Schemes for Distributed-Memory Architectures*, Distributed Memory Computing Conference 5, Charleston (Caroline du Sud), 1990.

[BMAl] J.-Y. Blanc, P. Michallon, B. Tourancheau, D. Trystram & F. Vincent : *Analysis of Basic Numerical Routines on the FPS T20 Hypercube*, European Symposium on Hypercubes, Rennes, 1989.

[Calv] C. H. Calvin : *Etude et implantation d'algorithmes de produit matrice-vecteur creux sur la Connection Machine*, rapport de DEA, Université J. Fourier (Grenoble), 1991.

[Ciof] J. M. Cioffi : *Limited Precision Effects in Adaptive Filtering*, IEEE Transactions on Circuits and Systems, special issue on adaptive systems, vol. 34, n°7, 1987.

[ClMP] G. A. Clark, S. K. Mitra & S. R. Parker : *Block Implementation of Adaptive Digital Filters*, IEE Transactions on Circuits an Systems, vol. 28, n°6, 1981.

[Colo] L. Colombet : *Résolutions successives de systèmes linéaires sur Connection Machine*, rapport de DEA, Université Joseph Fourier, Grenoble, 1990.

[CoTr] P. Comon & D. Trystram : *An Incomplete Factorization Algorithm for Adaptive Filtering*, Signal Processing, vol. 13, n°4, 1987.

[Dahl] E. H. Dahl : *Mapping and Compiled Communication on the Connection Machine System*, Distributed Memory Computing Conference 5, Charleston (Caroline du Sud), 1990.

[DBMS] J.J. Dongarra, J. R. Bunch, C. B. Moler & G. W. Steward : *LINPACK User's Guide*, SIAM Publications, Philadelphia, 1979.

[DeTo] F. Desprez & B. Tourancheau : *Modélisation des performances de communication sur le tnode avec le Logical System Transputer Toolset*, La lettre du Transputer, septembre 1990.

[DTW1] D. Delesalle, D. Trystram & D. Wenzek : *Tout ce que vous voulez savoir sur la Connection Machine (sans oser le demander)*, rapport de recherche LMC, Grenoble, 1990.

[DTW2] D. Delesalle, D. Trystram & D. Wenzek : *Optimal Communication Schemes on an SIMD Distributed-Memory Hypercube*, rapport de recherche LMC, Grenoble, 1991.

[Dong] J. J. Dongarra : *The LINPACK Benchmark: An Explanation*, Speed-up, vol. 1, n°1, 1987.

[DuRB] L. Dubost, J. Ryckbosch & Y. Berthet : *Spécifications générales du logiciel Paratop de recherches de parades topologiques dans le SNC,* note technique EDF-DER HR 36.2083 LB/JR/YB/AC, 1989.

[Ea] K. Ea : *Enchainement de calculs de répartition dans l'approximation du courant continu, survol d'un ensemble de méthodes*, note technique EDF-DER HR 341054.

[FlPo] R. Fletcher & R. J. Powell : *On the Modification of LDL^t Factorisation,* Mathematics of Computation, vol. 28, 1974.

[Frai] P. Fraignaud : *Performance Analysis of Broadcasting in Hypercubes*, Hypercube and Distributed Computers Conference Proceedings, Rennes, Elsevier Science Publisher, 1989.

[Fryd] H. Frydlender : *Parallélisation de l'algorithme de rétropropagation du gradient récursif*, rapport technique LMc RR854M, 1991.

[GGMS] P. E. Gill, G. H. Golub, W. Murray & M. A. Saunders : *Methods for Modifying Matrix Factorizations*, Mathematics of Computation, vol. 28, n°126, 1974.

[GoMe] G. H. Golub & G. Meurant : *Résolution numérique des grands systèmes linéaires*, collection CEA-EDF, Eyrolles, 1981.

[GoVL] G. H. Golub & C. Van Loan : *Matrix Computations*, John Hopkins University Press, 1983.

[Gree] A. Greenbaum : *Comparison of Splittings Used with the Conjugate Gradient Algorithm*, Numerische Math., vol. 33 , 1979.

[Gust] J. Gustafson : *Reevaluating Amdahl's Law*, Communications of the ACM, vol. 31, n°5, 1988.

[Hayk] S. Haykin : *Adaptive Filter Theory*, Prentice Hall, 1986.

[Hers] A. Herscovici : *Introduction aux grands ordinateurs scientifiques*, Eyrolles, 1986.

[HeSt] M. Hestenes & E. Stiefele : *Methods of Conjugate Gradient for Solving Linear Systems*, Journal Res. Nat. Bur. Stan., vol. 49, 1952.

[HoJe] R. W. Hockney & C. R. Jessope : *Parallel Computing II*, Adam Hilger, 1988.

[HwBr] K. Hwang & F. Briggs : *Computer Architecture and Parallel Processing*, Mac Graw Hill, 1984.

[Inm1] Inmos : *The Transputer Data Book*, Inmos Limited Editeurs, 1989.

[Inm2] Inmos : *Introducing the INMOS T9000 Transputer*, La Lettre du Transputer, numéro hors-série spécial T9000, 1991.

[JoHo] S. L. Johnsson & C. T. Ho : *Distributed Routing Algorithms for Broadcasting and Personnalized Communication in Hypercubes*, IEEE Transaction on Computers, vol. 38, n°9, 1989.

[Kaza] E. Kazamarande : *Etude de la parallélisation d'algorithmes de mise à jour des facteurs d'une matrice*, rapport de DEA, Université Joseph Fourier, Grenoble, 1989.

[LaTr] P. Laurent-Gengoux & D. Trystram : *Resolution of Large Sparse Linear Systems*, Rapport de recherche, Ecole Centrale de Paris, 1986.

[Mant] T. A. Manteuffel : *An Incomplete Factorization Technique for Positive Definite Linear Systems*, Math. Comp., vol. 34, 1980.

[MeMT] M. D. Mesarovic, D. Macko & Y. Takahara : *Structuring of Multilevel Systems*, procedures IFAC on multivariable control systems, Düsseldorf, 1968.

[MeVa] J. A. Meijerink & H. A. Van Der Vorst : *An Iterative Solution Method for Linear Systems of which the Coefficient Matrix is a Symmetric M-matrix*, Journal of Computational Physics, vol. 31, 1977.

[MiSa] A. Mitchel & J. Sanders : *Dateline 1995!*, Parallelogram, n° 32, 1990.

[Modi] J. J. Modi : *Updating Cholesky Factors in Parallel*, rapport technique, Université de Cambridge, 1988.

[MuWa] T.Muntean & P. Waille : *L'architecture des machines SuperNode*, La Lettre du Transputer, n° 7, 1990.

[Nico] D. A. Nicoles : *Esprit Project 1085 Reconfigurable Transputer Processor Architecture*, Conpar 88 (Additional Papers), Manchester, 1988.

[OrVo] J. M. Ortega & R. G. Voigt : *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM review 27, n°2, 1985.

[Peri] Perihelion Software : *The Helios Operating System*, Prentice Hall, 1989.

[QuRo] Patrice Quinton & Yves Robert : *Algorithmes et architectures systoliques*, Masson, 1989.

[Robe] Y. Robert : *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, Manchester University Press, 1990.

[RoTV] Y. Robert, B. Tourancheau & G. Villard : *Data Allocation Strategies for the Gauss and Jordan Algorithms on a Ring of Processors*, Information Processing Letters 31, 1989.

[Ryc1] J. Ryckbosch : *Sur les calculs de répartition dans les réseaux électriques à topologie variable*, note technique EDF-DER HR 34.2004 JR/AC, 1988.

[SaS1] Y. Saad & M. H. Schultz : *Topological Properties of Hypercubes*, IEEE Transactions on Computers, vol. 37, n°7, 1988.

[SaS2] Y. Saad & M. H. Schultz : *Data Communication in Parallel Architectures*, Parallel Computing, n°11, 1989.

[StWa] Q. F. Stout & B. Wagar : *Intensive Hypercube Communication, Prearranged Communication in Link-Bound Machines*, Journal of Parallel and Distributed Computing, n°10, 1990.

[Telm] Telmat Informatique : *T.NodeOverview*, documentation technique Telmat Informatique, 1989.

[Try1] D. Trystram : *Expérimentation d'algorithmes de préconditionnement pour des grands systèmes linéaires creux*, thèse, Université de grenoble, 1984.

[Try2] D. Trystram : *Communications et réseaux*, séminaire cmap (Grenoble), 1991.

[TrVi] D. Trystram & F. Vincent : *Programmation avancée du Transputer, architectures et mécanismes*, La Lettre du Transputer, avril 1990.

[TuRo] L. W. Tucker & G. G. Robertson : *Architecture and Applications of the Connection Machine*, Computer, 1988.