NUMERO D'ORDRE :

ANNEE : 1998

USTL

LIFL Laboratoire d'Informatique Fondamentale de Lille

CNRS

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

J.Gaber

Plongements et manipulations d'arbres dans les architectures distribuées

Thèse soutenue le 15 Janvier 1998, devant la commission d'examen :

Directeur de thèse :	B. TOURSEL	LIFL, Université de Lille
Rapporteurs :	A. FREVILLE	Université de Valenciennes
	H. GUYENNET	Université de Franche-Comté

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

Table des matières

1 Introduction **13**

1.1 L'évolution et les limitations des systèmes monoprocesseurs 13

1.2 Les machines parallèles 15

1.2.1 Architectures MIMD 16

1.2.1.1 Architectures MIMD à mémoire partagée 16

1.2.1.2 Architectures MIMD à mémoire distribuée 17

1.2.1.2.1 Le fonctionnement standard 17

1.2.1.2.2 La tendance actuelle 18

1.2.2 Architectures SIMD 22

1.3 Les réseaux d'interconnexion 27

1.3.1 Les topologies dynamiques 28

1.3.2 Les topologies statiques 29

1.3.2.1 L'hypercube 30

1.3.2.2 La grille 31

1.3.2.3 L'arbre binaire complet 31

1.3.2.4 La grille d'arbres 34

1.4 Conclusion 36

2 Le plongement de graphes **37**

3 Plongement statique dans une grille **47**

3.1 Introduction 47

3.2 Le H-arbre 50

3.3 Définition du plongement $X(h_0, \ell, L)$ 52

3.3.1 La contraction 52

3.3.2 Première approche de plongement 53
3.3.3 Le plongement amélioré 54
3.4 Analyse de l'expansion 55
3.5 Analyse de la dilatation 56
3.6 Analyse de la congestion 58
3.7 Analyse de la charge 59
3.8 Bilan de l'analyse 60
3.9 L'algorithme distribué du plongement 60
3.10 Le temps de propagation 61
3.11 Parallélisme et performances 62
3.12 Les améliorations du plongement en X 64
3.13 Conclusion 66

4 Les mouvements de données dans une architecture distribuée et synchrone 71

4.1 Les opérations de mouvement de données dans une architecture SIMD . . . 72
4.2 Description des algorithmes des procédures du mouvement de données . . . 78
4.2.1 Notations 78
4.2.2 Descriptions des algorithmes 79
4.2.2.1 La procédure Rank 80
4.2.2.2 La procédure Concentrate 81
4.2.2.3 La procédure Distribute 82
4.2.2.4 La procédure Generalize 83
4.2.2.5 La procédure Sort 83
4.3 Complexité des opérations de mouvement de données 84
4.4 Extensions 84
4.5 $O(\frac{M}{N}T)$ et $O(\frac{M}{N}+T)$ 86
4.6 Conclusion 87

5 Plongement d'arbres quelconques et dynamiques 89

5.1 Introduction 89
5.2 Optimisation des mouvements de données 94
5.2.1 Les collisions dans une grille 96
5.3 Les stratégies de plongement 102

5.4 Application de l'optimisation au problème de plongement 103
5.4.1 Les opérations de communication inter-sommets 104
5.4.2 analyse du plongement irrégulier 104
5.4.2.1 Cas des arbres complets 105
5.4.2.2 Cas des arbres quelconques 108
5.4.2.3 Le test de collision 110
5.4.2.4 La migration 111
5.4.2.5 Extensions 112
5.4.2.6 Expérimentations 112
5.5 Conclusion 112

6 Plongement aléatoire d'arbres quelconques et dynamiques 117
6.1 Introduction 117
6.2 Ce qui a été fait 118
6.3 Définition du modèle de plongement dynamique 120
6.4 Description de l'algorithme de plongement aléatoire 123
6.5 Le résultat principal 124
6.6 Matrices stochastiques 125
6.6.1 Valeurs propres et vecteurs propres 125
6.6.2 Calcul de la limite d'une suite de matrices stochastiques 127
6.7 Distribution de probabilité 128
6.8 Chaînes de Markov finies 129
6.9 Analyse du plongement 131
6.10 Exemples 136
6.11 L'utilisation des opérations de mouvement de données 140
6.11.1 Expérimentations sur la MasPar 141
6.12 Expérimentations en utilisant PVM et PM^2 141
6.13 Conclusion et perspectives 143

7 Perspectives 145

Résumé

1. Domaine abordé

Cadre général: La mise en œuvre d'algorithmes manipulant des structures irrégulières et dynamiques sur des machines parallèles à mémoire distribuée.

Cadre précis: Etude d'algorithmes de plongement d'arbres sur architectures parallèles et distribuées.

2. Problématique traitée

Exécuter un algorithme parallèle sur un réseau de processeurs, émuler une architecture par une autre, représenter des structures de données ou réaliser physiquement en VLSI un réseau logique sont des problèmes qui sont modélisés, dans la littérature, par des problèmes de plongement de graphes (graph embedding, ou mapping problems). En effet, un algorithme parallèle peut être représenté par un graphe dans lequel les sommets représentent les tâches et les arêtes les communications nécessaires aux calculs. De la même manière, une architecture parallèle distribuée peut être représentée par un graphe.
D'une manière générale, le plongement d'un graphe G dans un graphe hôte H consiste à projeter les sommets de G sur les sommets de H et à associer à chaque arête de G un chemin dans H. Un plongement efficace de G dans H doit être effectué de telle sorte que:

- les calculs soient répartis équitablement sur les sommets du graphe hôte G (i.e., la charge soit faible).

- les communications soient locales (i.e., l'étirement maximun des arêtes de G sur des chemins dans H soit faible).

L'objectif de la thèse est de proposer des algorithmes de plongements efficaces d'arbres quelconques et dynamiques dans les architectures communément utilisées telles que les grilles, les hypercubes et les réseaux de stations de travail.

3. Contribution de la thèse

La thèse est composée des trois parties suivantes :

- **Plongement d'arbres binaires complets et statiques** : Dans la première partie, nous présentons des algorithmes de plongement d'arbres binaires dans une grille bidimensionnelle. Ces méthodes de plongements concernent les arbres binaires, complets et statiques, c'est à dire dont on connaît a priori la taille et la forme. L'approche de plongement que nous décrivons dans cette partie est une amélioration de la technique de plongement en H. En effet, cette dernière soufre d'un defaut majeur : la taille de la grille nécessaire pour plonger un arbre donné est asymptotiquement deux fois la taille de cet arbre. La statégie de plongement proposée améliore le plongement en H en termes d'expansion (i.e., le rapport entre la taille de la grille et la taille de l'arbre que l'on plonge) et de la dilation (i.e, l'étirement maximun des arêtes de l'arbre sur des chemins dans la grille) [72, 71, 70]. Ces algorithmes ont été repris et améliorés dans le cadre de plongement d'arbres quaternaires par A.Bellaachia et M.Jiber de l'Université Alakhawayn du Maroc [101, 102, 73].

- **Plongement d'arbres quelconques et dynamiques** : Dans la deuxième partie de la thèse, nous présentons des algorithmes de plongement d'arbres quelconques et dynamiques. En d'autres termes, le plongement d'un arbre est effectué sans connaître a priori ni sa forme ni sa taille. Notre démarche dans cette partie est différente de l'approche classique de plongement. En effet, dans cette dernière, un plongement est mis en œuvre en étirant des arêtes de l'arbre sur des chemins dans la grille cible, comme cela a été fait dans la partie précédente. Pour exploiter le plongement, en acheminant des messages le long de ces liens, on se heurte à des problèmes de congestion. Nous avons donc transformé le problème de plongement en un problème d'optimisation d'un routage existant [66, 68, 69]. Ce dernier résout des problèmes aléatoires de mouvement de données dans une architecture parallèle distribuée et synchrone.

- **Plongement d'arbres quelconques et dynamiques** : Contrairement aux algorithmes de plongement déterministes définis dans les deux parties précédentes, les algorithmes de plongement proposés dans cette partie sont probabilistes. En effet, dans un contexte distribué, l'utilisation de choix aléatoire mais locaux pour placer les sommets d'un arbre quelconques et dynamiques est une bonne alternative à l'utilisation d'un système de contrôle centralisé. En général, la mise en œuvre de mécanismes dynamiques de distribution, en reposant sur un état global, est difficile voire impossible à obtenir.

  Afin d'analyser le comportement aléatoire de ces algorithmes, nous avons utilisé des outils mathématiques issus de la théorie des chaînes de Markov et des résultats d'analyse numérique sur les itérations des matrices carrées. Cette analyse nous a permis la mise en œuvre d'algorithmes de plongement d'arbres quelconques et dynamiques dans des topologies arbitraires [67].

4. Titre de la thèse

plongements et manipulations d'arbres dans les architectures parallèles et distribuées.

5. Organisation de la thèse

5.1. Présentation générale

La thèse est organisée en quatre parties. La première partie est une introduction générale. Elle définit le cadre de la thèse en présentant les principales architectures parallèles existantes. La mise en avant de la diversité et des limitations des réseaux d'interconnexion permet d'appuyer les motivations et les objectifs de la thèse, et plus précisément le problème de la mise en œuvre d'algorithmes manipulant des structures arborescentes dans les architectures distribuées. A l'intérieur de cette partie, les définitions générales et les terminologies relatives à la notion de plongement de graphes sont également présentées tout en montrant les raisons pour lesquelles son étude est si utile.

La deuxième partie est consacrée à la présentation d'une approche de plongement d'arbres binaires complets dans les grilles bidimensionnelles, et en l'évaluant en termes de mesures de plongement de graphes présentées dans la première partie.

Dans la troisième partie, la notion de mouvement de données dans une architecture distribuée et synchrone est abordée. Nous analysons des opérations de mouvement de données existantes et largement employées, et nous montrons que ces dernières peuvent être utilisées d'une manière optimisée sous certaines conditions. En ramenant le problème de plongement à un problème de mouvement de données (i.e., un problème de routage), une approche de plongement d'arbres quelconques et dynamiques est présentée.

Dans la quatrième partie, la méthode de plongement présentée est une méthode probabiliste qui permet de plonger des arbres irréguliers et dynamiques dans une topologie arbitraire. Les résultats obtenus dans cette partie sont démontrés en utilisant une analyse basée sur la théorie des chaînes de Markov.

Afin de valider les résultats présentés dans le rapport, des évaluation ont été faites sur la machine massivement parallèle MasPar, et en utilisant les environnements de programmation distribuée PVM et PM^2 sur un réseau de stations de travail.

Enfin, des perspectives du travail sont présentées à la fin du rapport.

5.2. Plan : vue globale

Partie I : Introduction générale
1. Présentation des principales architectures parallèles
2. Le plongement de graphe : définitions et terminologies

Partie II : Plongement statique

3. Plongement statique d'arbres binaires complets dans une grille bidimensionnelle

Partie III : Plongement dynamique et déterministe

4. Les mouvements de données dans une architecture distribuée et synchrone

5. Première approche, détérministe, de plongement d'arbres quelconques et dynamiques dans les architectures communément utilisées telles que les grilles et les hypercubes : le plongement irrégulier

Partie III : Plongement dynamique et aléatoire

6. Deuxième approche, probabiliste, de plongement d'arbres quelconques et dynamiques dans une topologie arbitraire : le plongement régulier

7. Conclusions et perspectives

6. Contenu des chapitres

1 Présentation des principales architectures parallèles

Ce chapitre décrit les architectures parallèles existantes, caractérisées principalement par la topologie du réseau de communication. Les tendances actuelles sont brièvement présentées.

2 Le plongement de graphe : définitions et terminologies

Ce chapitre présente les définitions générales et les terminologies relatives à la notion de plongement de graphes. Les motivations pour lesquelles l'étude de plongements est si utile sont illustrées à l'aide de trois exemples de plongements.

3 Plongement statique d'arbres binaires complet dans une grille

Ce chapitre décrit une approche proposée pour plonger un arbre binaire complet dans une grille bidimensionnelle en exploitant sa structure récursive. Une évaluation analytique basée sur les mesures de plongement de graphe présentées dans le chapitre précédent est décrite. Une validation effectuée sur la machine massivement parrallèle est également présentée.

A.Bellaachia et M.Jiber [3] ont repris et amélioré cette approche dans le cadre de plongement d'arbres quaternaires. Nous avons d'ailleurs réecrit ce chapitre à l'issu de ces améliorations. Les résultats de leurs évaluations sur la machine MasPar sur un exemple d'application de type diviser-pour-régner, l'algorithme de tri par fusion (merge-sort), sont présentées à la fin du chapitre.

4 Les mouvements de données dans une architecture distribuée et synchrone

Dans ce chapitre, on décline les opérations de mouvement de données élaborées par D.Nassimi et S.Sahni. Ces opérations couvrent un ensemble d'outils permettant de copier et de déplacer des données à travers les processeurs d'une architecture à mémoire distribuée et synchrone. Cela permet aux données dispersées dans la mémoire distribuée de la machine de se trouver à la bonne place au bon moment. Plus précisément, elles permettent de résoudre les problèmes aléatoires de mouvement de donnés (i.e., des problèmes généraux de routage). L'implantation de ces opérations s'effectue à l'aide de quelques procédures de base bien définies. Une description détaillée de ces procédures est effectuée dans ce chapitre. Nous ferons référence au contenu de ce dernier dans la suite.

5 Première approche, détérministe, de plongement d'arbres quelconques et dynamiques

Les opérations de mouvements de données sont largement utilisées dans littérature pour la mise en œuvre de nombreux algorithmes (par exemple, certains algorithmes sur les graphes qui nécessitent des communications générales, à la différence des communications point à point). Mais ces opérations sont souvent utilisées comme un outil monolithique. En analysant les algorithmes des procédures qui mettent en œuvre ces opérations et leurs preuves de correction, nous montrons que pour certains problèmes de routage, on peut réduire la liste d'appels aux procédures de base pour résoudre ces problèmes. Ce qui permet ainsi de réduire le coût d'exécution de leurs réalisations.

Nous définissons ensuite dans ce chapitre, deux stratégies de plongement d'arbres dynamiques et quelconques. La première stratégie de plongement est une méthode déterministe et qui autorise le mécanisme de migration de processus (i.e., les sommets de l'arbre). La deuxième méthode de plongement est une méthode probabiliste, et, est présentée dans le chapitre suivant. Puisque le problème de communications entre les sommets des arbres plongés est un problème de mouvement de données, les deux méthodes utilisent les opérations déjà présentées. Nous analysons ainsi l'impact de ces plongements sur l'utilisation des procédures de mouvement de données pour implanter les opérations de base de manipulation d'un arbre. Plus précisément, on établit les contraintes à vérifier par un algorithme de plongement afin de bénéficier d'une utilisation optimisée de ces procédures.

6 Deuxième approche, probabiliste, de plongement d'arbres quelconques et dynamiques

Nous commençons par décrire dans ce chapitre des algorithmes aléatoire de plongement dynamique d'arbres binaires dues à Bhatt et Cai (qui ont proposé initialement ce problème) et à Leighton. Nous définissons ensuite un modèle de plongement dynamique qui permet de construire des algorithmes probabilistes de plongement et nous montrons comment ces derniers peuvent plonger en direct un graphe quelconque de M sommets dans une topologie quelconque de N sommets. Afin d'analyser le comportement aléatoire de ces algorithmes, nous utilisons des outils mathématiques issus de la théorie des chaînes de Markov et des

résultats d'analyse numérique sur les itérations des matrices carrées. Une présentation de ces deux derniers est faite dans ce chapitre. Le résultat de l'analyse montre que l'approche de plongement permet de plonger un arbre quelconques de M sommets dans une topologie arbitraire de N sommets de telle sorte qu'un sommet du réseau ne représente qu'au plus $O(M/N+\varepsilon)$ sommets de l'arbre, où ε dépend de la manière avec laquelle l'arbre est projeté dans le réseau pour effectuer le plongement.

7 Conclusions et perspectives

Ce petit chapitre présente les perspectives de la thèse. Durant notre travail, nous avons entrepris des contacts avec F.T.Leighton et F.Chung. F.T.Leighton a révisé le travail concernant l'optimisation des mouvement de données et les deux approches, déterministe et probabiliste, de plongement d'arbres quelconques et dynamiques. Les remarques qu'il a émise nous ont aidé à mieux présenter l'approche probabiliste. F.Chung a révisé le travail concernant l'utilisation de résulats issus de théorie des chaînes de Markov. Nous avons inclus ces remarques dans ce chapitre.

Partie 1

Chapitre 1

Introduction

Ce chapitre présente une étude des différents types d'architectures parallèles avec comme critère de comparaison leur réseau d'interconnexion et leurs modes de fonctionnement. Ce chapitre ne se veut pas une présentation exhaustive des architectures parallèles. Pour une approche plus complète du domaine, citons les ouvrages de références d'architectures et d'algorithmiques parallèles tels que [81, 62, 4, 75].

Nous consacrerons à la fin de ce chapitre une section entière à l'étude de plongement de graphes dans les architectures parallèles à mémoire distribuée.

1.1 L'évolution et les limitations des systèmes monoprocesseurs

La machine von Neuman au niveau architectural se compose d'une unité centrale de traitement (UC), d'une mémoire principale (MP) stockant les instructions et les données et d'une connexion liant ces deux composants. A chaque cycle d'exécution, l'unité de traitement cherche une donnée ou une instruction à exécuter depuis la mémoire. Par conséquent, quelle que soit la vitesse du processeur, les performances globales de la machine sont limitées par la restriction physique due à cette interconnexion processeur-mémoire qualifiée de goulot d'etranglement (the von Neuman bottleneck) [133]. En effet plus les processeurs sont puissants et rapides, plus la mémoire doit l'être pour alimenter les processeurs en données dans un flux optimisant leurs traitements. L'évolution des performances des processeurs s'accompagne cependant d'un écart croissant entre l'unité centrale UC et la mémoire principale MP, les temps d'accès à la mémoire évoluant plus lentement que les performances des processeurs [52]) [1]. L'évolution des performances de ces derniers ne va pas sans induire d'autres problèmes essentiellement liés à la vitesse et aux limitations d'intégration (packaging). Une étude plus approfondie du domaine est effectuée dans [40, 52].

L'architecture des ordinateurs a bénéficié en effet sur le plan technologique, de la très rapide

1. Les temps d'accès et de cycle (respectivement 60 et 100 ns actuellement) des composants DRAM qui constituent la mémoire principale ne diminuent que de quelques pourcents chaque année, alors que les performances des CPUs doublent tous les 18-24 mois [125]

évolution de la microélectronique (densité d'intégration, augmentation de la fréquence). Cette évolution est encore prévisible pour quelques années encore, mais il est inévitable que les limites pratiques ou fondamentales finissent par ralentir cette croissance. La question de la contribution possible de technologies autres que le CMOS dans la conception des ordinateurs a été d'ailleurs soulevée [26, 99, 13].

Parallèlement à l'évolution technologique des processeurs, des progrès sur le plan architectural ont été réalisés. En effet, des améliorations sur ce plan ont été ainsi apportées pour pallier le problème lié aux débit mémoires: l'utilisation systématique de caches, des mots mémoires plus larges (1 mot MP = k mots UC), l'entrelacement des adresses mémoire (m bancs mémoires), etc.

Les progrès sur le plan architectural ont conduit également par exemple, à la spécification de processeurs de types RISC[2] dont la commercialisation se généralise (exemple des processeurs SPARC, T9000, Mips R8000, Dec 21064, T.I. Supersparc, PowerPC [124, 125, 126])[3]. On parle également d'architecture *pipeline*, *Superpipeline*, *Superscalaire* et de *VLIW* dont une étude peut être trouvée dans [62, 125].

Il est important de noter que ces améliorations architecturales font intervenir en fait un certain type de fonctionement en parallèle. Dans la machine pipeline, par exemple, le processeur se compose de plusieurs étages (les unités de traitement et de contrôle y sont découpées en étages). Les instructions décomposées en n phases élémentaires de longueur 1 cycle, sont exécutées en parallèle de la manière suivante. La phase k de l'instruction i est exécutée en même temps que la phase $k-1$ de l'instruction $i+1$. on parle alors de parallélisme temporel. Dans les machines Superscalaires, le mécanisme d'exécution en parallèle consiste à exécuter plusieurs instructions à chaque cycle grâce à des unités fonctionnelles travaillant en parallèle. On parle alors de parallélisme spatial. A titre d'exemple, les processeurs DEC21164 et le MIPS R8000 peuvent lancer jusqu'à quatre intructions par cycle, le Power2 peut traiter jusqu'à six instructions par cycle [125]. Notons aussi que ces améliorations architecturales introduites dans les systèmes monoprocesseurs font appel à un parallélisme intra-processeur. Plus précisément, il s'agit d'un parallélisme implicite qui n'est pas explicitement voulu par le programmeur mais qui est dû aux particularités architecturales apportées au hardware.

Cependant, malgré l'accroissement de la puissance de calcul individuelle des processeurs, les performances des systèmes monoprocesseurs restent toujours en deçà de la puissance de calcul requise actuellement dans de nombreux domaines (synthèse d'images, intelligence artificielle, calcul scientifique, etc...). Pour faire face aux possibilités limitées de ces machines séquentielles, qui ne sont pas dues uniquement aux limitations technologiques d'intégration, mais aussi aux limitations des interconnexions processeur-mémoire, on a recours au parallélisme inter-processeur. L'exploitation de ce parallélisme reste vraisemblablement l'approche idéale pour une véritable augmentation des performances des calculateurs.

L'évolution de la technologie et les améliorations architecturales (densité d'intégration, architectures de type RISC) sont à la source du gain d'intérêt pour les architectures parallèles: les processeurs sont bon marché, facilement conditionnables et consomment

2. N.d.T. de la terminologie anglo-saxonne Reduced Instruction Set Control, à l'opposé de CISC, Complex Instruction Set Control

3. Les processeurs Intel Pentium ont une architecture CISC

peu d'énergie. Il est économiquement faisable de concevoir une architecture contenant un grand nombre de processeurs.

Le parallélisme peut être vu sous deux angles, l'un matériel et l'autre logiciel. Ainsi on parle des architectures parallèles et de la programmation ou algorithmique parallèle. Du point de vue matériel, cela veut dire fournir des unités de traitement pouvant être actives simultanément. Du point de vue logiciel, l'exploitation du parallélisme s'établit par le découpage d'un programme en sous-tâches pouvant être exécutées en parallèle (i.e., le parallélisme de contrôle), ou par l'identification des traitements sur des données qui peuvent être effectuées en parallèle (i.e., le parallélisme de données). Notons que ceci laisse entendre également qu'il faut disposer de moyens de communication (matériels et logicils) entre les traitements parallèles coopérants.

1.2 Les machines parallèles

Les machines parallèles sont composées, d'une manière générale, de plusieurs noeuds de calcul reliés entre eux par un réseau d'interconnexion.

Etant donné la diversité des architectures parallèles non conventionnelles (massivement parallèles, cellulaires, systoliques, neuronales...), il est difficile de donner une définition précise de ce que doit être un noeud de calcul, il peut être un processeur dont la structure est proche de celle d'un ordinateur séquentiel ou un processeur spécifique ou encore un processeur beaucoup plus simplifié. On en déduit aussi que rien n'exclut la superposition des différents types de parallélisme évoqués dans la section 1.1.

Néanmoins, on peut noter trois principales caracteristiques d'une machine parallèle :

- le nombre de processeurs et leurs architectures (nature, taille, vitesse, ...).
- l'organisation de la mémoire : si les processeurs possèdent leur propre mémoire privée ou partagent un même espace mémoire (mémoire partagée, distribuée ou virtuellement partagée).
- la topologie du réseau d'interconnexion : comment les processeurs sont inter-connectés, ou sont connéctés à la mémoire.

Une machine dotée de n processeurs doit être idéalement n fois plus rapide qu'une machine dotée d'un seul processeur (en supposant que les processeurs sont tous identiques). Ce n'est pas le cas en pratique pour des raisons se situant à tous les niveaux de l'exploitation : l'application ou le type d'application considérée, et l'outil informatique dans ses aspects matériels et logiciels. Parmis ces raisons on peut citer :

- pour une taille de problème donnée, une loi fondamentale, la loi d'Amdahl dit que l'accélération sur toute machine parallèle est limitée par la fraction séquentielle du code. En d'autres termes, la puissance crête ne peut être atteinte que sur des programmes complètement parallélisés.

- à la difficulté de conception d'un logiciel exploitant pleinement les ressources matérielles des machines (les paralléliseurs automatiques, etc), s'ajoutent des problèmes tels que le placement de tâches. En effet, pour utiliser pleinement la puissance de calcul des machines parallèles, il faut que le travail à réaliser soit partagé le plus équitablement possible entre les différents processeurs.
- les communications constituent les aspects les plus limitatifs des machines parallèles. En effet, il ne suffit pas de disposer de porocesseurs qui calculent vite, mais que ces processeurs puissent communiquer entre eux aussi rapidement qu'ils opèrent et accèdent à leurs propres données. Le réseau d'interconnexion est le facteur déterminant de la performance d'une machine parallèle. Nous consacrerons la section 1.3 à leurs études.

La grande diversité des machines parallèles a donné lieu, à plusieurs taxonomies parmi lesquelles on peut citer la plus couramment utilisée, celle de Flynn [45]. Cette classification prend en compte deux critèrte: le type du flot d'instructions et le type du flot des données traitées par les processeurs. Elle ne permet pas cependant de tenir compte de nombreux facteurs tels que l'organisation de la mémoire ou encore de la granularité des processeurs [30].

Nous décrivons dans ce qui suit très brièvement les trois principaux groupes de cette classification et qui sont les suivants:

1. SIMD (Single Instruction Multiple Data Stream)
2. MIMD (Multiple Instruction Multiple Data Stream) avec mémoire partagée
3. MIMD avec mémoire distribuée

1.2.1 Architectures MIMD

Dans les machines de type MIMD, plusieurs instructions pouvant être différentes sont exécutées parallèlement par plusieurs processeurs sur les données propres à chaque processeur. Les machines de ce type n'ont pas d'horloge globale et opèrent donc en mode asynchrone. On distingue deux types d'architectures parallèles: les architectures à mémoire commune, *parfois appelés les multiprocesseurs*, où tous les processeurs peuvent accéder à la totalité de la mémoire, et les architectures à mémoire distribuée, *parfois appelés les multicalculateurs*, où chaque processeur dispose localement d'une mémoire privée. La distinction entre mémoire distribuée et mémoire partagée recouvre la distinction entre machines moyennement à massivement parallèles et machines à faible nombre de processeurs.

1.2.1.1 Architectures MIMD à mémoire partagée

Dans les architectures à mémoire partagée (voir la figure 1.1), tous les processeurs partagent un même espace de données global. Dans ce cas, tous les processeurs peuvent accéder à n'importe quel banc mémoire (la mémoire est multi-bancs) via un réseau d'interconnexion, chaque processeur ayant le même temps d'accès en l'absence de conflits. Ce

modèle de multiprocesseur est appelé *modèle UMA* pour Uniform Memory Access [62]. La mémoire joue un double rôle: d'une part, le rôle classique de la mémoire contenant les instructions et les données propres au programme en cours d'exécution sur chaque processeur, et, d'autre part, un rôle de communication entre les processeurs par le biais des variables partagées. Cette mémoire commune est donc très sollicitée (quasiment à chaque instruction) et son accès forme un goulot d'étranglement. En pratique, chaque processeur a sa propre antémémoire (une mémoire cache) et/ou une mémoire locale qui peut être privée ou non (suivant que les autres processeurs peuvent y accéder ou pas). L'utilisation du cache réduit la fréquence d'accès mémoire par les processeurs, mais pose des problèmes de cohérence des données (entre les copies d'une même donnée présentes dans différents caches). Des protocoles doivent être mis en oeuvre pour maintenir la cohérence des mémoires caches locales ce qui augmente la complexité du matériel et du logiciel. Ce partage de la mémoire qui créee le phénomène de contention dû aux conflits d'accès limite le nombre de processeurs qu'une telle machine peut comporter.

Diverses topologies de réseaux ont été mises en oeuvre pour connecter les processeurs aux différents bancs mémoires: l'utilisation d'un bus est conceptuellement la topologie la plus simple. Cette architecture est communément appelée architecture SMP (Symmetric Multiprocessor). Pour des systèmes utilisant un petit nombre de processeurs (moins de 16), il est possible d'utiliser le crossbar qui est le réseau reconfigurable idéal (parce que permettant toutes les permutations des entrées vers les sorties). Les machines Cray (à l'exception de la T3D) font partie de de la famille de machines utilisant un crossbar pour les accès mémoires.

L'utilisation des réseaux à commutation de circuits entre les processeurs et les mémoires assurent à la fois un débit important et un temps de latence minimal, mais ces réseaux deviennent vite inefficaces quand le nombre de processeurs atteint quelques centaines d'unités, que ce soit pour le Crossbar (augmentation quadratique), ou pour des réseaux multiétages (complexité de la commande [35], [87], [8]). Une autre approche utilisé dans la machine M3S[4] [120] [92] consiste à relier les processeurs aux mémoires par des liens séries à haut débit.

1.2.1.2 Architectures MIMD à mémoire distribuée

1.2.1.2.1 Le fonctionnement standard

Dans les architectures parallèles MIMD à mémoire distribuée, chaque Unité de Traitement (UT) a sa propre mémoire locale et privée, et exécute les instructions indépendamment des autres Unités de traitements (voir la figure 1.2). Les UT n'ont pas accès à la mémoire des autres unités de traitements. Lorsqu'une unité de traitement a besoin d'accéder aux données d'une autre UT, une requête explicite doit être émise à travers le réseau d'interconnexion ; on parle alors de communication par messages par opposion à la communication par variables partagées dans les architectures à mémoire commune. Le mécanisme de communication par échanges de messages, généralement moins efficace que le partage de mémoire, se traduit par un grain de parallélisme moins fin pour les architectures à mémoire distribuée; le **grain** de parallélisme d'une application est défini ici comme étant

4. Multiprocesseur à Mémoire Multiport Série

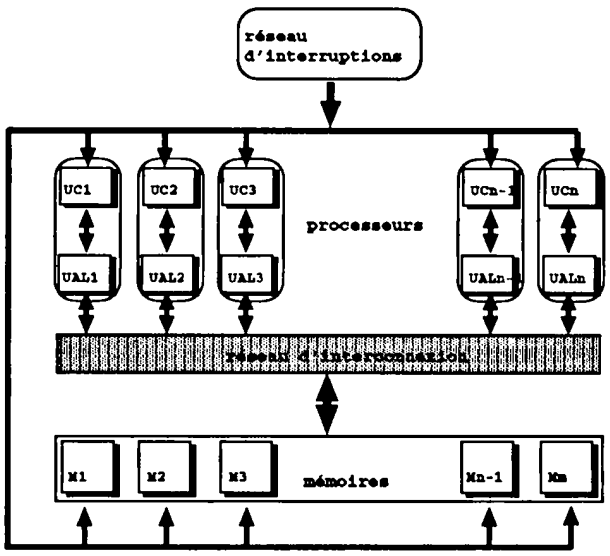

FIG. 1.1 - *modèle d'exécution MIMD à mémoire partagée*

le rapport entre la charge en calcul et la charge en communication. Notons cependant que les échanges d'informations les plus fréquents, acquisition de l'instruction et manipulation de données locales, s'effectuent entre mémoire et processeur qui résident sur le même noeud et n'utilisent pas le réseau pour communiquer entre eux. Le délai introduit par la transmission à travers le réseau ne pénalisera que les échanges entre processus parallèles.

Plusieurs réseaux d'interconnexion peuvent être utilisés pour ce type d'architecture: anneaux, étoiles, arbres, hypercubes, les cycles connectés en cube (cube-connected cycles CCC), les réseaux papillon (butterfly) et omega etc. Cependant, si l'hypercube a été adopté dans les premières machines MIMD à mémoire distribuée, actuellement, nous observons un regain d'intérêt pour les réseaux meshes 2D (on montre dans [32] que pour une même largeur de bissection[5], les réseaux de faible dimensions comme les tores possèdent des temps de latence[6] plus faibles et des débits plus élevés que les réseaux de dimensions élevées, en l'occurence les hypercubes). A titre d'exemples de ce type d'architecture, citons les systèmes à base de Transputers, le Cosmic Cube du Caltech [123] et ses successeurs industriels, les hypercubes commerciaux dont les plus connus sont les iPSC/1 et /2 et la machine Paragon [64].

1.2.1.2.2 La tendance actuelle

Actuellement, de nouveaux modes de fonctionnement sur ce type d'architecture commencent à être généralisés; sur une architecture MIMD à passage de message, le parallélisme de données peut être implanté selon le mode SPMD. ce mode de fonctionnement se distingue par le modèle de programmation se rapprochant des architectures SIMD. Une autre tendance consiste à contourner la limitation des multiprocesseurs par des architectures caractérisés essentiellement par une gestion de la mémoire similaire à celle

5. Le nombre minimum de liens qui, une fois retirés, séparerait le réseau en deux sous réseaux
6. Temps requis pour l'envoi d'un message de la source vers la destination

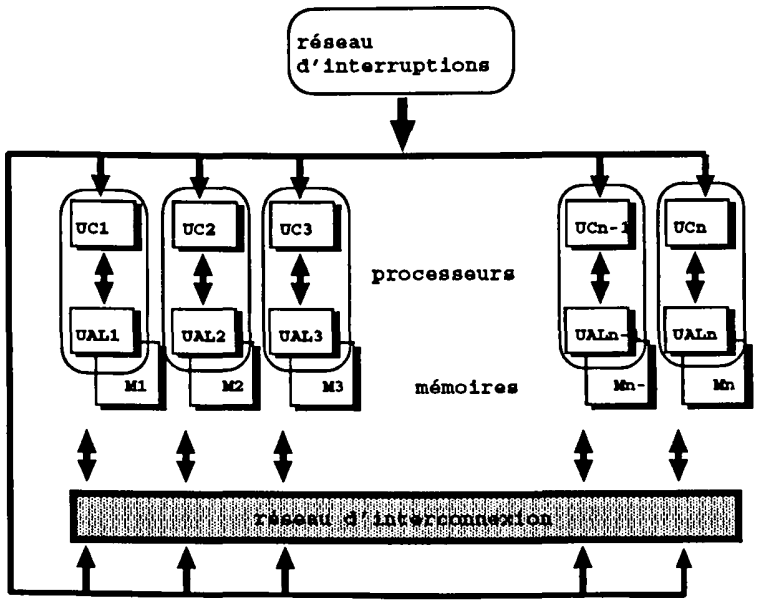

FIG. 1.2 - *modèle d'exécution MIMD à mémoire distribuée*

des architectures à mémoire partagée. On voit également apparaître la notion de machine virtuelle.

Le fonctionnement en mode SPMD : Dans le mode SPMD (Single Program Multiple Data), un même programme s'exécute sur tous les noeuds, mais les instructions qui s'exécutent à un instant donné sur des processeurs différents peuvent être différentes (par exemple en cas de branchement conditionnel). Les machines ayant ce mode de fonctionnement possèdent généralement un ou plusieurs dispositifs de synchronisation qui peuvent être implémentés de manières différentes. Ainsi, dans la CM5, la notion de *barrière de synchronisation* a été introduite pour synchroniser les processeurs par l'utilisation d'un réseau spécial, le réseau de contrôle (différent du réseau de données) [62], le Cray T3D fournit des primitives matérielles plus élaborés [109] dont le *Barrier Synchronization* et le *Eureka Synchronization* pour les recherches parallèles. Il faut aussi noter que contrairement au SIMD qui impose un mode de fonctionnement synchrone et un réseau capable de fournir un flux d'instructions tous les cycle de calcul, la synchronisation dans le mode SPMD est lié au programme à exécuter et non pas à l'architecture. La machine Paragon [64] par exemple contrairement aux deux machines citées ci-dessus utilise uniquement des mécanismes logiciels fonctionnant à partir du réseau de type grille 2D. On parle aussi de plus en plus de programmes SPMD sur des réseaux de stations de travail.

la mémoire virtuellement partagée : On distingue généralement deux modèles de multiprocesseurs à mémoire physiquement distribuée et virtuellement partagée [62] : le modèle NUMA (Non Uniform Memory Access) pour architecture à accès mémoire non uniforme et le modèle COMA (Cache-Only Memory Architecture) pour architecture n'utilisant que des caches. Ces deux modèles ont pour but de contourner

les limitations des modèles UMA en réduisant les accès au réseau d'interconnexion (processeur-bancs mémoires).

- le modèle NUMA : La mémoire partagée est physiquement distribuée en mémoires locales. Les temps d'accès varient suivant que la donnée est présente dans la mémoire locale ou dans une autre mémoire. On appelle ainsi *mémoire home* d'une donnée, le module mémoire qui la contient. Les performances sont donc étroitement liées à la répartition des données dans les différentes mémoires locales. Dans ce modèle, l'architecture peut être hiérarchisée en connectant via un réseau, des multiprocesseurs (clusters) UMA ou NUMA, et ces clusters peuvent être également connectés à des modules de mémoire globale partagée [62, 41]. Le Cray T3D [109] et la machine DASH [88] sont des exemples de ce modèle de multiprocesseurs.
- le modèle COMA : Ce modèle est un cas particulier du modèle NUMA. La mémoire distribuée est convertie en caches (d'où la terminologie *ALLCACHE memory* dans la machine KSR1). Ces caches forment un espace d'adressage global. La machine KSR1 [62, 79] est un exemple de machine COMA fonctionnant avec un réseau hiérarchique à base d'anneaux.

Des environnements pour le parallélisme :

Des systèmes à échanges de messages comme PVM (Parallel Virtual Machine), MPI (Message Passing Interface) ou encore LANDA (Local Area Network for Distributed Applications) ont été developpés pour offrir des environnements facilitant le développement d'applications parallèles sur les machines MIMD à mémoire distribuée [114, 110]. Il s'agit non seulement des machines parallèles mais aussi de réseau hétérogène de stations de travail. En effet, ces outils basés sur des bibliothèques de communications et de manipulations de messages (par des échanges de messages explicites), masquent la répartition géographique des sites et rendent ainsi transparentes les communications à travers le réseau. Le réseau hétérogène de machines est dès lors utilisé comme une seule machine parallèle. Notons que ces logiciels peuvent être vus comme une amélioration du modèle client-serveur [24].

La programmation distribuée s'appuie classiquement sur le modèle de processus (ou tâches) communiquants. Un processus dans cet approche est qualifié de *processus lourd* en raison de son contexte jugé chargé(segment de données, code, pile, fichiers ouverts, ...). Cette approche est remise en question particulièrement pour la programmation des applications irrégulières à cause du coût élevé de manipulation et gestion de processus sur une même machine ou un même processeur (au niveau d'une même mémoire réside plusieurs processus, sachant qu'ils sont exécutés de façon imbriquée dans le temps par une même unité centrale, le passage d'un processus à un autre étant le fait des interruptions).

Un nouveau type de programmation appelé communément la programmation multithreadée est proposé. Un thread est une partie du code d'un processus qualifié de *processus léger* à cause de son contexte réduit. Il s'exécute en concurrence avec les autres threads du processus et partage avec eux l'espace mémoire du processus et à travers laquelle ils communiquent. Le coût de manipulation et gestion de processus

sur une même machine ou un même processeur (changement de contexte, ...) est donc réduit. Des environnements à base de threads comme PM^2 [115] a été implanté au dessus de la couche PVM.

Les réseaux locaux :

Pour faire du traitement distribué à faible coût, une solution consiste à utiliser un réseau de stations UNIX. Ce type de réseau permet des performances essentiellement pour des applications de granularité importante, c'est-à-dire nécessitant des communications moins fréquentes, chaque communication pouvant porter sur des tailles de données assez importantes. Ces réseaux ont naturellement des performances en deça des réseaux d'interconnexion des machines parallèles : temps de latences élevés et débits plus faibles. Le réseau n'étant pas très performant, le mode SIMD n'est évidemment pas envisageable parce qu'il faudrait pouvoir distribuer un flux d'instruction à chaque cycle processeur, ces dernier utilisant le même réseau pour leurs communications. Les modèles de programmation possibles sont donc le MIMD et le SPMD, la synchronisation logicielle (sous PVM, Express, etc.) étant effectuée en utilisant le réseau par échange de message.

Il existe plusieurs type de réseaux locaux pour l'interconnexion de l'ensemble de stations, les débits allant de quelques Mbps[7] à quelques Gbps[8]. Un réseau Ethernet a un débit de 10 Mbps, le Fast Ethernet peut atteindre 100 Mbps, le réseau *FDDI*[9] possédant un débit de 100 Mbps, le réseau à commutation très rapide *GIGAswitch à base de FDDI*, est un crossbar pouvant offrir un débit crête de 3,6 Gbps [14], et le réseau ATM[10] offre un débit de 155 Mbps. Du côté des machines, citons comme exemple le parc de stations *Alpha* de DEC (*Alpha AXP Farms*). Les stations sont connectées via un réseau GIGAswitch. Notons par ailleurs que le terme architecture *cluster* désigne le système à mémoire distibuée qui peut être doté de machines ou processeurs différents, le tout est relié à la façon d'un réseau local ou via un concentrateur ou commutateur (un anneau FDDI, un commutateur ATM ou un bus propriétaire). Notons aussi le projet MPC de BULL et LIP6 qui offre une plateforme matérielle qui permet de créer des réseaux de PCs sur bus PCI, avec une latence de $4\mu s$, et des mécanismes d'écriture directe dans la mémoire du voisin (Remote Write) ce qui permet de réduire la latence logicielle.

Outre les soucis causés par la structure matérielle des machines MIMD il y a, bien sûr ceux posés par leur programmation. Ce qui caractérise fondamentalement le modèle MIMD, c'est que le parallèlisme inhérent à chaque application doit être explicité par le programmeur. Pour parvenir à expliciter le parallélisme, il faut donc commencer par détecter les différentes tâches, programmer leur exécution par un langage capable de produire des blocs d'instructions séparés et possédant des moyens de contrôle pour gérer les communications entre ces blocs et enfin placer explicitement les tâches sur les processeurs. Il faut en plus gérer le temps et les problèmes de synchronisation puisque le modèle MIMD fonctionne d'une manière asynchrone d'où le développement des modèles mathématiques (comme les

7. Méga bits par seconde : 10^6 bps
8. Gigabits par seconde : 10^9 bps
9. Fiber Distributed Data Interface
10. Asynchronous Transfer Mode

réseaux de Petri), qui permettent de prévoir, dans une certaine mesure, le déroulement des opérations, et donc de déceler d'éventuels problèmes temporels.

Face à ces problèmes, on a développé aussi l'idée de l'extraction automatique du parallélisme. L'idée de base est la suivante : en examinant la façon dont se déroule l'exécution d'un programme séquentiel, on constate que certaines suites d'instructions pourraient être exécutées en parallèle. Une architecture matérielle et logicielle capable de décomposer le programme tout en respectant les dépendances temporelles entre les différentes suites d'instructions est alors proposée. Le langage Prolog a fait l'objet de plusieurs recherches à fin de mettre à profit l'exploitation automatique du parallélisme dans ce langage [21]. L'idée de l'extraction automatique du parallélisme a produit aussi plusieurs familles de machines, comme les machines Data Flow et les machines à réductions ou machines symboliques. Le traitement symbolique se distingue du traitement numérique, notamment par la complexité de la représentation des données. Le traiement numérique (ou le calcul numérique) fait souvent intervenir, par exemple pour des calculs matriciels, une structuration régulière des données. En revanche, dans le traitement symboliques les structures des données se caractérisent par leurs irrégularités. On ne manipule pas seulement des nombres mais aussi des symboles et des liens entre ses symboles. Les machines symboliques s'appuient sur deux théories mathématiques équivalentes de formalisation du calcul : le lambda-clacul (à partir duquel est issue la programmation fonctionnelle) et la logique combinatoire. En lambda-calcul, pour exécuter un programme, on applique une suite de substitutions et de réductions, on remplaçant dans une expression donnée une opération par son résultat après avoir renplacé les arguments formels par les arguments réels, autrement dit appliqué une fonction sur un argument. En logique combinatoire, la notion d'argument n'existe pas, le programmes avec ses données est transformé dans une expression à calculer à base de combinateurs avant de procéder à des réductions successives suivant des régles de réduction.

Les langages utilisés dans les architectures à mémoire comune sont dits *orienté procédure* (PASCAL Modula) [9], l'interaction des processus est basée sur des variables partagées. Ces langages fournissent des mécanismes élémentaires d'exclusion mutuelle pour résoudre le problème de l'accès concurrent à des variables partagées de la mémoire commune. Les langages utilisés dans les architectures à mémoire distribuée sont du type *orienté messages*, qui contrairement aux langages orientés procédures, n'ont pas de variables partagées. Le programmeur construit son programme comme un ensemble de processus s'exécutant en parallèle sur les différents processeurs et utilise des primitives d'échanges de messages pour définir et gérer les communications et la synchronisation inter-processus.

1.2.2 Architectures SIMD

Les machines parallèles SIMD sont des machines synchrones et se caractérisent par un nombre élevé d'unités de traitement. Chaque unité de traitement exécute la même instruction au même instant. En d'autres termes, il y a seulement un seul compteur ordinal contrôlant l'exécution du programme. Chaque unité de traitement est réduite à une Unité Arithmétique et Logique (UAL) et une mémoire locale (ML). Les couples (UAL,ML) sont généralement appelés processeurs élémentaires (PE). Ainsi, un seul processeur (appelée

contrôleur ou séquenceur) génère le flux de contrôle et envoie les instructions à tout les PE. Chacun de ces PE contient un bit de contexte qui indique s'il est actif ou non pour l'exécution d'une instruction. Lors de l'exécution d'une instruction mémoire à mémoire par exemple, l'opération peut avoir lieu sur tous les PEs, mais seuls les PEs actifs rangent le résultat en mémoire [94].

Les processeurs élémentaires sont reliés entre eux par un réseau d'interconnexion. Les réseaux les plus utilisés sont les hypercubes et les réseaux de type grille 2D. L'ensemble des PEs est relié à l'ordinateur frontal qui joue le rôle du "host" à travers le contrôleur. La machine hôte qui peut ne pas être unique, supporte le développement et la compilation des programmes; c'est typiquement une station de travail. Le contrôleur communique les instructions et les données de la machine hôte vers tous les PEs et rassemble les résultats provenant des PEs. Il a accès au réseau d'interconnexion, aux canaux d'entrée/sortie et à sa propre mémoire. Le contrôleur et la machine hôte peuvent être regroupés en une même unité. Sur la connection machine, le contrôleur est nommé séquenceur, sur la machine MasPar, il est nommé ACU (Array Control Unit).

Plusieurs machines de ce type ont existé telles que ILLIAC IV, STARAN, DAP, Zhephir [119]. On inclut aussi dans cette catégorie les réseaux systoliques [80]. Parmi les machines les plus representatrices de cette catégorie aujourd'hui, citons la Connection Machine CM-2 [55] les machines MasPar MP-1 et MP-2 [108].

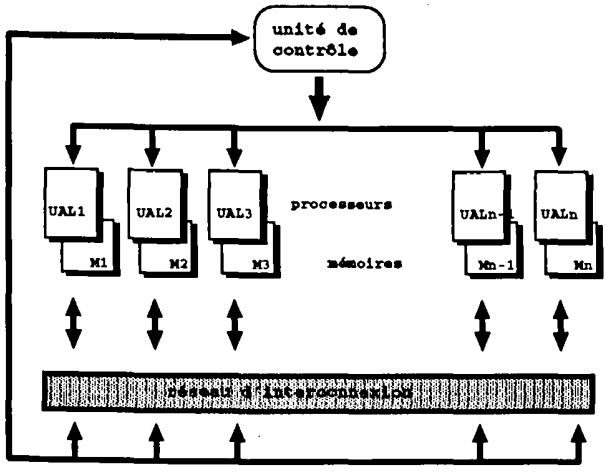

FIG. 1.3 - *Le modèle d'exécution SIMD*

La puissance de calcul des machines SIMD vient du fait que chaque instruction y est exécutée simultanément sur un grand nombres de données. Contrairement à ce qui se passe dans les machines MIMD, on n'a pas vraiment affaire à des tâches, mais plutôt au cas où des mêmes instructions s'exécutent sur plusieurs données différentes en parallèle.

Des applications informatiques concernant des domaines importants comme le traitement d'images, le traitement du signal, l'aéronautique, etc, se prêtent bien au traitement en mode SIMD. En effet, ce qui caractérise ces applications, c'est l'organisation spatiale des

données à traiter (par exemple, on représente souvent une image par un tableau de pixels, on définit ainsi une correspondance spatiale entre les éléments du tableau affectés aux PEs et les points de l'image réelle).

En fait, l'algorithmique parallèle SIMD consiste tout d'abord à représenter les données par des structures mathématiques simples et répétitives, puis à trouver l'algorithme qui applique les instructions souhaitées sur chaque élément de ces structures. Et ce faisant, on explicite le parallélisme en établissant une correspondance (transposition) spatiale des données et de l'architecture SIMD. Il est également souhaitable que la structure matérielle de la machine reflète aussi fidèlement que possible la structure de données. Notons que pour obtenir une bonne adéquation entre une application à mettre en œuvre et une structure matérielle, deux stratégies opposées sont possibles.

- Construire une structure matérielle ad hoc
- Contraindre le problème à se résoudre dans la structure disponible

Dans le premier cas, on construit une machine de même géométrie que les données à traiter. Par exemple, pour le calcul de transformées de Fourier rapides FFT, algorithme très utilisé dans les applications de reconnaissance des formes, on utilise souvent une topologie dite en papillon directement câblée [119, 81]. Dans ce cas, le champ d'application de la machine est évidemment restreint puisque spécialisé. A l'inverse, on peut prendre une machine SIMD existante ayant une géométrie matérielle fixée. Dans la plupart des cas, cette géométrie est un hypercube ou une grille ce qui signifie notamment que les unités de traitement ne sont connectés qu'à un nombre limité de leurs voisins. Mais il faut alors contraindre le problème à se résoudre dans la topologie disponible et c'est là qu'intervient l'algorithmique parallèle.

On va évoquer brièvement deux machines SIMD dites massivement parallèles : la machine cellulaire Connection Machine CM-2 [55] et la machine MasPar MP-1 [108].

CM2 : La machine commercialisée, Connection machine est plus au moins proche de ce que doit être un ordinateur cellulaire[11]. le processeur de base est très simple, il comprend une unité de calcul ne pouvant traiter qu'un bit et il est doté d'une mémoire locale. Cette machine était destinée au départ à l'interrogation déductive de grosses bases de connaissanse. Sa configuration maximale comporte 65 536 processeurs élémentaires interconnecté par un réseau d'une structure hiérarchique à deux niveaux. Au premier niveau, un hypercube de dimension 12 soit 4096 noeuds reliés entre eux par des liens permettant un haut débit d'information. Au deuxième niveau, chaque noeud contient 16 processeurs disposés en grille. Chaque nœud comprend également quatre boîtiers mémoire qui fournissent 4 kbits de mémoire à chacun des processeurs, une unité de contôle qui décode les instructions et commande les 16 processeurs de

11. L'architecture des machines cellulaires s'inspire de l'architecture SIMD et de celle de systèmes dits automates cellulaires. Chaque processeur-cellule effectue un traitement simple et synchrone sur les données venant de ses voisins immédiats. La plupart du temps, les cellules qui constituent ces machines sont organisées en grille. La puissance de l'architecture provient effectivement du fait que l'instruction est exécutée au même instant par plusieurs dizaines de milliers de cellules. Ces architectures étaient destinées initialement aux traitement rapide d'images numérisées ou aux recherches en intelligence artificielle.

facon synchrone et un router qui gère les communications entre processeurs s'effectuant à travers les connexions de l'hypercube.

MasPar : La machine MasPar est une machine massivement parallèle composée d'un frontal du type station de travail et d'une unité du parallélisme de donnée (DPU). On accède au DPU par l'intermédiaire du frontal. Le DPU est composé lui même de deux parties: un contrôleur ou séquenceur nommé ACU (Array Control Unit) qui est un processeur style RISC et le tableau de processeurs élémentaires PE. Le processeur scalaire ACU cherche et decode les instructions, calcule les adresses et les valeurs des données scalaires et envoie le même signal de contrôle au tableau de processeurs. Les processeurs sont interconnectés via un réseau de proximité appelés X-Net. Le Xnet connecte directement chaque PE à ces 8 voisins dans une grille 2D, dans les directions nord, sud, est, ouest, nord-est, nord-ouest, sud-est et sud-ouest. Ce réseau est uniforme; les PEs actifs peuvent accéder à d'autres PEs se situant à une distance et une direction uniformes. Plus précisément, un PE actif donné peut communiquer avec un PE quelconque dont il existe un lien en ligne droite entre eux et ceci dans l'une des 8 directions, et tous les PEs dans la même direction[108]. Les PEs inactifs peuvent servir en étages de pipeline pour des expéditions de longues distances traversant plusieurs PEs[108]. Chaque PEs a exactement 8 voisins puisque les liens de communications sont toriques, et de ce fait, le réseau est une grille torique.

Un deuxième réseau général appelé Global Router permet de connecter des PE quelconques. Le Global Router est un crossbar 3-étages (un réseau de Clos). Les ports du Router sont multiplexés par 16 PE groupés en cluster. Ce réseau permet des communications entre tous les PE des clusters, mais si un PE est connecté à lui même ou à un autre PE du même cluster, l'accès au cluster est géré séquentiellement. Le Global Router est plus général que le X-Net mais plus lent. La bande passante du X-Net est 16 fois celle du Global Router [112] [39].

Notons que sur ces machines dotées de dizaines de milliers de processeurs, le parallélisme exploité est un parallélisme à grain fin (i.e., le rapport du nombre de données traitées sur le nombre de processeurs utilisés est proche de 1). Les réseaux d'interconnection sont donc supposés offrir une facilité de mouvements des données entre processeurs.

Le modèle d'exécution SIMD correspond directement au modèle de programmation *parallélisme de données*; la même action est effectuée sur des données en parallèle [46]. Notons que les machines vectorielles pipelines bien qu'elles soient parfois présentées comme MIMD ou MISD sont des machines à parallélisme de données.

Parmi les langages utilisés pour la programmation à parallélisme de données, citons le High Performance Fortran HPF. Ce langage permet l'expression du parallélisme de données indépendamment de la machine cible et ne se limite pas aux machines SIMD mais recouvre aussi les machines MIMD. HPF manipule des objets (des tableaux) au moyen de directives de distribution et d'alignement des éléments de tableau sur un ensemble de processeurs virtuels. Le mappage des processeurs virtuels (des grilles) sur les processeurs physiques qui dépend de l'implantation est à la charge du compilateur guidé par des directives. D'autres langages, extensions du langage C, intègrent des constructions pour la programmation des machines massivement parallèles, il s'agit du C* pour les Connection Machine et de MPL

pour la machine MasPar [94].

Le modèle PRAM:

Notons par ailleurs que la machine parallèle SIMD à mémoire partagée reste un modèle théorique, qui fait abstraction de la majorité des problèmes liés à la mise en oeuvre d'un algorithme parallèle sur une machine parallèle.

Le modèle PRAM [12] est le modèle abstrait de machines parallèles le plus populaire [4, 75, 30, 81]. Une PRAM d'ordre N consiste en un ensemble de processeurs p_1, p_2, ..., p_N et en une mémoire globale partagée. Cette mémoire globale consiste en M cases (ou blocs) mémoires de même taille, et que l'on peut adresser individuellement. Chaque processeur peut possèder sa propre mémoire locale et son propre contrôleur local si l'on considère le modèle PRAM asynchrone donc non SIMD. Durant chaque étape de calcul, chaque processeur peut avoir accès, en lecture et en écriture, à une case mémoire quelconque de la mémoire globale. La communication entre deux processeurs par le biais de la mémoire se fait en deux étapes; le processeur i désire communiquer avec le processeur j (i.e., lui passer une valeur), i doit écrire la valeur dans une case mémoire connue par j, et j ira lire la valeur à cette case. Le modèle de base est divisé en trois sous-classes selon que deux processeurs ou plus peuvent accèder à une même case mémoire simultanément.

Dans une PRAM *à lecture exclusive et à écriture exclusive* EREW [13], les accès mémoire des N processeurs doivent vérifier qu'à chaque étape, au plus un processeur lit ou écrit sur une case donnée de la mémoire commune. Dans une PRAM *à lectures concurrentes et à écriture exclusive* CREW [14], plusieurs processeurs peuvent lire le contenu d'un même case au même instant, mais un seul processeur peut écrire à un instant donné. Dans une PRAM *à écritures concurrentes* CW [15], plusieurs processeurs peuvent écrire dans une même case au même instant sous la contrainte d'un arbitrage de conflits [81]. Dans le modèle CW *faible*, les écritures simultanées ne sont autorisées que si les processeurs écrivent tous 0. Dans le modèle CW *commun*, les écritures simultanées ne sont autorisées que si les processeurs écrivent tous la même valeur. Dans le modèle CW *arbitraire*, on sélectionne arbitrairement une valeur et on écrit uniquement cette valeur. Dans le modèle CW *prioritaire*, le processeur de plus petit indice écrit sa valeur. Dans le modèle CW *combiné*, la valeur écrite est la combinaison des valeurs proposées par un opérateur associatif et commutatif.

Le modèle PRAM convient idéalement dans le cadre d'étude du parallélisme, mais ne peut prédire des temps d'exécutions corrects pour les algorithmes parallèles sur machines parallèles réelles ayant la mémoire distribuée. En effet, la PRAM ne tient pas compte des liens de communication et on ne se soucie pas de placer les bonnes données aux bons endroits et pour que l'on puisse les retrouver au bon moment dans la mémoire distribuée parmi les processeurs. Le modèle PRAM fait donc abstraction de la majorité des problèmes liés à la mise en oeuvre d'un algorithme parallèle sur une machine parallèle. De plus, une mémoire globale partagée n'est pas facilement réalisable matériellement [4] [81]. La PRAM peut être simulée en distribuant la mémoire partagée parmi les processeurs liés par le réseau d'interconnexion. En cas de lecture, le paquet contenant les données lues est routé à travers

12. Acronyme provenant de Parallel Access Random Machine
13. Exclusive-Read, Exclusive-Write
14. Concurrent-Read, Exclusive-Write
15. Concurrent-Write

le réseau au processeur ayant fait la requête. En cas d'écriture, le paquet contenant les données est routé à travers le réseau au processeur détenant la partie de la mémoire désirée. D.Nassimi et S.Sahni ont présenté dans [35] ces deux opérations, appelées *lecture à accès aléatoire* (RAR) et *écriture à accès aléatoire* (RAW), dans le cas d'une machine parallèle synchrone, dont le réseau peut être un hypercube, une grille ou le graphe mélange-parfait. F.Stout et R.Muller ont présenté également ces opérations de mouvement de données dans le cas où le réseau est une pyramide dans [106]. Ces opérations permettent non seulement de simuler la PRAM, mais résout également le problème de mouvement de données pour de nombreux algorithmes (algorithmes sur les graphes, algorithmes d'analyse d'images, géométrie algorithnique, ...).

1.3 Les réseaux d'interconnexion

La définition et l'optimisation du réseau d'interconnexion restent les problèmes fondamentaux des architectures parallèles. En effet les réseaux d'interconnexion forment l'ossature des machines parallèle.

Un réseau d'interconnexion peut être modélisé par un graphe. Un graphe complètement connecté modélise le réseau idéal où chaque processeur est connecté à tous les autres. Un tel réseau est malheureusement limité à un petit nombre de noeuds à cause des limitations dues essentiellement aux contraintes physiques comme celles liées au nombre de broches ou à la surface VLSI des circuits.

Plusieurs propriétés concernant le graphe modélisant le réseau ont été définies pour caractériser les connexions entre les couples de processeurs. Parmi ces propriétés de connexion on trouve:

- **Le degré** d'un sommet est le nombre de ses voisins, ou encore le nombre de ses liens incidents. Un graphe est dit régulier, si tous les sommets ont le même degré, appelé alors le degré du graphe.

- **La distance** entre deux sommets est la longueur du plus court chemin qui les relie, évalué en nombre de liens traversés.

- **le diamètre** du graphe est le maximum des distances dans le graphe.

- **La largeur de bissection** est le nombre minimum de liens dont la destruction entraîne la séparation du réseau en deux moitiés ayant un nombre de sommets identiques (à un près).

Ces propriétés purement géométriques des graphes nous fournissent un premier ensemble de critères de comparaison des réseaux. On trouve aussi d'autres propriétés matérielles associées par exemple aux liens comme la largeur du canal, c'est-à dire la quantité d'informations élémentaires d'un transfert physique entre processeurs voisins.

La géométrie du réseau d'interconnexion d'une machine parallèle devrait présenter les caractéristiques suivantes :

1. un faible diamètre, afin de minimiser le temps de communication entre deux sommets quelconques. En effet, plus le diamètre est grand plus le temps de latence[16] est important.

2. un faible degré, pour réduire le nombre de connexions nécessaires, et par conséquent le coût matériel. Ce n'est certes pas un problème pour de petits réseaux, mais cela peut le devenir lorsqu'on considère des machines à grand nombre de processeurs à cause des limitations technologiques.

3. doit permettre de relier un très grand nombre de sommets pour obtenir un taux de parallélisme élevé (le cas des machines massivement parallèles).

4. une bissection large. La largeur de bissection, comme le diamètre, est souvent un facteur essentiel pour estimer la vitesse à laquelle un réseau peut effectuer un calcul. En effet, pour de nombreux problèmes, il est parfois nécessaire que les données traitées et/ou stockées dans une moitié du réseau doivent être transmises à l'autre moitié. La bissection est également un important paramètre pour la réalisation en VLSI du réseau [81].

On distingue principalement deux types de réseaux de communication : les réseaux à **topologie statique** et les réseaux à **topologie dynamique**. Nous présentons dans ce qui suit ces deux types de topologies.

1.3.1 Les topologies dynamiques

Dans les topologies dynamiques ou reconfigurables, le support physique est fixé, mais des commutateurs ou connecteurs permettent de modifier le schéma de connexion du réseau. Les processeurs sont connectés à un ensemble de commutateurs, ces derniers étant organisés suivant un graphe donné. L'établissement des communications entre les processeurs se fait pendant l'exécution par le positionnement de ces commutateurs.

Ces réseaux ont été très étudiés dans le cadre des architectures à mémoire commune, pour le partage de cette mémoire (sous forme de bancs mémoires). Le plus simple de ces réseaux est le bus utilisé en temps partagé, mais ne permet de connecter qu'une trentaine au plus de noeuds pour des raisons de saturation [91]. Le plus complexe et le plus performant des réseaux dynamiques est le crossbar que l'on peut représenter par une grille où les processeurs sont en tête des lignes et les mémoires en tête des colonnes. A cause du nombre de points de croisement qui est proportionnel au produit du nombre de processeurs par le nombre de mémoires, ce réseau est malheureusement le plus coûteux. Le crossbar est dit *non bloquant* parce que l'on peut toujours établir une liaison entre toute entrée libre et toute sortie libre, et cela sans modifier les liaisons déjà établies. En effet, toute entrée est directement connectée à toute sortie dans un réseau crossbar. Le bus et le crossbar

16. Temps requis pour l'envoi d'un message de la source vers la destination

bornent ainsi le spectre du prix et de la performance pour pratiquement tous les réseaux d'interconnexion à topologie dynamique.

Dans une autre alternative qui permet de repousser les limites physiques du crossbar, les connecteurs (ou commutateurs) sont organisés en étages (d'où le terme multi-étages), les sorties de l'étage i étant directement connectées aux entrées de l'étage i+1. Les commutateurs les plus utilisés sont des crossbar 2 vers 2, c'est le cas des réseaux papillon et ses variantes. Bien que L'intérêt pour ces réseaux est qu'ils conservent quelques qualités du réseau idéal (le crossbar) avec une complexité moindre, le délai de transmission des données ainsi que la complexité de la commande sont largement augmentés. Ce sont les raisons pour lesquelles ces réseaux ne sont utilisés que dans des architectures de faible dimension. Le réseau de Clos possède un autre schéma de connexion; c'est un réseau constitué de trois étages de connecteurs $n \times q$, $q \times q$ et $q \times n$. Chacun des connecteurs est un réseau crossbar. Un réseau de Clos est réarrangeable et sans blocage. Un réseau est dit *réarrangeable* s'il a le même nombre d'entrées que de sorties et si on peut toujours établir une liaison entre toute entrée libre et toute sortie libre, quitte à modifier des liaisons déjà établies.

Les réseaux multi-étages ont fait l'objet de nombreuses études. Les réseaux papillon (butterfly) et ses variantes les réseaux de Beneš, les rèseaux Oméga (connu aussi sous le nom de réseau de mélange), les réseaux base (Baseline), etc... sont quelques exemples de ces réseaux largement commentés dans l'ouvrage de Leighton [81].

Parmi les projets de machines qui ont utilisé ce type de réseau, on peut citer le projet Supernode [107] qui a utilisé un réseau de Clos pour connecter des Transputers, et la machine CHiP (Configurable Highly Parallel) [128] de l'université de Purdue, formée d'un réseau de processeurs interconnectés par une grille de circuits d'aiguillage (switch) programmables de façon statique. Le SP1 [51] d'IBM est un exemple de machine commerciale utilisant un réseau multi-étage pour la communication. La machine massivement parallèle MasPar utilise aussi un réseau de Clos (3 étages de crossbar).

Parmi les projets actuels dans le domaine des réseaux reconfigurables, citons le projet ARP (Architecture à Réseau Partagé) mené actuellement à l'Université de Lille. Le réseau ARP, qui n'est pas un réseau multi-étages, est un réseau d'interconnexion à reconfiguration dynamique et asynchrone permettant l'allocation de ressources de communication (à la demande) pendant l'exécution d'une application parallèle. Pour une présentation plus complète du réseau citons les réferences [61, 23].

1.3.2 Les topologies statiques

Un réseau à topologie statique se caractérise par un graphe non complètement connecté dont les sommets sont les processeurs et les arêtes les liens de communication. Ces liens sont fixes entre processeurs à la construction de la machine, et ne peuvent être modifiés pour une connexion directe vers d'autres processeurs. Les réseaux d'interconnexion à topologies statiques sont donc particulièrement concernés par le problème de mouvement de données entre processeurs communicants. Dans ce qui suit, nous citons les topologies les plus répandues.

1.3.2.1 L'hypercube

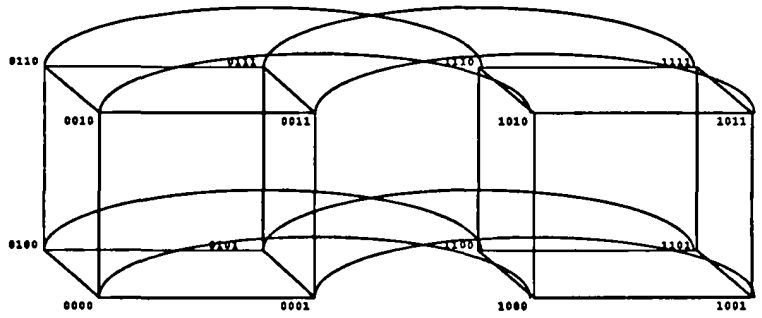

FIG. 1.4 - *hypercube de dimension 4*

L'hypercube est un des réseaux les plus remarquables et les plus efficaces pour la programmation parallèle. Il est souvent adopté pour base architecturale d'une machine parallèle généraliste (i.e., non dédiée à des calculs spécifiques). Il peut en effet simuler efficacement tout autre réseau de même taille [81]. Un hypercube de dimension n posséde $N = 2^n$ sommets, et possède un degré et un diamètre égaux à n (cf figure 1.4). Chaque sommet pssède un numéro codé par un mot binaire de longueur n (n bits). Deux sommets sont adjacents si et seulement s'ils diffèrent d'un seul bit dans leurs représentations binaires. En conséquence, chaque sommet possède $log\ N$ voisins, un pour chaque bit. Une arête de l'hypercube est dans la dimension k, si elle relie deux sommets qui diffèrent par leur k-ième bit. La structure de l'hypercube est récursive, un hypercube de N sommets se construit à partir de deux hypercubes de $\frac{N}{2}$ sommets en reliant le sommet i d'un des hypercubes de $\frac{N}{2}$ sommets au sommet i de l'autre hypercube pour tout i, $0 \leq i < \frac{N}{2}$. En plus de sa structure récursive, l'hypercube possède de bonnes propriétés, un faible diamètre en $log\ N$ et une large bissection égale à $\frac{N}{2}$.

Bien que l'hypercube soit un réseau performant du point de vue calculatoire, un inconvénient relatif à son utilisation en tant qu'architecure de machine parallèle est que le degré de chacun de ses sommets (ou le nombre des ports physiques) croît avec sa taille [81]. Ceci implique qu'un processeur conçu pour un hypercube de N sommets ne peut pas être utilisé pour un hypercube de $2N$ sommets. De plus, la complexité de la partie communication de chaque processeur peut devenir assez importante lorsque N est grand.

Dans le but de surmonter ce problème, des variantes de l'hypercube, préservant les qualités algorithmiques mais possédant un degré borné (3 ou 4), ont été proposées. Ces variantes appelés réseaux hypercubiques de degré borné sont: le papillon, le papillon rebouclé, le réseau de Beneš (qui sont des réseaux multi-étages), les cycles-connectés-en-cube (CCC), le graphe du mélange parfait et le graphe de Bruijn. Leighton [81] montre que ces réseaux peuvent simuler de façon efficace tout hypercube ayant le même nombre de processeurs, bien qu'ils possédent moins de liens. De plus, tous les réseaux hypercubiques de degré borné sont algorithmiquement équivalents à une constante de vitesse près. En d'autres termes, les variantes de l'hypercube ont une puissance de calcul équivalente, et ils peuvent

se simuler les unes les autres avec un facteur de ralentissement constant.

Parmi les machines dont la topologie est un hypercube, citons le Cosmic Cube [123], le Ncube, l'iPSC [131], et la CM-2.

1.3.2.2 La grille

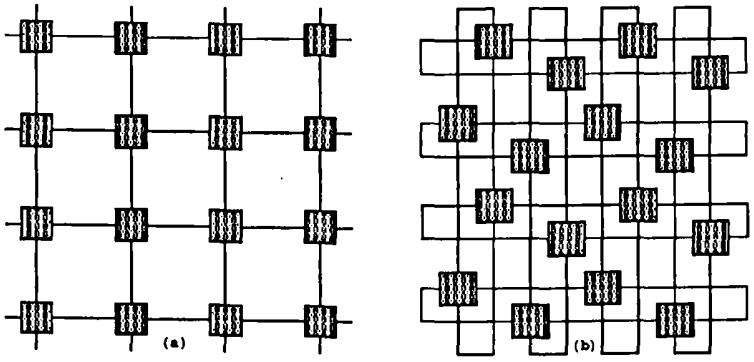

FIG. 1.5 - *Les réseaux (a) grille 2D 4x4 et (b) tore 2D 4x4*

La grille ou mesh de dimension r et de côté N possède N^r sommets que l'on peut voir comme les points de coordonnées entières comprises entre 0 et $N-1$ dans un espace euclidien de dimension r (cf figure 1.5(a)). La grille $N \times N \times ... \times N$ est communément appelée *grille r-dimensionnelle N-aire*. Dans le cas particulier où $N = 2$, on a un hypercube de dimension r. Chaque sommet est connecté à ceux dont une coordonnée diffère exactement de 1 des siennes dans chacune des dimensions. La grille n'est pas un graphe régulier puisque les sommets internes ont un degré $2r$ et ceux situés à la périphérie un degré inférieur (les degré sont compris entre r et $2r$). Le diamètre est $r(N-1)$, et la bissection est N^{r-1} quand N est pair et légèrement plus grande quand N est impair [81]. Un des intérêts majeurs de la grille est son extensibilité : on peut ajouter indéfiniment des noeuds au réseau sans modifier la structure des noeuds existants, en l'occurrence le degré (ou le nombre de ses ports physiques). En ce sens, c'est la plus modulaire des topologies. Malheureusement son diamètre augmente en fonction des paramètres r et N. La grille torique (ou le tore) est une amélioration de la grille, permettant la connexion physique des deux sommets les plus éloignés sur chaque direction (cf figure 1.5(b)), c'est un graphe régulier. La grille et sa variante la grille torique sont utilisées par des machines comme l'Ametek, la MasPar, la machine Paragon [50] et la machine Cray T3D.

1.3.2.3 L'arbre binaire complet

Un arbre binaire complet de k niveaux, numérotés de 0 à $k-1$, posséde $2^k - 1$ noeuds. chaque nœud au niveau i est connecté à son père au niveau $i+1$, et à ses deux fils au niveau $i-1$. Le nœud racine, au niveau $k-1$, ne possède pas de père et les feuilles au niveau

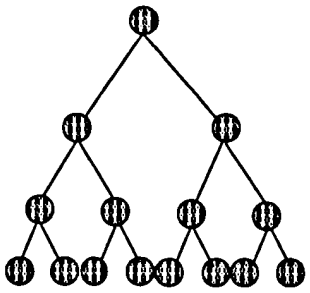

FIG. 1.6 - *un arbre binaire complet de hauteur* 4

0 n'ont pas d'enfants. Notons la simplicité du réseau arbre (régularité, faible degré). Une implémentation matérielle directe d'une machine dont les connections forment un arbre risque d'avoir des liaisons entre niveaux dont la longueur augmente d'une manière exponentielle avec le nombre du niveau. Comme le montre la figure 1.7, le lien connectant un noeud au niveau i à son père au niveau $i+1$ a une longueur proportionnelle à 2^i. La longueur maximale des liens pour un arbre ayant n feuilles est donc $O(n)$. On trouve ce lien maximal entre le niveau $log\, n$ et $log\, n+1$. Bien entendu, cette approche d'implémentation de l'arbre est indésirable du point de vue pratique. En effet, l'utilisation de l'espace dans lequel les processeurs et les liens sont placés est très mauvaise [4]. De plus, cela induit de mauvaises performances pour les algorithmes. Si le temps de communication entre deux niveaux est proportionnel à la longueur du lien, alors le temps d'exécution des communications est en $O(n)$. Pour pallier à ces problèmes, on plonge l'arbre binaire complet dans une grille en utilisant la représentation dite en H, comme l'indique la figure 1.8.

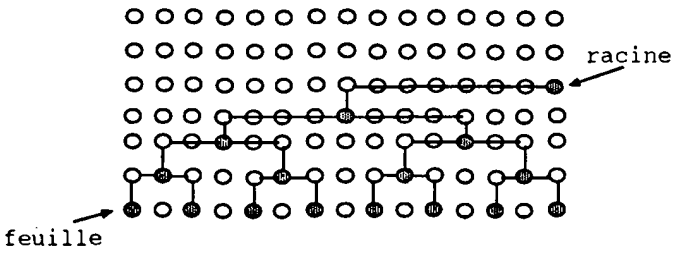

FIG. 1.7 - *Une implémentation directe d'un arbre ayant 8 feuilles dans une grille 2D. Le lien connectant la racine au niveau* 4 *à un de ses deux fils au niveau* 3 *a une longueur proportionnelle à* 2^3. *Ce lien possède la longueur maximale.*

Des machines multiprocesseurs de la forme d'un arbre binaire ont été conçues telles que la machine X-tree [31], la machine DAC et la machine P-tree [57].

Une variation de l'arbre binaire complet est le X-arbre. Un X-arbre est un arbre binaire complet auquel on ajoute des arêtes entre les sommets d'un même niveau. La figure 1.9.(a) montre un X-arbre contenant huit feuilles.

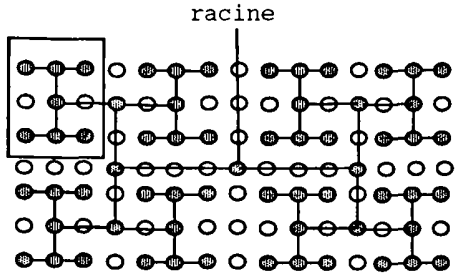

FIG. 1.8 - *Un arbre en H de 6 niveaux dans une grille 2D*

La pyramide est une généralisation bidimentionnelle du X-arbre qui est une pyramide unidimensionnelle. La figure 1.9.(b) illustre un exemple d'un réseau pyramide 4×4. Dans un tel réseau, $\frac{4n}{3} - \frac{1}{3}$ processeurs sont distribués sur $1 + log_4\ n$ niveaux, n étant une puissance de 4. A chaque niveau, les processeurs sont connectés pour former une grille. Au niveau $log_4 n$ appelé apex, il y a un processeur, et au niveau 0 appelé base, il y a n processeurs disposés en grille $\sqrt{n} \times \sqrt{n}$. Au niveau i, $0 \leq i \leq log_4 n$, la grille consiste en $n/4^i$ processeurs. Un processeur au niveau k est également connecté à son processeur père au niveau $k+1$ et à 4 processeurs fils au niveau $k-1$. Parmi les projets qui ont été initiés sur les machines pyramidales, citons le projet Sphinx de l'ETCA et l'Université Paris Sud, dans lequel l'architecture de type multi-SIMD proposée regroupe un grand nombre de PE 1 bit. La machine était destinée aux applications dont les algorithmes utilisent des structures de données arborescentes [28]. Une classe importantes de problème requiert en effet des structures de données arborescentes. Les représentations multirésolution d'une image, les techniques de compression de données ne retenant que l'information utile dans une image (les méthodes quadtree), et les méthodes de segmentation en régions sont des problèmes ayant des traitements associés à des structures de données pyramidales [60].

Par ailleurs, d'autres exemples d'algorithmes sur le réseau pyramide, comme des algorithmes sur les graphes, sont données dans [106]. On en déduit d'ailleurs que le recours à une architecture ne reflétant pas la structure de donnée associée au traitement souhaité, nécessiterait des mécanismes de communications matériels ou logiciels moins efficaces mais plus généraux (routage général, à la différence de communications point à point).

Leighton [81] montre que la multigrille (voir la figure 1.9.(c)) est un sous-graphe de la pyramide et montre aussi que les deux réseaux sont pratiquement algorithmiquement équivalents. L'avantage des multigrilles sur les grilles est le suivant ; le diamètre d'une simple grille $N \times N$ est en $O(\sqrt{N})$ alors que celui d'une multigrille (et d'une pyramide également) $N \times N$ est en $O(log\ N)$.

Une autre variation de la topologie en arbre est l'arbre élargi (fat-tree) (voir la figure 1.10) implanté sur la CM-5. L'arbre élargi conserve la structure arborescente, mais évite les goulots d'étranglement en fournissant des capacités de communications uniformes. Une telle topologie offre une bonne résistance aux pannes en permettant de doubler chaque lien et chaque contrôleur [119]. Dans la CM-5, les processeurs sont les feuilles, et les noeuds intérmediaires des contrôleurs et des routeurs. La CM-5 est dotée des trois réseaux suivants. Le réseau de données dont la topologie est un arbre élargi quaternaire et le réseau

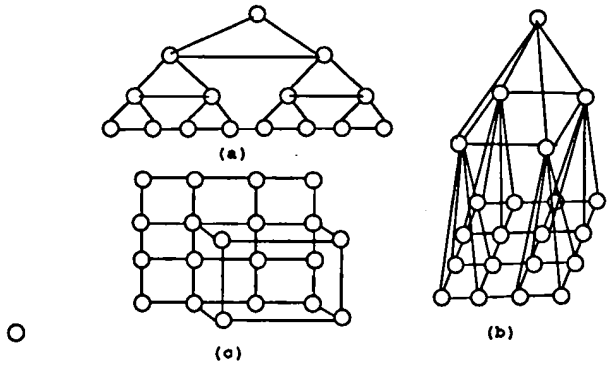

FIG. 1.9 - *Un X-arbre de 8 feuilles (a), une pyramide* 4×4 *(b) et une multigrille* 4×4 *(c)*

de contrôle qui est un arbre binaire complet, sont utilisés pour réaliser les fonctionalités associées au mode SPMD (diffuser les programmes en dupliquant le code sur tous les processeurs, réaliser les barrières de synchronisation, ...). Le réseau de diagnostic organisé en un arbre binaire (pas nécessairement complet), est utilisé pour la maintenance matérielle de la machine [62, 41].

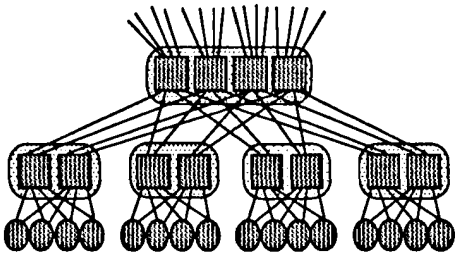

FIG. 1.10 - *L'arbre élargi (fat-tree) implanté sur la CM5*

1.3.2.4 La grille d'arbres

La grille d'arbres est un réseau hybride basée à la fois sur les grilles et les arbres. Une grille d'arbres (mesh of trees) bidimensionnelle $N \times N$ est obtenue à partir d'une grille de processeurs $N \times N$ en ajoutant des processeurs et des liens de communications pour former un arbre binaire complet en chaque ligne et chaque colonne (voir la figure 1.11). Les interconnexions en arbre sont les seuls liens entre processeurs. Les feuilles de l'arbre sont les N^2 processeurs originals de la grille. Ces processeurs ont un degré 2. Les autres processeurs rajoutés sont les sommets internes des arbres et ont un degré 3. Le diamètre d'une grille d'arbre est $4 \log N$. Les grilles d'arbres ont un petit diamètre, une grande bissection et une structure hautement récursive (notons la différence avec les arbres et les grilles qui ont un grand diamètre et/ou une petite bissection). La décomposition récursive

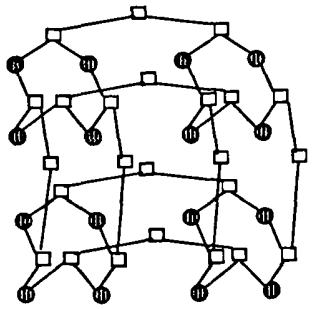

FIG. 1.11 - *La grille d'arbres* 4×4

peut être exploité pour la mise en œuvre des algorithmes parallèles récursifs de la manière suivante. Les processeurs racines peuvent être utilisés pour des communications globales, et les processeurs du plus bas niveaux pour les calculs. La décomposition récursive est également utile pour la réalisation VLSI du réseau. La grille d'arbres est un réseau trés puissant pour le calcul parallèle. Leighton montre dans [81] que la grille d'arbre peut être présentée comme un réseau de degré borné dérivant du *graphe bipartie complet* $K_{N,N}$[17] (dans ce graphe, les processeurs accèdent directement aux mémoires de tous les autres processeurs, c'est le réseau idéal). Ceci signifie que la grille d'arbres a pratiquement la même puissance de calcul que $K_{N,N}$.

Notons que les trois réseaux, la grille d'arbres $N \times N$, la pyramide $N \times N$ et la multigrille $N \times N$ sont obtenus en combinant de façon astucieuse les structures d'arbre binaire complet et de grille bidimensionnelle. Les trois réseaux ont $\Theta(N^2)$ sommets, une bissection $\Theta(N)$ et un diamètre de $\Theta(log\ N)$. La grille d'arbres est cependant significativement plus puissante que la pyramide et la multigrille (pour une approche plus complète sur ce réseau, citons [81, 136]).

Notons enfin que dans sa forme générale, la grille d'arbres $N \times N \times ... \times N$ de dimension r est construite en ajoutant des arbres à une grille de dimension r et de côté N.

TAB. 1.1 - *Diamètres et bissections de quelques réseaux*

Réseau	Taille	Diamètre	Largeur de bissection
grille de dimension r et de côté n	n^r	$r(n-1)$	$n^{r-1} + o(n^{r-1})$
arbre bianire complet de r niveaux	2^{r-1}	$2(r-1)$	1
grille d'arbres $n \times n$	$3n^2 - 2n$	$4log\ n$	n
hypercube de dimension r	2^r	r	2^{r-1}
papillon de dimension r	$(r+1)2^r$	$2r$	$\Theta(2^r)$
CCC de dimension r	$r2^r$	$5r/2 - 1$	$\Theta(2^r)$
graphe de Bruijn de dimension r	2^r	r	$\Theta(2^r)/r$

17. Le *graphe complet* K_N est le réseau idéal où chaque processeur est connecé à tous les autres. Si on sépare chaque processeur en deux parties (parite calcul et partie mémoire), on obtient le *graphe biparti complet* $K_{N,N}$

1.4 Conclusion

Certes, des liens directs existent entre processeurs dans un réseau de communication, mais ces liens peuvent être limités et diffèrent d'une topologie à une autre. Nous avons donc de bonnes raisons pour étudier les problèmes de plongement de graphes.

En effet, les problèmes suivants se modélisant comme des problèmes de plongement de graphes:

- analyser l'émulation d'un réseau par un autre ce qui permet de comparer la puissance de calcul de ces réseaux, et permet aussi de rapatrier les algorithmes conçus pour le premier réseau dans le deuxième.
- distribuer et placer des tâches modélisées sous forme d'un graphe dans les processeurs de la machine.
- trouver une représentation de stockage efficace des structures de données.
- trouver des méthodes de placement éfficaces pour réaliser physiquement un réseau logique en VLSI.

Nous consacrons le chapitre suivant à la présentation des définitions générales et des terminologies relatives à la notion de plongement de graphes.

Chapitre 2

Le plongement de graphes

Dans ce chapitre, nous allons présenter les définitions générales et les terminologies relatives à la notion de plongement de graphes. Ces définitions peuvent être également trouvées dans [81, 119, 19]. Nous allons donner aussi les motivations pour lesquelles l'étude de plongements est si utile. Nous concluons ensuite avec trois exemples de plongements.

Soient $G(V_G, E_G)$ et $H(V_H, E_H)$ deux graphes, V étant l'ensemble des noeuds et E l'ensemble des arcs. Un *plongement* du graphe G dans le graphe H est défini par la donnée d'une application f qui associe les sommets de V_G aux sommets de V_H, et d'une application p_f qui associe à une arête (u, v) de E_G un chemin $p_f(f(u), f(v))$ dans H, allant du sommet $f(u)$ au sommet $f(v)$.

Dans le cas où l'application f est non injective, en particulier si H a moins de sommets que G, le plongement est parfois appelé *placement* (*mapping*). Dans le reste du document, on ne parlera que de plongement, f étant injective ou non, et on designera le plus souvent un plongement par f sans toujours préciser l'application p_f.

On peut trouver plusieurs manières de plonger un graphe G (le graphe que l'on plonge est parfois appelé *graphe logique ou virtuel*) dans le *graphe hôte* H. Pour mesurer l'efficacité des plongements, beaucoup de paramètres ont été définis, dont les plus étudiés sont *la charge*, *la dilatation* , *la congestion*, et *l'expansion*. Ces paramètres sont décrits ci-dessous.

- *la charge* d'un plongement est le nombre maximum de sommets du graphe logique G qui sont plongés sur un seul sommet du graphe hôte H. Une charge égale à 1 correspond à un plongement injectif. Dans le cas des plongements non injectifs, ce paramètre prend une grande importance.

- *La dilatation* d'une arête (u, v) de G sous le plongement f est la longueur du chemin $(f(u), f(v))$ dans H. La dilatation du plongement f est la dilatation maximale prise sur toutes les arêtes de G sous f. Dire que G se plonge avec une dilatation égale à 1 dans H est équivalent à dire que G est un sous-graphe partiel de H. Dans ce cas, l'image d'une arête (u, v) de G est réduite à l'arête $(f(u), f(v))$ de H.

- *la congestion* d'une arête e dans H est le nombre de chemins, images d'arêtes de G qui contiennent e. La congestion du plongement est la congestion maximale prise sur

toutes les arêtes e de H.

- *L'expansion* d'un plongement est le rapport entre le nombre de sommets H et le nombre de sommets de G. Ce paramètre mesure le degré d'utilisation des processeurs. Si le plongement est injectif, une expansion égale à 1 peut correspondre à une utilisation optimale des processeurs.

Le meilleur plongement est évidemment celui pour lequel la charge, dilatation, expansion, et congestion sont minimales. En effet, ces paramètres mesurent la vitesse et l'efficacité avec lesquelles le graphe cible peut simuler le graphe de départ. Si tous les paramètres sont constants alors le graphe cible est capable de simuler d'une manière efficace le graphe logique avec un facteur de ralentissement constant (i.e. à une constante près) [81]. Si les valeurs des paramètres ne sont pas toutes les plus petites, il faut alors voir si un compromis interessant peut être réalisé. En effet, pour la plus part des problèmes de plongement, il est presque impossible d'obtenir un plongement qui minimise tous ces paramètres simultanément [81, 119]. Notons par ailleurs que quand le plongement est injectif, la charge vaut 1 et la dilatation, l'expansion et la congestion sont les paramètres important permettant de mesurer l'efficaté du plongement. Par contre, quand f est non injective, la charge devient un paramètre très essentiel.

Notons qu'il existe deux types de plongements: le plongement *statique* et le plongement *dynamique*. Le plongement est statique quand la structure et la taille du graphe logique sont connues à l'avance (par exemple un arbre binaire complet de taille donnée). Ce type de plongement est souvent plus facile à construire que le plongement dynamique. Le plongement est dynamique quand on ne connait pas a priori la structure du graphe de départ. Il semblerait donc difficile dans ce cas de construire de façon deterministe un plongement d'une bonne efficacité.

Les résultats de plongement de graphes ont d'importantes applications dans le traitement parallèle. Elles fournissent une base théorique pour l'étude de nombreux problèmes se modélisant comme des problèmes de plongement de graphes [119]:

- Exécuter un algorithme parallèle sur un réseau de processeurs. Certains algorithmes parallèles peuvent être représentés par un graphe G. En effet, Supposons qu'un processus se décompose naturellement en un ensemble de sous-processus qui peuvent être exécutés en parallèle et peuvent communiquer entre eux. On obtient un graphe en représentant chaque processus par un noeud et chaque lien de communication par un arc entre les noeuds correspondants. Une machine distribuée peut aussi être représentée par un graphe H, dans lequel les sommets représentent les processeurs et les arêtes représentent les liens de communication entre les processeurs. Il s'agit donc de pouvoir simuler efficacement G sur H en plaçant les sommets de G sur les processeurs de H et en affectant un chemin physique de communication dans H à chaque arête de G. De même, dans [117], on montre aussi que le problème de trouver une représentation de stockage efficace d'une structure de données (l'encodage d'une structure de donnée), quand la structure du stockage et la structure de donnée sont representées comme des graphes, se réduit aussi à un problème de plongement de graphes.

- Emuler une architecture par une autre : ceci dans le but de faire exécuter sur de nouvelles machines des algorithmes qui s'effectuent déjà de manière efficace sur une machine connue (par exemple comment faire exécuter sur un hypercube un algorithme défini sur une grille ou sur un arbre binaire complet). Ce problème de simulation d'un réseau par un autre est également modélisé par un problème de plongement de graphe dans lequel les noeuds du graphe représentent les processeurs et les arcs du graphe représentent les liens de communications entre processeurs.
- Réaliser physiquement un réseau logique (circuit VLSI) en respectant les contraintes technologiques. En particulier, dans des structures à deux dimensions, respecter les longueurs de liens limitées, et l'espace nécessaire pour l'accueil des processeurs et les liens les connectant [4, 119, 105]. En plus de leur impact sur la faisabilité d'un réseau physique (l'intégration VLSI), dans certains réseaux actuellement disponibles, certains liens physiques peuvent être plus longs que d'autres. Ceci remet en cause la possibilité d'une hypothèse selon laquelle une quantité constante d'informations peut toujours être transmise le long de chaque lien à chaque étape. En effet, il peut arriver que des liens plus longs nécessitent plus de temps pour effectuer une communication [81, 4]. Réaliser ainsi physiquement un réseau logique, par exemple un arbre binaire complet sur un chip de deux dimensions, est également modélisé en un problème de plongement de cet arbre binaire complet dans une grille bidimensionnelle.

Dans ce qui suit, nous allons donner quelques exemples de plongements en montrant entre autres comment les algorithmes décrits pour les grilles et les arbres, peuvent être automatiquement mis en oeuvre sur un hypercube. Nous comprendrons ainsi quelques raisons pour lesquelles l'étude de plongements est si utile.

Un des premiers plongements qui a été étudié est celui du plongement d'une grille dans un hypercube. Beaucoup de travaux ont été éffectués sur les plongements de structures régulières, telles que les anneaux, les grilles de dimension deux et plus, et les arbres binaires complet sur un des réseaux les plus populaires, l'hypercube. En effet, ce dernier peut simuler de telles structures (avec une taille approximativement identique) avec un fateur de ralentissement faible et constant [81, 119].

Exemple 1

Il existe une très grande variétés d'algorithmes qui peuvent facilement être mis en oeuvre sur les grilles. Ces derniers sont ainsi courramment utilisées en imagerie, en calcul matriciel et pour la mise en oeuvre d'algorithmes manipulant des graphes. Plonger efficacement les grilles sur les hypercubes signifie que tous ces algorithmes peuvent être mis en oeuvre diretement sur les hypercubes sans perdre de façon significative de leurs performances.

Rappelons qu'un hypercube de 2^r sommet est le cas particulier d'une grille de dimension r et de côté 2. Prouver que toute grille de 2^r sommets et de dimension quelconque est un sous-graphe de l'hypercube de 2^r sommets, implique que ces grilles se plongent avec dilatation 1 dans les hypercubes de même taille. La preuve de ce résultat détaillé dans [81, 119] est basé sur le fait que l'hypercube est hamiltonien. Ce qui a pour conséquence directe qu'un réseau linéaire de N sommets est un sous-graphe de l'hypercube de N sommets. La suite de la preuve est basé sur le fait qu'une grille est le produit cartésien de réseaux linéaires. L'hypercube (qui est une grille de côté 2) est également le produit cartésien

d'hypercubes de tailles plus petites. La propriété de "sous-graphe" étant preservée par le produit cartésien, on déduit que la grille de dimension quelconque et de 2^r sommets est un sous-graphe de l'hypercube de 2^r sommets. La figure 2.1 illustre le plongementd'une grille 4×4 dans un hypercube de 16 sommets.

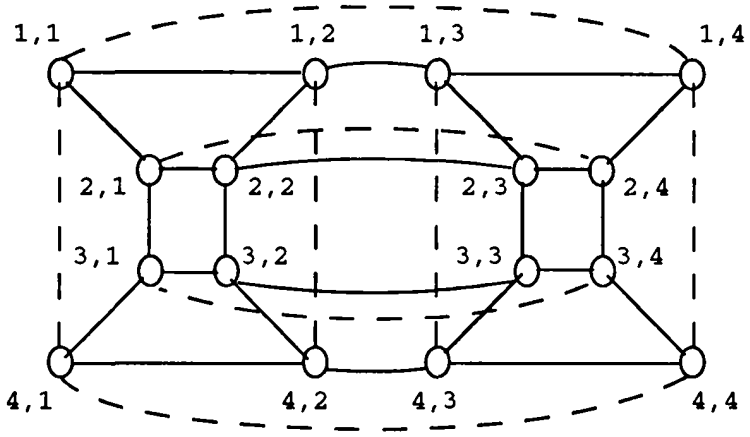

FIG. 2.1 - *Plongement d'une grille* 4×4 *dans un hypercube de* 16 *sommets. La dilation du plongement est* 1 *(les liens en pointillés sont les liens de l'hypercube non utilisés par le plongement de la grille.*

Les grilles dont la taille N n'est pas une puissance de deux ne sont pas des sous-graphes des hypercubes de taille N. La figure 2.2 illustre un plongement de dilatation 2 d'une grille 3×5 dans un hypercube de 16 sommets. Pour simplifier, considérons le cas d'une grille bidimensionnelle $N \times M$. Pour construire un plongement de dilation 1, deux sommets adjacents dans la grille doivent être adjacents dans l'hypercube. Ceci signifie que deux sommets adjacents d'une même ligne i dans la grille ont pour images des sommets de l'hypercube qui diffèrent d'un bit, d'un rang k_i. De même, deux sommets adjacents d'une même colonne j dans la grille ont pour images des sommets de l'hypercube qui diffèrent d'un bit d'un rang k_j. Il faut bien sûr que k_j soit différent de tous les k_i. On déduit donc que les sommets de l'hypercube doivent différer sur $\lceil M \rceil$ rangs pour les lignes et $\lceil N \rceil$ pour les colonnes. Pour que la grille $N \times M$ soit un sous graphe de l'hypercube de dimension d et puisse ainsi se plonger avec une dilation 1, il faut donc que $d \geq \lceil N \rceil + \lceil M \rceil$. Plus généralement pour qu'une grille $M_1 \times M_2 \times ... \times M_k$ soit un sous-graphe d'un hypercube de dimension d, il faut que $d \geq \lceil M_1 \rceil + ... + \lceil M_k \rceil$ [81, 119].

On peut cependant construire un plongement de dilatation 1 en dépit de la charge ou de l'expansion. Ainsi, on peut choisir de construire le plongement de de la grile 3×5 toujours dans son hypercube optimal de 16 sommets avec une dilatation 1 et une charge 2 ou bien construire un plongement de dilatation 1 mais sur hypercube de 32 sommets. Ce genre de compromis se fait souvent dans les plongements. En effet, il arrive souvent qu'il soit impossible de minimiser tous les paramètres à la fois.

Exemple 2

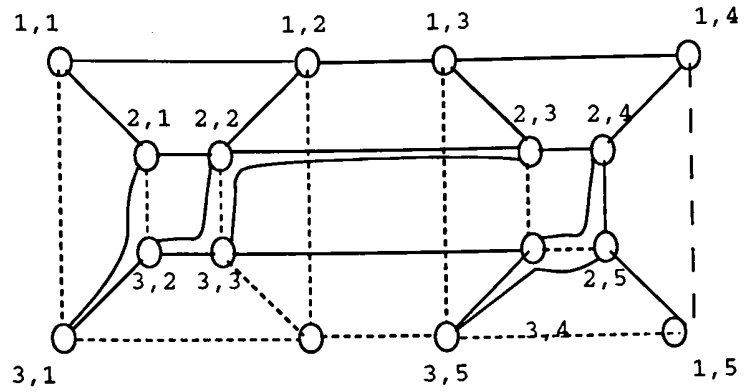

FIG. 2.2 - *Plongement de dilataion 2 d'une grille* 3×5 *dans un hypercube de* 16 *sommets. Par exemple, l'arête connectant les deux sommets de la grille* $(2,2)$ *et* $(3,2)$ *est projetée sur deux arêtes de l'hypercube [81].*

Les arbres sont des structures très importantes en informatique. Ils sont utilisés comme structures de données représentant des données hierarchiques ou interviennent comme graphe de contrôle intrinsèque à beaucoup de problèmes algorithmiques. En effet, nombreux sont les problèmes algorithmiques qui se décomposent naturellement en une structure d'arbre tels que les problèmes d'évaluation d'expressions fonctionnelles, d'optimisation combinatoire et de type "diviser-pour-règner" [56, 98]. Plonger efficacement les arbres dans les grilles et les hypercubes signifie que ces algorithmes peuvent être mis en oeuvre directement sur ces architectures. Une autre solution consiste a utiliser un réseau de processeurs connectés en arbre binaire complet. Ce réseau, par ses connexions simples et régulières et dont le nombre de ports de communication au niveau de chaque nœud ne dépasse par trois, est très pratique du point de vue intégration VLSI. Si le problème traité requiert un arbre dont le nombre des fils de chaque noeud dépasse deux, une simulation de cet arbre sur l'arbre binaire peut être effectuée comme cela a été fait dans [98]. Cependant, la réalisation VLSI d'un réseau en arbre sur un chip VLSI nécessite aussi un plongement efficace de l'arbre binaire complet dans une grille 2D. Ce dernier doit être éconmique en espace et en longueur de liens utilisés.

La figure 2.3 illustre un exemple de plongement naïf d'un arbre binaire complet dans une grille bidimensionnelle. Ce plongement n'est pas pratique étant donné que l'utilisation des processeurs de la grille et le placement des liens sont très mauvais. Le plongement d'un arbre binaire complet de N sommets nécessiterait une grille de taille $O(N log N)$ et une dilatation en $O(N/3)$.

Pour pallier à ces problèmes, un plongement dit plongement en H a été proposé [98, 58]. La figure 2.4 illustre un plongement en H d'un arbre binaire complet de 15 sommets. Ce plongement plonge un arbre de N sommets dans une grille de taille $O(N)$ et la dilatation est en $O(N^{1/2})$.

Notons que le plongement d'un arbre binaire complet dans la grille est un plongement statique étant donné que la structure de l'arbre est connue à l'avance. Dans le chapitre suivant, nous allons proposer d'autres méthodes de plongements dans le but d'améliorer

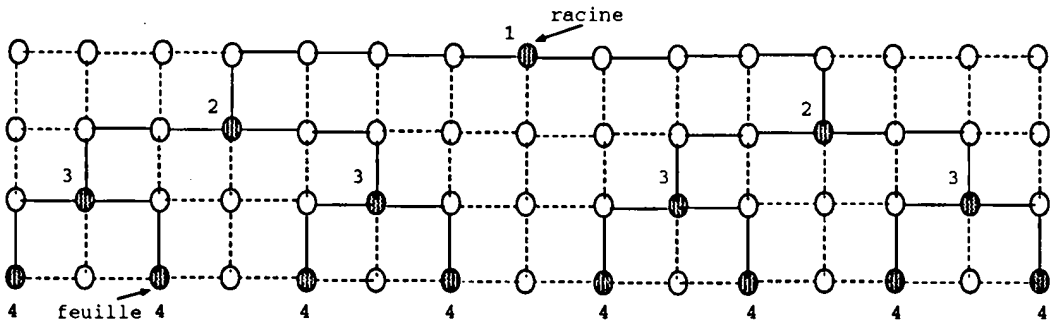

FIG. 2.3 - *Exemple d'un plongement d'un arbre binaire complet de* 15 *sommets dans une grille de* 4 × 15 *sommets. Les arêtes de la grille indiquées en traits pleins représentent les chemins associés aux arêtes de l'arbre. Les arêtes de la grille indiqués en traits pointillés sont inutilisées. Les sommets de la grille images des sommets de l'arbre sont représentés par des cercles pleins.*

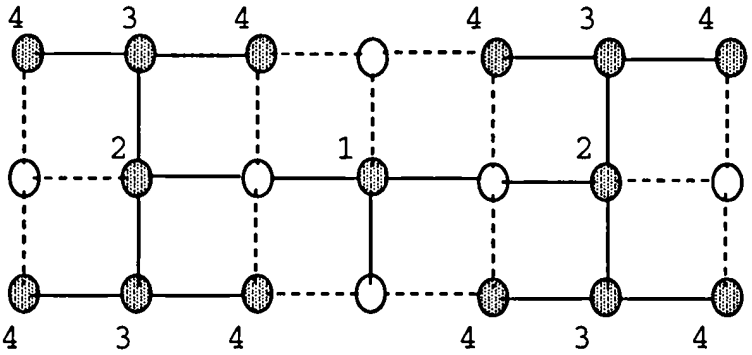

FIG. 2.4 - *Plongement en H d'un arbre binaire complet de* 15 *sommets dans une grille de* 3 × 7 *sommets.*

l'expansion et la dilatation [72]. Ces méthodes de plongements ont été reprises, améliorées et étendues aux plongements des arbres quaternaires (quadtrees) sur les grilles bidimensionnelles dans [101].

Exemple 3

Pour comprendre une des raisons "algorithmiques" pour lesquelles l'étude de plongements est si utile, nous commençons par donner dans ce qui suit l'exemple d'un algorithme déstiné à être mis en œuvre sur un réseau en arbre binaire complet. Cet algorithme standard, de calcul de préfixes en parallèle, peut être automatiquement mis en oeuvre sur un hypecube à l'aide d'un plongement de l'arbre binaire complet dans cette structure [16, 95, 96]. Nous allons ainsi voir que, bien que cet algorithme soit destiné au départ à être exécuté sur un arbre binaire complet, il est plus performant que l'algorithme qui résout le même probléme et qui est décrit pour l'hypercube !

Le calcul de préfixes en parallèle est couramment utlisé comme opération de base dans

de nombreux problèmes qui nécessitent ce calcul (voir des exemples d'applications dans [4, 75, 81]. Nous utiliserons d'ailleurs cette opération dans le chapitre 4 pour mettre en œuvre des algorithmes de routage.

Soit un ensemble de N éléments $x_0, x_1, ..., x_N$ muni d'un opérateur associatif quelconque $\oplus$. L'opération de calcul de préfixes consiste à calculer les sommes partielles $y_i = x_1 \oplus ... \oplus x_i$ pour $1 \leq i \leq N$.

L'algorithme de préfixes en parallèle peut être mis en oeuvre sur un arbre binaire complet de la manière suivante. Soit un arbre binaire complet de N sommets étiquetés suivant la numérotation infixée (dans une numérotation infixée d'un arbre binaire, les étiquettes sont assignées aux processeurs suivant le parcours Gauche-Racine-droit). La figure 2.5 illustre la numérotation infixée d'un arbre binaire complet de 15 sommets.

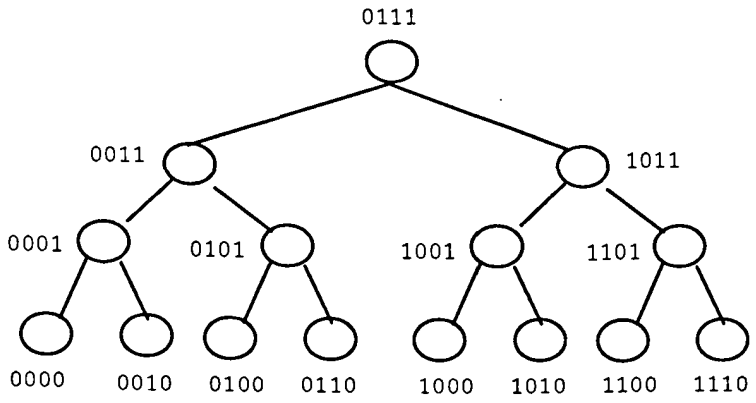

FIG. 2.5 - *Numérotation infixée d'un arbre binaire complet.*

Au départ, chaque processeur i détient l'élément x_i. Pour chaque processeur i, notons par $T(i)$ le sous-arbre de racine le processeur i. L'algorithme sur l'arbre binaire complet se déroule en deux phases. La première phase est une phase ascendante depuis les feuilles jusqu'à la racine. Dans cette première phase, chaque processeur reçoit la somme de son sous-arbre gauche et la somme de son sous-arbre droit, calcule la somme de $T(i)$ et transmet le résultat à son père. Dans la deuxième phase, chaque processeur i reçoit de son père la somme calculée par tous les processeurs d'indices inférieurs à ceux de $T(i)$, calcule la somme de tous les processeurs d'indices inférieurs à ceux de son sous-arbre droit, et transmet les valeurs appropriées à ses fils gauche et droit. Cet algorithme s'exécute en $4 log N$ étapes. Notons que dans cet algorithme, deux niveaux de l'arbre sont actifs à chaque étape. En pipelinant cet algorithme, on peut donc résoudre k opérations de calcul de préfixes en $2k + 4 log N$ étapes [95, 96].

L'algorithme de calcul de préfixe peut être implementé sur un hypercube pour s'exécuter en $log N$ étapes comme nous allons le voir en détail dans le chapitre 4. Cependant, étant donné que dans cet algorithme, tout les processeurs sont actifs à chaque étape, cette implémentation ne peut pas s'exécuter de façon pipeline. Ainsi pour pouvoir exécuter k

opérations de calcul de préfixe en $O(k + log\ N)$ sur un hypercube de N processeurs, il suffit de considérer l'algorithme de calcul des prefixes sur l'arbre binaire complet. Plus précisément, il suffit de plonger l'arbre binaire complet sur l'hypercube pour simuler cet arbre et pouvoir exécuter ainsi l'algorithme décrit ci-dessus de façon pipeline [95, 96].

Par ailleurs, on sait qu'une numérotation infixée d'un arbre binaire complet de $N - 1$ sommets induit un plongement de dilatation 2 dans un hypercube de N sommets [16]. La figure 2.6 illustre ce plongement dans le cas $N = 16$. Dans ce plongement, le fils gauche d'un

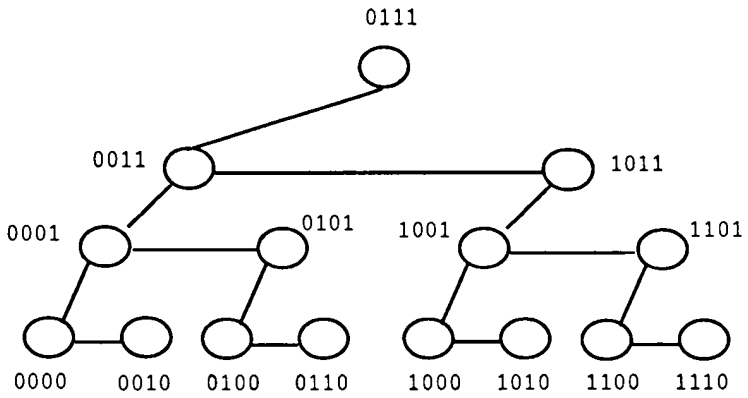

FIG. 2.6 - *Plongement de dilataion 2 d'un arbre binaire complet sur un hypercube.*

sommet interne est directement connecté à son père, tandis que le fils droit est connecté à son père via son frère le fils gauche, d'où la dilataion 2 du plongement. Les arêtes de la figure 2.6 sont les arêtes physiques de l'hypercube. Le processeur inutilisé 1111 connecté au processeur racine 0111 peut être considéré comme une racine auxiliaire qui peut servir à reçevoir la somme totale de tous les processeurs.

En utilisant ce plongement, on peut ainsi exécuter k opérations de calcul de préfixe en $O(k + log\ N)$ sur un hypercube de N processeurs en utilisant l'algorithme déstiné au départ à être mis en œuvre sur un arbre binaire complet.

Conclusion

Beaucoup de travaux ont été effectués sur le plongement statique d'arbres binaires complets dans différents réseaux. Citons par exemple ceux dans l'hypercube et ses variantes [81, 11, 113], le réseau hexagonal [33], la grille bidimensionnelle [57, 58, 63, 105, 129, 54] et le réseau pyramide [1]. D'autres travaux de plongements concernant les grilles sur différents réseaux peuvent être trouvés dans [37, 38, 121]. Dans [3], il est montré aussi comment plonger plusieurs réseaux dans un hypercube.

Dans les chapitres suivants, nous allons nous intéresser au départ, au problème de plongement d'arbres binaires complet dans une grille bidimensionnelles. Ensuite, nous allons étudier dans le reste du rapport, le problème de plongement d'arbres qui peuvent être dynamiques et quelconques.

Partie 2

Chapitre 3

Plongement statique dans une grille

3.1 Introduction

La grille bidimensionnelle est un des réseaux les plus utilisés comme topologie dans les machines parallèles commerciales. Citons par exemple la machine MasPar [93, 18, 108], et la machine Intel Paragon [119]. Un des avantages les plus évidents de la grille est le fait d'être le réseau le plus simple, parmi les réseaux connus, tout en permettant la mise en œuvre de nombreux algorithmes parallèles utiles dans les domaines de traitement d'images et du calcul matriciel avec ses nombreuses applications. En effet, l'utilisation des grilles, de par sa structure, est adaptée aux calculs scientifiques orientés vers les opérations vectorielles (manipulation de vecteurs, matrices ou images).

Comme nous l'avons signalé au premier chapitre, il est bien sûr souhaitable que la structure matérielle de la machine reflète aussi fidèlement que possible la structure du problème à traiter. Or, ce n'est pas toujours le cas, par exemple quand il s'agit de structures d'arbres et d'un réseau en grille, on est dès lors amené à contraindre le problème à se résoudre dans la topologie dont on dispose.

Les arbres forment une classe très importante en informatique grâce à leur propriété de chemin logarithmique de la racine vers une feuille quelconque. Comme structures de données, ils sont facilement manipulables ; les liens inter-sommets sont simples et réguliers. Comme graphes de contrôle, ils interviennent dans les algorithmes de type diviser pour régner. Ces algorithmes possèdent très souvent une structure instrinsèque en arbre dans laquelle les sommets représentent les tâches calculatoires et les arêtes les communications nécessaires aux calculs. Mettre en œuvre efficacement ces algorithmes sur une grille requiert donc un plongement efficace (au sens des paramètres présentés dans le chapitre 2) de l'arbre dans la grille.

Les arbres ont suscité beaucoup d'intérêt dans de nombreux travaux et de nombreux algorithmes efficaces ont été élaborés (voir [15, 98, 59, 130, 25, 4, 75, 111] pour exemples). Une machine dont la structure reflète la structure d'arbre a été également proposée [57, 98,

58]. *la machine arbre* est une architecture parallèle où les processeurs sont connectés pour former un arbre binaire complet. Des machines multiprocesseurs de la forme d'un arbre binaire ont été conçues telles que la machine X-tree [31], la machine DAC et la machine P-tree [57].

Dans ce chapitre, nous présentons divers algorithmes de plongements d'arbres binaires complets dans une grille bidimensionnelle. Ces plongements sont statiques puisque la forme de l'arbre est connue à l'avance. Notons que l'utilisation d'un arbre statique pour résoudre un problème donné est justifiée étant donné que l'on peut toujours ajuster la taille des tâches calculatoires affectées aux sommets selon la taille de l'arbre.

Le plongement d'arbres a été étudié sur différents réseaux tels que l'Hypercube [81, 135, 132, 11, 12, 113, 127], le réseau hexagonal [33], la grille [57, 58, 98, 63, 7, 6, 129, 104, 54] et le réseau pyramidal [1]. Le plongement d'un arbre binaire complet dans une grille 2D intéresse également la réalisation VLSI des réseaux des systèmes parallèles. En effet, pour le VLSI, la configuration d'arbre est très attractive grâce aux interconnexions simples et régulières nécessaires entre les sommets [98]. Chaque sommet communique seulement avec ses voisins immédiats, le coût de conception est donc fortement réduit en comparaison avec les réseaux à un nombre (ou degré) relativement large des liens entre processeurs comme les hypercubes. Pour une implémentation de l'arbre, économique en espace et en longueur des liens sur un chip VLSI, une stratégie de plongement de la structure arborescente sur une grille est donc nécessaire [57, 105].

Soit h un entier positif. L'arbre binaire complet de hauteur h, dénoté par T_h, est un arbre binaire composé de h niveaux numerotés de 1 à h et ayant $2^h - 1$ sommets. Le sommet au niveau h est appelé la *racine* et a un degré 2. Les sommets au niveau 1 sont les *feuilles.* Les sommets internes(non-feuilles) ont un degré 3. Chaque sommet interne au niveau i est connecté à son sommet père au niveau $i-1$ et à ses deux sommets fils au niveau $i+1$.

Il est évident que le meilleur plongement est celui dans lequel la dilatation, l'expansion, la congestion, et la charge sont faibles. Cependant, ces mesures ne sont pas toutes petites simultanément en général, et minimiser un critère peut obliger à en augmenter un autre. De tels compromis se rencontrent souvent dans les problèmes de plongement. Comme les plongements developpés ici sont statiques, on s'intéresse particulièrement à minimiser l'expansion, la dilatation et la charge. Les performances des algorithmes parallèles s'estiment en temps d'exécution et en nombre de processeurs utilisés. Réduire le nombre de processeurs et la longueur des liens utilisés tout en maintenant les mêmes performances sinon mieux est donc très utile. Rappelons que l'expansion est le rapport entre la taille de la grille et la taille de l'arbre que l'on plonge, la dilatation est la distance maximale dans la grille entre les sommets images de deux sommets adjacents dans l'arbre, et la charge est le nombre maximum de sommets de l'arbre placés dans un même sommet de la grille.

Le problème du plongement d'arbres binaires complets dans une grille bidimensionnelle, est abordé habituellement en cherchant à plonger l'arbre dans une grille optimale (i.e. ayant le plus petit nombre de sommets possible supérieur ou égal à celui de l'arbre) et en essayant de minimiser soit la dilatation soit la congestion. En effet, aucun des plongements construits ne permet de minimiser ces critères simultanément [20]. De par la structure hiérarchique des arbres, les plongements sont construits, en général, suivant une approche récursive; les sous-arbres sont plongés dans des sous blocs de la grille qui sont ensuite associés

pour construire l'arbre complet suivant un schéma donné [57, 58, 98, 33]. L'exemple d'un plongement suivant un schéma en H sera donné dans la section suivante. Dans la plupart des cas, les plongements sont injectifs; un sommet au plus de l'arbre est affecté à un sommet de la grille. L'expansion dans ces plongements est en conséquence supérieure à 1. Afin d'obtenir une plus grande efficacité en termes d'expansion et de dilatation, on s'est autorisé ici à placer plus qu'un sommet de l'arbre dans un sommet de la grille. Les plongements construits sont donc non-injectifs.

Il existe de nombreux algorithmes s'exécutant sur les arbres qui n'utilisent qu'un seul niveau de l'arbre à une étape donné et qui utilisent tous les niveaux de haut en bas ou de bas en haut (comme le calcul de préfixes en parallèle). Il existe aussi des algorithmes où les sommets et les arêtes de niveaux différents peuvent être actifs simultanément (par exemple, en pipelinant le calcul de préfixes en parallèle).

L'algorithme de plongement d'un arbre binaire complet doit viser à satisfaire au moins les conditions suivantes :

1. Les sommets de l'arbre appartenant à un même niveau doivent être projetés sur des sommets différents de la grille. Ainsi, tout traitement s'effectuant sur les sommets d'un même niveau s'effectue en une seule étape.

2. Les arêtes de l'arbre issues d'un même niveau doivent être projetées sur des arêtes disjointes de la grille et de même longueur. Ainsi, toute communication s'effectuant sur les arêtes du même niveau s'effectue en une seule étape.

3. La dilatation du plongement doit être minimisée afin de réduire le temps de communication entre les sommets de l'arbre plongé.

4. La taille de la grille hôte nécessaire doit être minimisée afin que la taille de l'arbre pouvant y être plongé soit la plus grande possible.

Dans le type de grilles que nous considérons ici, chaque sommet est connecté à ses 8 voisins immédiats. Cette grille est parfois appelée dans la littérature *grille étendue* (appelé en anglais *X-mesh* ou *X-Net*) [54, 101]. La figure 3.1 illustre un exemple d'une grille étendue 4×4. La machine massivement parallèle MasPar est un exemple de machine dotée d'une telle grille.

Notons par ailleurs que le réseau en grille appelé X-Net de la machine SIMD MasPar présente une certaine particularité : le X-Net est *un réseau uniforme*; tous les processeurs actifs envoient leur messages vers d'autres processeurs se situant à une distance et une direction uniformes. Ceci implique que dans ce réseau, si l'on veut que des communications s'effectuant sur les arêtes du même niveau de l'arbre se réalisent en une seule étape, il est alors souhaitable que ces arêtes de même niveau soient projetées suivant des chemins de même longueur et de directions uniformes dans la grille. Un plongement d'arbre dans un réseau en grille uniforme doit donc verifier les conditions (1) à (4). Les plongements étudiés ici seront mis œuvre sur le réseau X-Net de la MasPar afin de verifier leurs performances.

Nous débutons dans la section 5.3 en décrivant un des plongements les plus connus d'arbres binaires complets dans une grille bidimensionnelle, le plongement en H, qui vérifie les

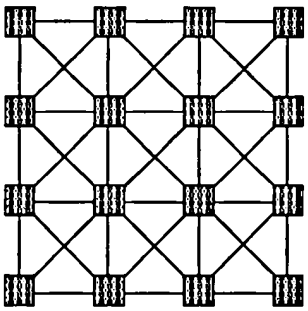

FIG. 3.1 - *Une grille étendue* 4×4

conditions (1)-(2). Nous avons choisi ce dernier pour cette raison. En effet, les autres plongements décrits dans la littérature ne vérifient pas les critères (1) à (4), et ils sont difficilement utilisables sur une architecture assez contraignante comme la grille uniforme, X-Net, de la MasPar. Dans la section 5.2, nous définissons ensuite la notion de contraction d'arbres binaires complets qui nous sera utile. Dans les sections suivantes, nous montrons comment plonger un arbre binaire contracté dans une grille en respectant les conditions (1) à (4). Ensuite, une comparaison analytique avec le plongement en H sera effectuée. Dans la section 5.4.1, nous allons décrire l'exemple d'un algorithme distribué de plongement. Dans la section suivante, mous appliquons les algorithmes du plongement en H et d'un plongement proposé pour mettre en œuvre un algorithme s'exécutant sur un arbre binaire complet sur le réseau X-net de la Maspar. Dans la section 3.12, nous mentionnons les résultats de M. Jiber et A.Bellaachia qui ont amélioré l'étude présentée ici dans le cadre de plongements d'arbres quaternaires dans les grilles étendues [101, 73, 102].

3.2 Le H-arbre

Un des plongements les plus connus d'arbres binaires complets dans une grille bidimensionnelle est le plongement en H. Ce plongement a été largement étudié et utilisé dans [57, 98, 58, 33, 4]. Il a été proposé pour la simulation des arbres dans un réseau de processeurs disposé en une grille et également pour l'implémentation VLSI de l'arbre dans un chip bidimensionnelle.

Ce plongement est obtenu de la manière récursive suivante. La racine de l'arbre est placée sur le sommet situé au centre de la grille. Le plongement d'un arbre T_h, où h est pair, est construit à partir de ses deux sous-arbres T_{h-1} en connectant leurs racines par un lien *vertical* de la grille. Le plongement de T_h, pour h impair, est construit à partir de ses deux sous-arbres T_{h-1} en connectant leurs racines par un lien *horizontal* de la grille. Le plongement se poursuit ainsi d'une manière récursive jusqu'à atteindre une unité de base. Cette dernière représente un arbre de hauteur3. La figure 3.2 illustre le plongement en H d'un arbre T_7 dans une grille. Notons que ce plongement verifie les conditions (1)-(2).

Soit S_h la taille de la grille nécessaire pour effectuer le plongement en H d'un arbre binaire complet de hauteur h (avec une charge égale à 1). On a [58, 33] :

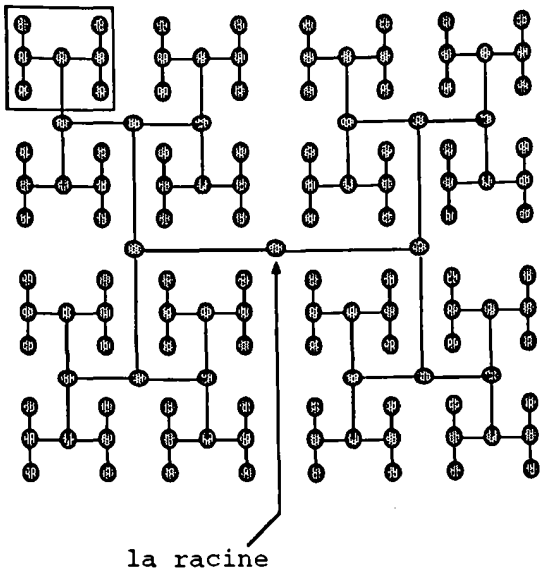

FIG. 3.2 - *Plongement en H d'un arbre T_7 sur une grille bidimensionnelle.La partie encadrée désigne l'unité de base qui représente un arbre de hauteur* 3.

$$S_h = \begin{cases} (2^{(h+1)/2} - 1)(2^{(h+1)/2} - 1) & \text{si } h \text{ est impair} \\ (2^{(h+2)/2} - 1)(2^{h/2} - 1) & \text{si } h \text{ est pair} \end{cases} \quad (3.1)$$

Etant donné qu'un arbre de hauteur h possède $2^h - 1$ sommets, un plongement qui consiste à affecter un sommet de l'arbre à un sommet de la grille nécessite une grille avec au moins 2^h sommets. La taille S_h est donc asymptotiquement deux fois cet optimum. La taille de la grille nécessaire pour plonger un arbre T_h est déterminée en comptant le nombre de sommets de la grille qui détiennent des sommets de l'arbre mais aussi les sommets inutilisés. Par exemple, le plongment en H illustré dans la figure 3.2 d'un arbre de hauteur 7 exige une grille 15×15.

Il est important de noter aussi que dans ce plongement, deux sommets adjacents quelconques de l'arbre sont placés sur des sommets de la grille connectés par un chemin horizontal ou vertical (i.e., empruntant toujours la même dimension). En effet, on peut utiliser le plongement en H tel que décrit, pour plonger les arbres binaires complets dans une grille étendue bidimensionnelle. Ceci implique cependant, d'une part, que les arêtes en X ne seront jamais empruntées, et d'autre part, que deux arêtes seulement sont utilisées au niveau de chaque sommet. Notons que la grille étendue est la grille classique augmentée d'arêtes en X. Dans ce qui suit, nons allons donc étudier des plongements permettant de bénéficier de ces arêtes ajoutées. Nous montrons également que ces plongements permettent d'obtenir une meilleure dilatation et une meilleure expansion que le plongement en H.

3.3 Définition du plongement $X(h_0, \ell, L)$

3.3.1 La contraction

L'idée sous-jacente aux plongements proposés ici consiste à contracter (ou compresser) l'arbre avant d'effectuer le plongement. Le but est d'utiliser les 4 voire les 8 liens de connexions disponibles entre les sommets de la grille. La contraction est définie de la manière suivante. Soit $G(V, E)$ un arbre binaire complet de $N = 2^h - 1$ sommets avec $h \geq 0$, V étant l'ensemble des sommets et E l'ensemble des arêtes.

Définition 1 *La contraction C de degré l de $G(V, E)$ est un $G'(V', E')$. Chaque sommet v'_i de V' remplace un sous arbre V_i de G. Chaque V_i a une hauteur $log_2 l$ et pour tout $i \neq j$, V_i et V_j sont disjoints. Le sommet v'_i est le père de v'_j s'il existe une arête dans E entre une feuille quelconque de V_i et la racine de V_j.*

Une contraction C d'un arbre binaire complet préserve la forme et la régularité de la topologie d'arbre. En effet, l'action d'une contraction consiste tout simplement à compresser l'arbre de départ, en compressant des sommets avec leurs pères. La figure 3.3 illustre l'exemple d'une contraction de degré 4 d'un arbre binaire complet. Cette contraction appliquée à un arbre de hauteur h, compresse les sommets du niveau $2i$ avec leurs sommets pères au niveau $2i - 1$, pour $1 \leq i \leq \lceil \frac{h}{2} \rceil$. La hauteur de l'arbre contracté obtenue est $\lceil \frac{h}{2} \rceil$ (soit $\frac{h}{2}$ si h est pair et $\frac{h+1}{2}$ si h est impair. Dans ce qui suit, nous allons nous intéresser seulement à la contraction de degré 4 pour construire nos plongements. Dans le reste du chapitre, h désignera toujours la hauteur de l'arbre avant contraction, T_h désigne un arbre binaire complet de hauteur h et $T_c(\lceil \frac{h}{2} \rceil)$ cet arbre contracté.

Notons aussi le point suivant, à titre d'exemple, les deux arbres de hauteurs respectives 5 et 6 conduisent à un arbre contracté de hauteur 3 (puisque $\lceil \frac{5}{2} \rceil = \lceil \frac{6}{2} \rceil = 3$). La différence se situe aux niveaux des feuilles de l'arbre contracté, si $h = 5$ alors les feuilles de $T_c(3)$ contiennent un sommet (i.e., les feuilles de $T(5)$). Si $h = 6$ alors les feuilles de $T_c(3)$ contiennent un sous arbre de hauteur 2 (i.e., les feuilles de $T(6)$ et leur pères).

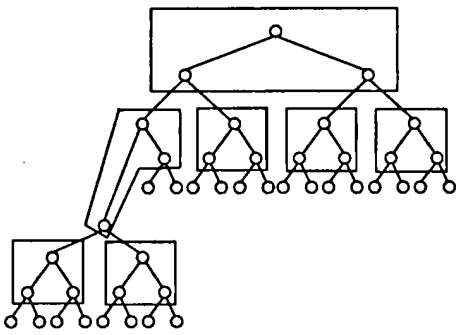

FIG. 3.3 - *Contraction de degré 4 d'un arbre binaire.*

3.3.2 Première approche de plongement

La statégie de plongement proposée ici comme une stratégie alternative du plongement en H des arbres binaires complets dans une grille a pour but de tirer parti au mieux des arêtes de connexions disponibles de la grille. Plus précisément, si l'on veut exploiter les 4 connexions disponibles au niveau des sommets de la grille, nous allons choisir un plongement qui convient aux arbres de degré 4 (i.e., chaque sommet possède 4 fils) d'où l'intérêt de la contraction décrite ci-dessus.

Le plongement d'un arbre contracté est mis en œuvre de la manière suivante. La racine est plongée dans un sommet situé au centre de la grille. Le plongement est construit en connectant les 4 exemplaires de plongement de ses 4 sous-arbres, séparés par une ligne et une colonne de la grille. Le plongement s'effectue ainsi d'une manière récursive. La figure 3.4.(a) illustre ce schéma de plongement, R désigne la racine de l'arbre contracté et les r_i, $1 \leq i \leq 4$, désignent les racines de ses 4 sous-arbres. Rappelons-que le sommets R et r_i représentent des sous arbres de hauteur 2 de l'arbre de départ (si les r_i sont des feuilles, les hauteurs des sous arbres qu'elles contiennent peuvent être de hauteur 2 ou 1 suivant si l'arbre de départ est pair ou impair). La figure 3.4.(b) montre le plongement d'un arbre contracté $T_c(3)$. Notons que cet arbre contracté peut résulter de la contraction d'un arbre $T(5)$ ou $T(6)$.

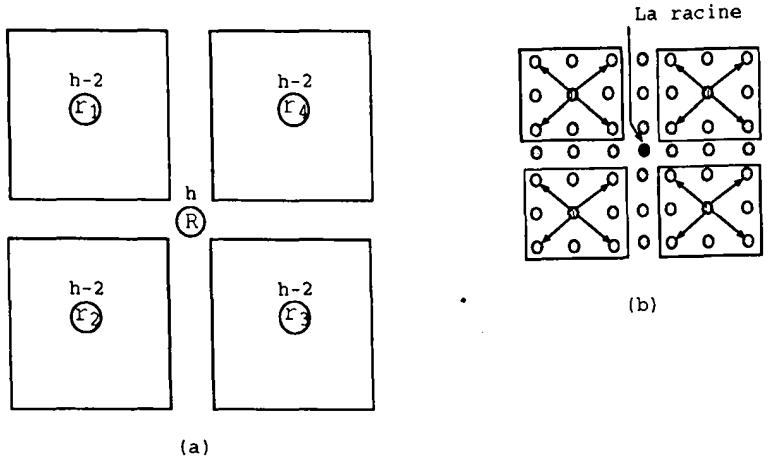

FIG. 3.4 - *(a) Le plongement d'un arbre contracté $T_c(\lceil \frac{h}{2} \rceil)$ (i.e., $T(h)$) est construit d'une manière recursive à partir des 4 exemplaires de plongement de ses 4 sous-arbres $T_c(\lceil \frac{h}{2} \rceil - 1)$ (i.e., $T(h-2)$). La racine est plongée dans un sommet situé au centre de la grille. Les étiquettes au dessus des sommets indiquent les niveaux dans l'arbre, avant contraction, des racines des sous arbres contenus dans R et r_i. (b) Ce schéma illustre le plongement d'un arbre $T_c(3)$.*

Lemme 1 *Soit Q_h la taille de la grille nécessaire à ce type de plongement pour plonger*

un arbre de hauteur h. On a

$$Q_h = \begin{cases} (2^{(h+1)/2}-1)(2^{(h+1)/2}-1) & \textit{si } h \textit{ est impair} \\ (2^{h/2}-1)(2^{h/2}-1) & \textit{si } h \textit{ est pair} \end{cases}$$

preuve

Soit la grille $M(h) \times M(h)$ nécessaire pour effectuer le plongement d'un arbre de hauteur h. Le plongement est effectué récursivement à partir de 4 exemplaires du plongement de l'arbre de hauteur $h-2$. On a donc

$$M(h) = 2.M(h-2) + 1$$

ce qui implique que pour $k \geq 1$, on a

$$M(h) = 2^k.M(h-2k) + 2^k - 1$$

Pour k pair, posons $k = \frac{h-2}{2}$. Puisque que $M(2) = 1$, on obtient

$$M(h) = 2^{\frac{h}{2}} - 1$$

De même, pour k impair, on pose $k = \frac{h-1}{2}$. Puisque que $M(1) = 1$, on a donc

$$M(h) = 2^{\frac{h+1}{2}} - 1.$$

3.3.3 Le plongement amélioré

Dans le plongement que l'on vient de décrire, beaucoup de sommets de la grille sont inutilisés. En effet, quand h est impair, la taille de la grille nécessaire pour ce plongement est identique à la taille nécessaire pour effectuer un plongement en H. A titre d'exemple, le plongement d'un arbre de hauteur 7 ayant 127 sommets exige une grille 15×15. Ainsi, seulement 56% des sommets de la grille sont utilisés.

Rappelons que le plongement d'un arbre contracté $T_c(\lceil \frac{h}{2} \rceil)$ (i.e., $T(h)$) est construit d'une manière recursive à partir des 4 exemplaires de plongement de $T_c(\lceil \frac{h}{2} \rceil - 1)$ (i.e., $T(h-2)$) jusqu'à atteindre une unité de base. Cette unité de base désignera le plongement de l'arbre contracté $T_c(2)$ (voir les parties encadrées de la figure 3.4). Dans ce qui suit, on désignera par h_0 la hauteur de l'arbre représenté par l'unité de base avant sa contraction. Dans le cas de la figure 3.4, h_0 vaut 4. Notons que l'unité de base peut être configurée différement, comme nous allons le voir dans la section 3.12

Afin d'éviter de laisser trop de processeurs inutilisés, nous allons modifier le plongement décrit dans la section précedente. Ce plongement d'une manière générale est illustré dans la figure 3.5.(a). En effet, nous allons éviter de laisser la ligne et la colonne qui séparent les unités de base, et nous plaçons la racine de l'arbre $T_c(3)$ dans un des sommets inutilisés comme cela est illustré dans la figure 3.5.(b). Nous donnerons un peu plus loin l'algorithme distribué correspondant à ce plongement.

Dans le reste de ce chapitre, nous noterons un plongement effectué suivant la méthode que l'on vient de décrire par plongement en $\mathrm{X}(h_0, \ell, L)$, h_0 étant la hauteur de l'arbre (avant sa contraction) plongé dans une unité de base de taille $\ell \times L$. Dans les sections suivantes, nous allons effectuer une comparaison analytique avec le plongement en H.

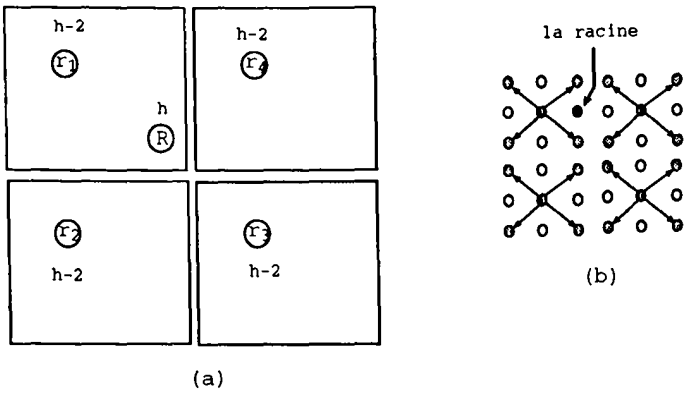

FIG. 3.5 - *(a).Le plongement d'un arbre contracté est construit d'une manière recursive à partir des 4 exemplaires du plongement de ses sous arbres jusqu'à atteindre l'unité de base à la hauteur* $\lceil \frac{h_0}{2} \rceil$. *(b).Ce schéma illustre le plongement d'un arbre* $T_c(3)$.

3.4 Analyse de l'expansion

Dans cette section, nous allons présenter les expressions analytiques correspondantes à l'espace occupé par un plongement en $X(h_0, \ell, L)$, et que l'on comparera à celles du plongement en H. La taille de la grille nécessaire pour plonger un arbre T_h est déterminée en comptant le nombre de sommets de la grille qui détiennent des sommets de l'arbre mais aussi les sommets inutilisés.

Lemme 2 *Soit* P_h *la taille de la grille nécessaire pour plonger un arbre de hauteur* h *selon un plongement en* $X(h_0, \ell, L)$. $\ell \times L$ *est la taille de la grille utilisée par l'unité de base choisie. On a:*

$$P_h = (\ell \times 2^{\lceil \frac{h-h_0}{2} \rceil})\,(L \times 2^{\lceil \frac{h-h_0}{2} \rceil})$$

preuve.

Le plongement d'un arbre $T(h)$ de hauteur h, dans une grille $M(h) \times N(h)$ est effectué récursivement à partir de 4 exemplaires de plongement de ses sous-arbres $T(h-2)$ de hauteur $h-2$ (voir la figure 3.5). Nous avons donc

$$M(h) = 2.M(h-2),$$
$$N(h) = 2.N(h-2)$$

ce qui implique que pour $k \geq 1$, on a

$$M(h) = 2^k.M(h-2k),$$
$$N(h) = 2^k.N(h-2k)$$

Pour k pair, posons $k = (h-h_0)/2$. Puisque que $M(h_0) = \ell$ et $N(h_0) = L$, on a donc

$$P_h = M(h) \times N(h) = (2^{(h-h_0)/2} \times \ell)(2^{(h-h_0)/2} \times L).$$

De même, pour k impair, on pose $k = (h - (h_0 - 1))/2$. Puisque que $M(h_0 - 1) = \ell$ et $N(h_0 - 1) = L$, on a donc

$$P_h = M(h) \times N(h) = (2^{(h-(h_0-1))/2} \times \ell)(2^{(h-(h_0-1))/2} \times L).$$

A titre d'exemple, dans le plongement illustré dans la figure 3.5 dans lequel $\ell = L = 3$, nous avons $P_h = 9 \times 2^{h-4}$ quand h est pair et $9 \times 2^{h-3}$ quand h est impair.

Calculons l'expansion exp_X d'un plongement en $X(h_0, \ell, L)$. Rappelons que l'expansion est le rapport entre la taille de la grille et la taille de l'arbre que l'on plonge. Etant donné que le nombre de sommets d'un arbre binaire complet de hauteur h est $2^h - 1$, alors l'expansion du plongement en X est

$$exp_X = \begin{cases} \dfrac{\ell \times L \times 2^{h-h_0}}{2^h - 1} & \text{si } h \text{ est impair} \\ \dfrac{\ell \times L \times 2^{h-h_0+1}}{2^h - 1} & \text{si } h \text{ est pair} \end{cases}$$

On déduit donc que l'expansion est asymptotiquement $(\ell \times L)/2^{h_0}$ pour h pair et $(\ell \times L)/2^{h_0-1}$ pour h impair. Pour $\ell = L = 3$ par exemple (c'est le cas du plongement illustré dans la figure 3.5), l'expansion du plongement est 16/9 quand h est pair, et 8/9 quand h est impair.

Comparons le plongement en H et le plongement en X en terme de taille de grille nécessaire pour effectuer le plongement d'un arbre $T(h)$. En reprenant l'expression (3.1) de S_h et l'expression P_h donnée dans le lemme 2, on déduit que si la hauteur h est paire, la taille de la grille nécessaire pour plonger l'arbre en X est asymptotiquement $(\ell \times L)/2^{h_0+1}$ (on calcule le rapport P_h/S_h). Dans le cas où la hauteur h est impaire, la taille nécessaire pour plonger l'arbre en X est asymptotiquement $(\ell \times L)/2^{h_0}$ celle du plongement en H.

A titre d'exemple, la taille de la grille nécessaire pour plonger un arbre $T(h)$ en $X(4, 3, 3)$ (i.e., le plongement illustré dans la figure 3.5) est asymptotiquement 9/32 celle nécessaire au plongement en H puisque . Dans le cas où la hauteur h est impair, la taille de la grille nécessaire est asymptotiquement 9/16 celle nécessaire pour le plongement en H. On déduit donc que le plongement en $X(4, 3, 3)$ est de 2 à 4 fois plus économique que le plongement en H en ce qui concerne la taille de la grille nécessaire pour effectuer le plongement.

3.5 Analyse de la dilatation

Nous avons comparé les deux plongements en terme de taille de grille utilisée. Nous allons donné ici les dilatations de ces plongements. La dilation d'un plongement correspond à l'étirement maximum des arêtes de l'arbre sur des chemins dans la grille. Rappelons que selon la condition (2), les arêtes de l'arbre issues d'un même niveau doivent être projetées sur des arêtes disjointes et de même longueurs de la grille. Ainsi, toute communication s'effectuant sur les arêtes du même niveau s'effectue en une seule étape. Il est important de noter aussi que le plongement en X est effectué selon la stratégie du "Plus loin d'abord" (comme c'est le cas du plongement en H). En effet, l'étirement le plus long correspond à celui des arêtes entre la racine est ses 4 fils. En conséquence, la dilatation du plongement

en X d'un arbre $T(h)$ de hauteur h est la longueur du chemin séparant la racine de l'arbre contracté à un de ses fils racine de $T(h-2)$.

Lemme 3 *La dilatation du plongement en $X(h_0, \ell, L)$ est*

$$D_h^X = \begin{cases} \ell \times 2^{\lceil \frac{h-h_0}{2} \rceil - 2} & si\ \ell = L \\ Sup(\ell, L) \times 2^{\lceil \frac{h-h_0}{2} \rceil - 2} & sinon \end{cases}$$

preuve

Soient $M(h) \times N(h)$ la taille de la grille nécessaire pour plonger un arbre de hauteur h. Pour simplifier les écritures, nous supposons que h est pair. On sait que $M(h) = \ell \times 2^{\frac{h-h_0}{2}}$ et $N(h) = L \times 2^{\frac{h-h_0}{2}}$. Le sommet au centre de la grille est situé à la position $(cx_h = M(h)/2, cy_h = N(h)/2)$. Soit R_h la racine de l'arbre binaire de hauteur h que l'on plonge dans cette grille. la position du sommet auquel est assigné R_h peut être déterminée par rapport au centre (cx_h, cy_h), soit $(\alpha_x + cx_h, \alpha_y + cy_h)$, α_x et α_y étant des entiers positifs ou négatifs indépendants de h. Rappelons que le plongement est effectué d'une manière récursive jusqu'à atteindre l'unité de base de hauteur h_0. R_h possède donc les coordonnées $(x_h = \alpha_x + \ell \times 2^{\frac{h-h_0}{2}-1},\ y_h = \alpha_y + L \times 2^{\frac{h-h_0}{2}-1})$. La dilatation est la distance séparant R_h et R_{h-2} (voir la figure 3.6). On obtient donc

$$x_h - x_{h-2} = \ell \times 2^{\frac{h-h_0}{2}-1} - \ell \times 2^{\frac{h-2-h_0}{2}-1} = \ell \times 2^{\frac{h-h_0}{2}-2}$$
$$y_h - y_{h-2} = L \times 2^{\frac{h-h_0}{2}-1} - \ell \times 2^{\frac{h-2-h_0}{2}-1} = L \times 2^{\frac{h-h_0}{2}-2}$$

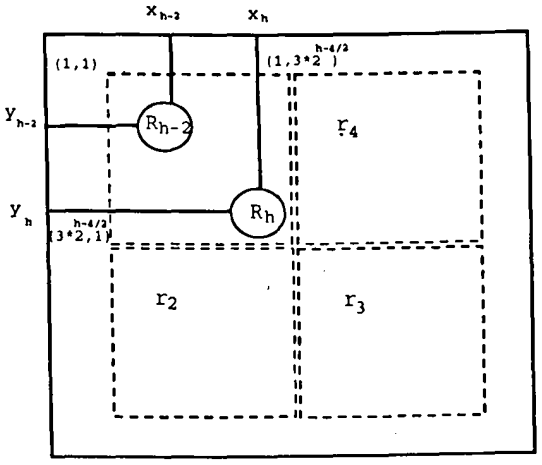

FIG. 3.6 - *La dilatation du plongement est la distance séparant R_h et R_{h-2}. Le plongement donné ici est le plongement en $X(4,3,3)$ de la figure 3.5.*

Si $\ell = L$, la distance entre les deux sommets R_h et R_{h-2} qui sont situés sur une même diagonale est la différence des coordonnées $x_h - x_{h-2}$, soit $\ell \times 2^{\frac{h-h_0}{2}-2}$. Si en revanche

$\ell \neq L$, alors la distance est égale à $sup(\ell, L) \times 2^{\frac{h-h_0}{2}-2}$. En effet, notons que calculer la distance entre les points R_h et R_{h-2} revient à calculer le diamètre d'une grille étendue dont R_h et R_{h-2} sont des sommets.

On déduit donc que la dilatation du plongement en X(h_0, ℓ, L) pour $h > h_0 + 2$ est

$$D_h^X = Sup(\ell, L).2^{\lceil \frac{h-h_0}{2} \rceil - 2}.$$

Dans le cas pour lequel $h \leq h_0+2$, le plongement est effectué en associant 4 unités de base, soit une grille $2\ell \times 2L$. En conséquence, le chemin le plus long est majoré par le diamètre de la grille étendue $2\ell \times 2L$. On déduit donc que la dilataion est majorée par $sup(2\ell, 2L)$ pour $h = h_0 + 1$ et $h = h_0 + 2$.

De même, pour $1 \leq h \leq h_0$, la grille se réduit à l'unité de base, la dilatation est donc $sup(\ell, L)$. Dans ce qui suit, nous supposons que h est toujours suérieure à h_0 pour simplifier les écritures.

La dilatation du plongement en H, notée par D_h^H, d'un arbre $T(h)$ de hauteur h est la suivante [58, 33]

$$D_h^H = \begin{cases} 2^{(h-2)/2} & si\ h\ pair \\ 2^{(h-3)/2} & si\ h\ impair \end{cases}$$

Dans le cas du plongement X$(4,3,3)$, la dilatation D_h^X du plongement d'un arbre de hauteur h selon le schéma illustré dans la figure 3.5 est la suivante

$$D_h^X = \begin{cases} 3.2^{(h-8)/2} & si\ h\ pair \\ 3.2^{(h-7)/2} & si\ h\ impair \\ 4 & si\ h = 5\ ou\ h = 6 \\ 1 & si\ h \leq h_0 = 4 \end{cases}$$

On déduit donc que la dilatation dans le plongement de l'arbre compressé est 3/8 inférieure à celle du plongement en H quand k est pair, est 3/4 inférieure quand k est impair (nous avons calculé le rapport D_h^X/D_h^H). Il est donc plus interessant de plonger l'arbre en X plutôt que de le plonger en H sur une grille bidimensionnelle.

3.6 Analyse de la congestion

La méthode de plongement en X présentée ci-dessus satisfait la condition (2) sur la construction des plongements. En effet, les arêtes de l'arbre connectant un niveau i au niveau $i-1$ ne sont jamais "étirés" sur un même chemin de la grille. Ceci implique que toute communication s'effectuant sur les arêtes du même niveau s'effectue en une seule étape et sans être retardée. La congestion limitée aux communications entre nœuds successifs de l'arbre de même niveau est égale à 1. En fait, pour construire les plongements, on s'est plus soucié d'optimiser la dilatation et l'expansion. Il faut noter cependant que l'impact de la dilatation et de la congestion dépend du modèle de commmunication utilisée sur la grille. En effet, il existe différents modèles de routage dans les machines parallèles. Un des modèles couramment rencontré sur les machines parallèles est le modèle de routage

par *commutation de messages*, appelé aussi *commutation de paquets* ou encore *store-and-forward*. Dans ce modèle, chaque message ou paquet est transmis de sommet en sommet jusqu'à sa destination et à chaque étape, le lien emprunté est aussitôt libéré. En conséquence, le temps de communication dépend essentiellement de la distance entre la source et la destination. Le réseau en grille de la machine Maspar utilise le modèle de routage *store-and-forward* [77]. Dans une autre technique de routage selon un protocole en série appelé *wormhole*, les messages sont divisés en flits qui progressent dans le réseau de sommet en sommet et un sommet intermediaire fait avancer le message reçu sans attendre de le recevoir entièrement. Dans un autre modèle de routage appelé modèle de *commutation de circuits*, un circuit physique est établi entre la source et la destination et ce circuit est reservé pour cette communication. Dans ce modèle de commutations de circuits ainsi qu'avec le routage wormhole, le temps de communication dépend plus de la longueur du message et l'impact de la distance entre la source et la destination est moins important en absence de contention sur le lien de communication. Dans ce cas, la congestion devient un paramètre important.

Nous avons choisi ici d'optimiser d'abord la dilatation, et de maintenir une congestion optimale (i.e, égale à 1) pour les communications entre niveaux successifs. Dans le plongement en H, la congestion est toujours égale à 1.

Lemme 4 *La congestion du plongement en $X(h_0, \ell, L)$ est majorée par $\lceil \frac{h-h_0}{2} \rceil - 2 + log\ L$, en supposant que $L = sup(\ell, L)$.*

preuve

Le plongement d'un arbre $T(h)$ est construit d'une manière recursive à partir des 4 exemplaires du plongement de $T(h-2)$ en connectant leurs racines jusqu'à atteindre la racine de l'arbre T_{h_0+2} (à un niveau de plus par rapport à l'unité de base). Le plongement continue ensuite en connectant la racine de T_{h_0+2} aux racines de 4 exemplaires d'unité de base de hauteur h_0. Les arêtes de l'arbre connectant un niveau i au niveau $i-1$ ne sont jamais "étirés" sur un même chemin de la grille. Comme le plongement est basé sur la stratégie du "le plus loin d'abord", la dilatation D_h du plongement est la longueur du chemin qui lie la racine de l'arbre contracté à l'un de ses sous arbres et est égale à $Sup(\ell, L) \times 2^{\lceil \frac{h-h_0}{2} \rceil - 2}$. L'une des arêtes e appartenant à ce chemin est donc empruntée au plus $log\ D_h$ fois, soit $\lceil \frac{h-h_0}{2} \rceil - 2 + log\ L$ fois, en supposant que $L = sup(\ell, L)$.

3.7 Analyse de la charge

Etant donné que le plongement en H est un plongement injectif, la charge vaut 1. Dans le plongement proposé la charge vaut 3 puisqu'on plonge l'arbre contracté. Notons cependant que la charge est équilibrée partout et vaut 3 si h est pair. Si h est impair, la charge est équilibrée sauf au niveau des feuilles. Il est important de noter que la charge du plongement en X est une constante alors que le gain en expansion est asymptotique et dépend de la hauteur de l'arbre à plonger.

3.8 Bilan de l'analyse

Le théorème suivant résume les résultats des analyses que l'on vient d'effectuer dans les sections précédentes:

Theorème 1 *Soit arbre binaire complet de* $N = 2^h - 1$ *sommets. Un plongement en* $X(h_0, \ell, L)$ *plonge cet arbre si h est pair (resp. impair) dans une grille de* $\frac{\ell \times L}{2^{h_0}}(N+1)$ *sommets (resp.* $\frac{\ell \times L}{2^{h_0-1}}(N+1)$*) avec une charge constante, une dilatation* $\frac{L}{\sqrt{2^{h_0+4}}}\sqrt{N+1}$ *(resp.* $\frac{L}{\sqrt{2^{h_0+3}}}\sqrt{N+1}$ *)et une congestion* $\lceil \frac{log(N+1)-h_0}{2} \rceil + log\, L - 2$*, en supposant que* $sup(\ell, L) = L$.

3.9 L'algorithme distribué du plongement

Le plongement de l'arbre sur la grille bidimensionnelle est mis en œuvre par une distribution de messages de la forme $M(k, m)$, où k est le niveau dans l'arbre contracté, et m la longueur du chemin dans la grille correspondant à l'arête entre deux sommets successeurs de l'arbre contracté. Chaque sommet de la grille peut participer au traitement si des sommets de l'arbre y sont plongés ou il peut servir comme un sommet de connexion qui fait progresser des messages sans exécuter aucun traitement. Un sommet qui reçoit un message détermine les directions vers les sommets contenant les racines de ses sous-arbres et envoie les messages de sorties dans ces directions. Une direction d est choisie entre les 8 directions possibles de l'ensemble $\{N, S, E, W, NE, SE, NW, SW\}$. L'algorithme donné ci-dessous est l'algorithme distribué de plongement d'un arbre contarcté selon l'exemple illustré dans la figure 3.5. Cet algorithme est initialisé par la transmission d'un message $M(h, 3 \times 2^{k-4})$ à partir du sommet situé au centre de la grille et auquel est assignée la racine dans les directions nord-est, nord-west, sud-est et sud-west. La valeur $3 \times 2^{k-4}$ est obtenue à partir de la dilatation du plongement en X(4, 3, 3) qui est égale à $3 \times 2^{\lceil \frac{h}{2} \rceil - 4}$ puisque $k = \lceil \frac{h}{2} \rceil$ est la hauteur de l'arbre contracté (voir la section 3.5).

`Un message M(k,m)` *est reçu depuis le voisin selon la dimension d*

`si` $m \neq 0$ *alors { C'est un sommet de connexion }*
transmettre M(k,m-1) au voisin selon la direction d {Les lignes sont directes, donc pas de changement de direction}

`sinon` *{ C'est un sommet de traitement qui contient un sommet de l'arbre contracté}*

`si` $k > 3$ *alors*
$m = 3 \times 2^{k-4}$
transmettre(k-1,m) au voisin NE
transmettre(k-1,m) au voisin SE
transmettre(k-1,m) au voisin SW
transmettre(k-1,m) au voisin NW

`sinon si` $k = 3$ *{C'est le sommet de l'arbre qui connecte les 4 unités de base}*
transmettre2(k-1,0) au 4 fils au niveau 2 {i.e., vers les racines des sous-arbres

plongés dans les 4 unités de base}

```
    sinon si  k = 2
        transmettre(1,0) au voisin NE
        transmettre(1,0) au voisin SE
        transmettre(1,0) au voisin SW
        transmettre(1,0) au voisin NW
    finsi
finsi
```

La procédure *transmettre2* invoquée dans l'algorithme ci-dessus se charge de router les messages depuis la racine de l'arbre contracté au niveau 3 vers ses 4 fils. Ces derniers sont les racines des sous-arbres plongés dans les unités de base. Les instructions de cette procédure dépendent de la manière avec laquelle on a configuré le plongement de $T_c(3)$, comme cela est illustré dans la figure 3.5.

3.10 Le temps de propagation

En plus des mesures des plongements de graphe, une mesure appelée *le temps de propagation* est souvent étudiée dans le cas des plongements des arbres binaires complets [98, 63, 33]. Cette mesure détermine la longueur du lien le plus long du plongement en comptant le nombre de sommets à traverser depuis la racine jusqu'à la feuille la plus éloignée de l'arbre. Ce temps va être apprécié en étudiant le temps de communication de l'algorithme de test que nous allons abordé dans la section suivante.

Soient R_h et T_h le temps de propagation de la racine aux feuilles respectivement dans le plongement en H et le plongement en X(h_0, ℓ, L), pour un arbre $T(h)$.

D'après l'analyse de la section 3.5, on sait que la dilataion $D_h^X = sup(\ell, L) \times 2^{\lceil \frac{h-h_0}{2} \rceil - 2}$ pour $h > h_0 + 2$. Pour simplifier les écritures, On suppose que $sup(\ell, L) = L$ et que h est pair et est supérieure à $h_0 + 2$.

Soient $k = \frac{h}{2}$ la hauteur de l'arbre contracté $T_c(k)$ obtenu à partir de la compression de l'arbre $T(h)$, et k_0 la hauteur de l'arbre compressé contenu dans une unité de base. La réecriture de l'expression de la dilation D_h^X en fonction de k donne

$$D_k^c = L \times 2^{k-k_0-2}$$

D_k^c correspond donc à la dilatation du plongement de l'arbre compressé $T_c(k)$.

Calculons la distance maximale, appelé T_k^c, qui sépare la racine des feuilles dans un arbre contracté.

Pour un arbre contracté, la distance maximale qui sépare la racine des feuilles est égale à la somme de la distance maximale entre la racine de $T_c(k)$ et la racine de $T_c(k-1)$ (i.e., la dilatation), et la distance maximale qui sépare la racine de $T_c(k-1)$ des ses feuilles. On obtient donc la relation de récurrence suivante

$$
\begin{aligned}
T^c_k &= T^c_{k-1} + D^c_k \\
T^c_{k-1} &= T^c_{k-2} + D^c_{k-1} \\
&\cdot \\
&\cdot \\
&\cdot \\
T^c_{k_0+1} &= T^c_{k_0} + D^c_{k_0+1}
\end{aligned}
$$

On a donc $T^c_k = T^c_{k_0} + D^c_{k_0+1} + \ldots + D^c_k$.

Puisque $\sum_{i=k_0+1}^{k} D^c_i = L \times 2^{k-k_0-1} - \frac{L}{2}$ et $T^c_{k_0}$ est une constante majorée par L

on obtient $T^c_k = L \times 2^{k-k_0-1} + \alpha$.

α étant une constante indépendante de k.

Puisque $k = \frac{h}{2}$, on a alors $T_h = L \times 2^{\frac{[h-h_0]}{2}-1} + \alpha$

Le temps de propagation dans le plongement en H est [57]:

$$R_h = \begin{cases} 3 \times 2^{(h-2)/2} - 2) & si\ h\ pair \\ 2^{(h+1)/2} - 2) & si\ h\ impair \end{cases}$$

Le rapport T_h/R_h est asymptotiquement $\ell/(3 \times 2^{h_0/2})$, pour h pair, et $\ell/(3 \times 2^{(h_0-1)/2})$ pour h impair.

A titre d'exemple, le temps de propagation dans le cas du plongement en X(4, 3, 3) de l'exemple 3.5 est donc

$$T_h = \begin{cases} 3 \times 2^{(h-6)/2} + \alpha & si\ h\ pair \\ 3 \times 2^{(h-5)/2} + \alpha & si\ h\ impair \end{cases}$$

Le rapport T_h/R_h est donc asymptotiquement 1/4, pour h pair, et 3/8 pour h impair. Le temps de propagation dans le plongement en X(4, 3, 3) est donc inférieur au temps de propagation dans le plongement en H.

Le temps de propagation dans le plongement en $X(h_0, \ell, L)$ est donc inférieur au temps de propagation dans le plongement en H.

3.11 Parallélisme et performances

Il existe de nombreux algorithmes s'exécutant sur des arbres et où seuls les sommets et les arêtes d'un même niveau de l'arbre sont actifs à chaque étape. Ces algorithmes peuvent posséder en plus la propriété que des niveaux consécutifs sont utilisés à des étapes consécutives comme c'est souvent le cas [75, 4, 81]).

Dans cette section, nous allons étudier les performances d'un algorithme s'exécutant sur un arbre binaire complet implémenté en utilisant le plongement en H et un plongement

en X. Cet algorithme simple consiste à générer un arbre binaire complet pour calculer la puissance de 2 d'un entier n donné [47]. Cet algorithme calcule la fonction suivante :

$\text{puissance}_2(0) = 1$
$\text{puissance}_2(n) = \text{puissance}_2(n-1) + \text{puissance}_2(n-1)$

L'algorithme s'exécute donc en deux phases. La première phase pour diffuser les valeurs à partir de la racine à tous les sommets de l'arbre jusqu'aux feuilles. La deuxième phase consiste à remonter les valeurs en calculant leurs sommes. A la fin de l'algorithme, la racine contiendra le resultat de l'opération.

Nous avons appliqué le plongement en H et le plongement en X avec h_0=4 et $\ell = L = 3$ (l'exemple du plongement de la figure 3.5 pour mettre en ouvre l'algorithme simple décrit ci-dessus sur la machine massivement parallèle SIMD MasPar MP-1. Les processeurs sont physiquement connectés par un réseau d'interconnexion d'une topologie en grille étendue appelé X-Net. Il est important de noter que le X-Net est *un réseau uniforme*; tous les processeurs actifs envoient leur messages vers d'autres processeurs se situant à une distance et une direction uniformes. Le réseau X-Net utilise le modèle de routage store-and-forward [77]. Les processeurs inactifs peuvent servir en étages de pipeline pour des expéditions à longues distances traversant plusieurs proceseurs [108].

Si l'algorithme est donc mis en œuvre sur un arbre implanté en H sur le réseau uniforme X-Net de la MasPar, le temps de propagation de la valeur de la racine vers les feuilles doit être multiplié par 2 puisqu'à chaque étape, deux communications sont nécessaires (verticales ou horizontales mais opposées). Dans le plongement proposé, le temps de communication doit être multiplié par 4 étant donné qu'à chaque étape, quatre communications diagonales sont nécessaires. De plus la charge de calcul des processeurs dans notre plongement est 3 fois celle due au H-arbre étant donné qu'il y a 3 sommets par processeur.

Notons par ailleurs que dans le réseau X-Net de la MasPar, une communication orthogonale, entre un processeur est un de ces quatre voisins (au nord, sud, est et ouest) est plus rapide qu'une communication diagonale entre le processeur est un de ces quatre voisins au nord-est, sud-est, sud-ouest et nord-ouest (voir la figure 3.7). Les liens physiques diagonaux sont donc plus longs que les liens physiques orthogonaux, ce qui remet en cause l'hypothèse selon laquelle une comunication d'un processeur avec un de ses 8 voisins coûte une unité de temps. Ce problème n'affectera que peu les performances du plongement en X par rapport au plongement en H.

La figure 3.8 donne le temps d'exécution de l'algorithme décrit ci-dessus implanté en utilisant les deux plongements sur un arbre en fonction de sa hauteur. Les performances sont meilleures quand la hauteur est paire (la hauteur est limitée ici à 13 puisqu'on dispose d'une grille $2^7 \times 2^7$).

Rappelons aussi que les performances des algorithmes parallèles s'estiment en temps d'exécution et en nombre de processeurs utilisés. Réduire le nombre de processeurs et la longueur des liens utilisés tout en maintenant les même performances sinon mieux est donc très utile. Nous avons montré dans les sections précédentes que par rapport au plongement en H, le plongement en $X(4,3,3)$ est plus économique en nombre de processeurs utilisés, et les longueurs des liens sont inférieurs. On peut donc conclure que le plongement en $X(4,3,3)$

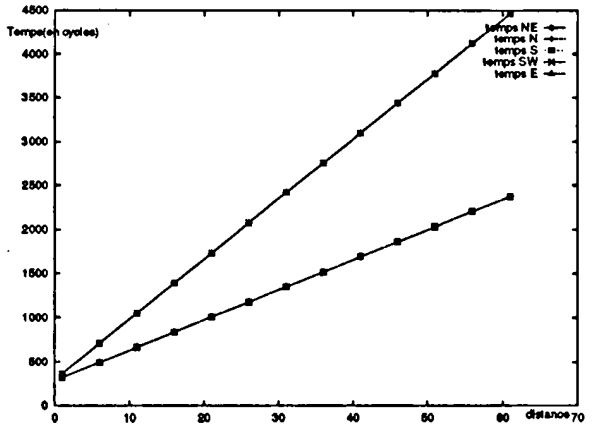

FIG. 3.7 - *Comportement du X-Net suivant la distance et la direction de la communication. Les communications selon une direction en diagonale (NE, NW, SE, SW) sont plus lentes que les communications selon une direction orthogonale (N, E, S, W). Les quatre droites de chaque cas se trouvent confondues dans la figure.*

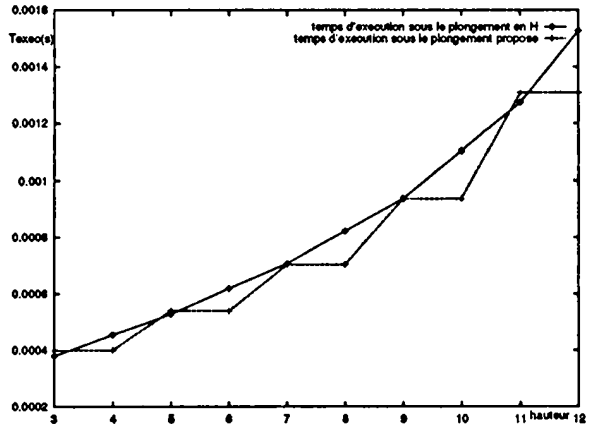

FIG. 3.8 - *Temps d'exécution en secondes de l'algorithme implanté avec les deux plongements en H et en X(3,4,4) en fonction de la hauteur de l'arbre.*

est meilleur que le plongement en H.

3.12 Les améliorations du plongement en X

Le plongement en X peut servir aussi à plonger des arbres quaternaires (i.e., des arbres à 4 fils). Les arbres quaternaires sont des structures de données très utilisées dans les traitements d'images non seulement parce qu'elles ne retiennent que l'information utile dans une image et donc économiques en espace mais également parce que toutes les opérations à effectuer ne nécessite pas de transformer l'image originelle [28]. Les arbres quaternaires sont

aussi utilisés dans la géométrie algorithmique (les méthodes de segmentation en régions) [60, 28].

La méthode de plongement en X a été améliorée par M.Jiber et A.Bellaachia dans [101, 102, 73] dans le cadre de plongement d'arbres quaternaires. L'unité de base a été modifiée en réduisant sa taille. Plus précisément, les valeurs de ℓ et L ont été modifiées pour réduire le nombre des processeurs inutilisés [101]. Nous avons d'ailleurs réécrit ce chapitre en fonction de ces améliorations en paramétrant l'unité de base par h_0, ℓ et L.

Rappelons que la notation $X(h_0, \ell, L)$ désigne un plongement d'arbres binaires, sachant que l'unité de base contient un arbre binaire $T(h_0)$ de hauteur h_0. Si on remplace h_0 par $k_0 = \frac{h_0}{2}$ (i.e., la hauteur de l'arbre compressé $T_c(h_0)$), et on considère que le plongement concerne les arbres contractés, alors $X_4(k_0, \ell, L)$ désignera un plongement d'arbres quaternaires. La différenciation par la notation X_4 nous permettra d'éviter toute ambiguïté. De la même manière, il suffit de remplacer h_0 par $2k$ dans les expressions analytiques données dans les section précédentes (dilatation, taille de grille nécessaire pour effectuer le plongement, ...).

Ainsi, pour plonger des arbres quaternaires, l'exemple de plongement de la figure 3.5 devient $X_4(2,3,3)$. Rappelons qu'un arbre quaternaire complet de hauteur h, noté $T_4(h)$, possède $\frac{4^h-1}{3}$ sommets. Nous référençons cette taille dans la suite par $|T_4(h)|$.

M.Jiber et A.Bellaachia ont définit deux algorithmes de plongements d'arbres quaternaires. Un plongement en $X_4(2,2,3)$ et un plongement en $X_4(3,5,5)$.

les mesures analytiques de ces plongements, listées dans la table ci-dessous [101, 102], montrent que le plongement en $X_4(3,5,5)$ améliore la dilatation et le temps de propagation de 17%. L'expansion est augmentée de 4% si on la compare avec celle du plongement en $X_4(2,2,3)$. Notons cependant que le plongement en $X_4(3,5,5)$ améliore le plongement en $X_4(2,3,3)$ en termes de longueurs de liens et de nombre de processeurs utilisés. Notons aussi que la congestion de $X_4(2,2,3)$ est égale à une constante 2 quel que soit h. Ce dernier s'avère donc utile si le ptotocole de communication utilisé est le routage wormhole.

Afin de compléter les mesures analytiques, un exemple de problème de type diviser-pour-reigner, l'algorithme de tri par fusion (merge-sort) a été mis en œuvre sur la grille X-Net de la MasPar [73, 102]. La figure 3.9 montre les temps d'exécution de l'algorithme tri par fusion (merge-sort) obtenus en implantant ce dernier avec chacun des trois plongements $X_4(2,3,3)$, $X_4(2,2,3)$ et $X_4(3,5,5)$.

plongement	Dilatation	Expansion	congestion	Temps de propagation
$X_4(2,3,3)$	$3 \times 2^{h-4}$	$9 \times 4^{h-2}/\|T_4(h)\|$	$h-2$	$3 \times 2^{h-3}$
$X_4(2,2,3)$	$3 \times 2^{h-4}$	$6 \times 4^{h-2}/\|T_4(h)\|$	2	$3 \times 2^{h-3}+1$
$X_4(3,5,5)$	$2.5 \times 2^{h-4}$	$6.25 \times 4^{h-2}/\|T_4(h)\|$	$h-2$	$2.5 \times 2^{h-3}$

Nous avons également implanté l'algorithme de tri par fusion en utilisant un plongement en H [74]. La figure 3.10 représente le facteur d'accélération obtenu en utilisant le plongement en $X_4(3,5,5)$ par rapport à un plongement en H.

La figure 3.11 représente la comparaison entre le temps d'exécution de l'algorithme tri par fusion en séquentiel et celui de l'exécution en parallèle en utilisant le plongement en $X_4(3,5,5)$ [74].

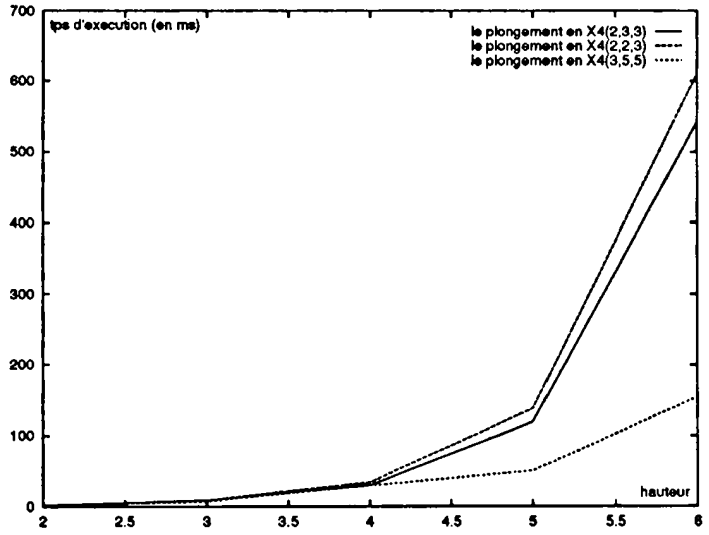

FIG. 3.9 - *les temps d'exécution de l'algorithme tri par fusion (merge-sort) obtenus en implantant ce dernier avec chacun des trois plongements.*

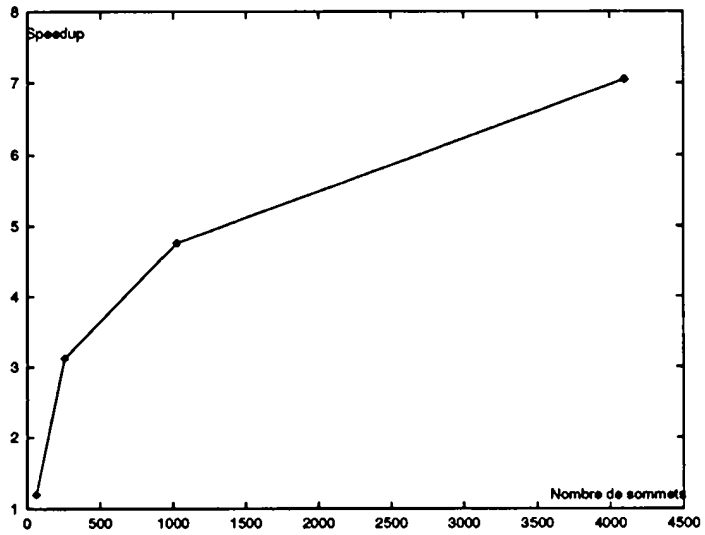

FIG. 3.10 - *Tracé du facteur d'accélération obtenu en utilisant le plongement en $X_4(3,5,5)$ par rapport à un plongement en H.*

3.13 Conclusion

Nous avons présenté dans ce chapitre une méthode de plongement d'arbres binaires complets sur une grille bi-dimensionnelle et qui vérifie les critères (1) à (4). Nous l'avons comparée avec un des plongements les plus connus sur une grille, le H-arbre, selon les mesures usuels de plongements.

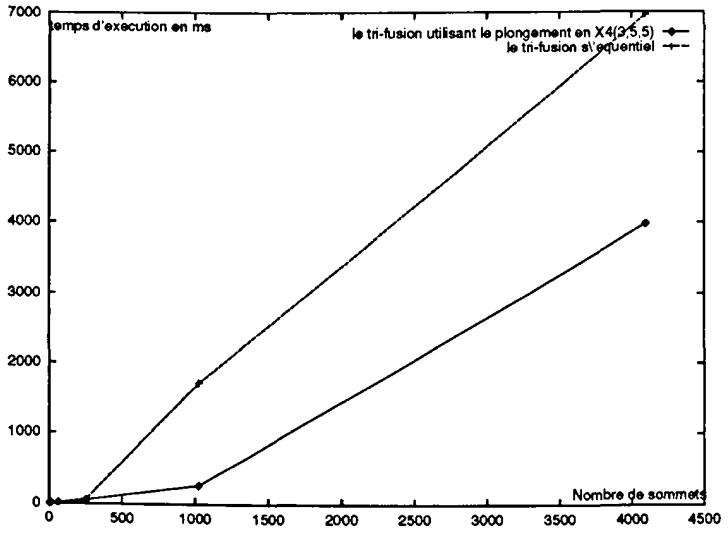

FIG. 3.11 - *Tracés du temps d'exécution de l'algorithme tri par fusion en séquentiel et en parallèle en utilisant le plongement en $X_4(3,5,5)$ sur la MaPar.*

Notons par ailleurs que pour plonger un arbre ayant n sommets, une grille de $O(n)$ processeurs est utilisée dans la méthode du plongement en H. Dans un plongement en X, une grille de $O(n)$ est également utilisé. De même, le lien de communication le plus long a une longueur $O(n^{1/2})$ pour les deux plongements. Mais il est important de noter que les constantes se cachant derrière O dans un plongement en X sont inférieures à celles du plongement en H (ceci est dû à la contraction ou la compression de l'arbre du départ).

Pour construire les plongements en X, nous avons "étiré" les arêtes de l'arbre sur les chemins dans la grille. Dans la suite, nous allons procéder autrement ; au lieu d'étirer les liens de l'arbre sur les liens de la grille, on va utiliser un algorithme de routage basé sur la notion du mouvement de données décrite dans [35]. On va pouvoir ainsi plonger un arbre général (pas forcément binaire) et quelconque (pas nécessairement équilibré) dans une grille multi-dimensionnelle. Plus précisément, nous allons construire des plongements permettant de plonger de manière dynamique un arbre quelconque dans une grille multi-dimensionnelle.

Partie 3

Chapitre 4

Les mouvements de données dans une architecture distribuée et synchrone

La notion de mouvement de données dans une architecture distribuée et synchrone couvre un ensemble d'outils permettant de copier et de déplacer des données à travers les processeurs de l'architecture parallèle. Cela permet aux données dispersées dans la mémoire distribuée de la machine de se trouver à la bonne place au bon moment.

Un problème de mouvement de données est défini de la manière suivante. Soit un ensemble de M données. Au départ, chaque donnée est stockée sur un des N processeurs du réseau. Supposons que des processeurs désirent lire une donnée détenue par un processeur, ou bien des processeurs désirent écrire leurs données dans d'autres processeurs. Le problème général consiste à router en le moins d'étapes possibles les données jusqu'à leurs destinations en utilisant un contrôle local.

Dans ce chapitre, nous décrivons les opérations de mouvement de données présentées dans [35]. Nous commencerons par présenter formellement ces opérations ainsi que les procédures de base qui les composent dans la section 5.3. En effet, ces opérations s'effectuent au moyen de quelques procédures bien définies. Nous décrirons ensuite plus en détail ces procédures de base et nous analyserons leurs complexités dans la section 5.2. Les complexités des opérations de mouvement de données seront discutées dans la section 5.4.

Dans les chapitres suivants, nous ne nous contenterons pas de nous restreindre à utiliser les opérations de mouvement de données telles qu'elles sont décrites par Nassimi et Sahni dans [35]. Nous allons voir dans ce chapitre que ces opérations permettent en effet de résoudre différents problèmes liés au mouvement de données. Ainsi, de nombreux algorithmes développés pour des architectures distribuées et synchrones utilisent ces opérations dans leurs élaborations (voir pour exemples,[44, 43, 2, 42, 5, 106]). Mais ces opérations ont été souvent utilisées comme un outil monolithique. Il est vrai qu'elles constituent un cadre se prêtant bien à la résolution de problèmes généraux de routage. Cependant, une meilleure compréhension de la forme de distribution des données et la nature des problèmes de routage à prendre en compte peut permettre d'améliorer l'utilisation des opérations de [35].

Plus précisément, nous allons voir qu'en fait, l'idée générale des mouvements de données de Nassimi et Sahni [35] est basé sur le fait qu'un problème quelconque de routage se décompose en un problème de tri et en un problème de routage monotone. Un problème de routage monotone est un problème dans lequelle l'ordre relatif des destinations des données est préservé. Un routage monotone lui même se résout en deux étapes, une étape de concentration et une étape de distribution. Pour certaines applications, si certaines contraintes, qui peuvent être faciles à satisfaire, sont vérifiées par le schéma de distribution des données, on peut rendre l'utilisation des algorithmes de mouvement de données plus efficace et optimale à un facteur constant près.

4.1 Les opérations de mouvement de données dans une architecture SIMD

Deux importantes opérations de mouvement de données dans une machine massivement parallèle synchrone ont été présentées dans [35], l'opération de *lecture à accès aléatoire* (RAR) et l'opération d'*écriture à accès aléatoire* (RAW).

Soient N processeurs disponibles numérotés de $i = 0, .., N-1$. $PE(i)$ désigne le processeur d'indice i. On suppose que les données sont contenues dans les mémoires des processeurs dans des variables enregistrements $G(i)$, où $G(i)$ indique la variable enregistrement dans le processeur $PE(i)$. L'enregistrement $G(i)$ contient un champ d'indice de destination $d(i)$ et un ou plusieurs champs de données $df(i)$ (i.e., $G(i) = (d(i), df(i))$).

La formalisation de ces deux opérations est la suivante :

- *Lecture à accès aléatoire (RAR)*: Chaque $PE(i)$ contient un indice dans $d(i)$, $0 \leq i < N$, indiquant que $PE(i)$ veut recevoir dans son enregistrement $G(i)$ une donnée de l'enregistrement $G(j)$ du processeur $PE(j)$, où $d(i) = j$. Si $PE(i)$ ne veut pas lire pendant ce cycle, alors $d(i) = \infty$.

- *Ecriture à accès aléatoire (RAW)*: Chaque $PE(i)$ contient un indice dans $d(i)$ indiquant qu'il veut envoyer une donnée de son enregistrement $G(i)$ à l'enregistrement $G(j)$ du processeur $PE(j)$, où $d(i) = j$. Si $PE(i)$ ne veut rien écrire pendant ce cycle, alors $d(i) = \infty$.

L'implémentation des deux opérations RAR et RAW s'effectue à l'aide de quelques procédures de base bien définies, données dans [35]. Ces procédures sont décrites formellement ci-dessous et seront détaillées un peu plus loin :

1. **Rank** : Un enregistrement $G(i)$ est dit sélectionné s'il est actif au cycle courant (i.e., son adresse de destination $d(i) \neq \infty$). Le rang de l'enregistrement sélectionné $G(i)$ est le nombre d'enregistrements sélectionnés d'indices inférieurs. Par exemple, les rangs des enregistrements séléctionnés (désignés par des astérisques sur les valeurs) suivants $(-, 2, -, 2^*, 3^*, 4, 4^*, 7^*)$ sont $(-, -, -, 0, 1, -, 2, 3)$.

2. **Concentrate**: Soient $G(i_r)$, $0 \leq r \leq j < N$, des enregistrements initialement dans les processeurs $PE(i_r)$. Les rangs r des enregistrements étant calculés par la procédure Rank. Un appel de la procédure Concentrate positionne l' enregistrement $G(i_r)$ sur le processeur $PE(r)$, $0 \leq r \leq j$. Les enregistrements sont ainsi concentrés sur les premiers processeurs de la machine suivant leurs rangs. Par exemple, supposons que $G(0:7) = (A, -, -, B, -, C, -, D)$. Après un appel de Concentrate, on a $G(0:7) = (A, B, C, D, -, -, -, -)$.

3. **Distribute**: c'est l'opération inverse de Concentrate. Soit $G(i)$, $0 \leq i \leq j < N$, des enregistrements initialement dans $PE(i)$. Soient $d(i)$ les destinations telles que $d(i) < d(i+1)$, pour $0 \leq i < j$ (i.e., les destinations sont dites monotones). Un appel de Distribute déplace les enregistrements $G(i)$ concentrés dans les premiers processeurs de la machine vers leurs processeurs $PE(d(i))$. Supposons que $G(0:7) = (A, B, C, -, -, -, -, -)$ et que $d(0) = 1$, $d(1) = 5$ et $d(2) = 6$. Après Distribute, on obtient $G(0:7) = (-, A, -, -, -, B, C, -)$.

4. **Generalize**: Elle suppose aussi que des enregistrements $G(i)$ placés sur les premiers processeurs $PE(i)$, $0 \leq i \leq j < N$, de la machine ont des destinations monotones $d(i)$. Les valeurs de d sont telles que $0 \leq d(0) < d(1) < \ldots < d(j) \leq N-1$ et $d(i) = \infty$ pour $j < i < N$. Un appel de Generalize duplique $G(i)$ dans les processeurs $PE(d(i-1)+1)$ jusqu'au $PE(d(i))$. Soient $G(0:7) = (A, B, C, -, -, -, -, -)$ et $d(0) = 1$, $d(1) = 5$ et $d(2) = 6$. Aprés un appel de Generalize, on a $G(0:7) = (A, A, B, B, B, B, C, -)$

5. **Sort**: Cette procédure trie les enregistrements $G(i)$ selon une clé prédéfinie. Soit $C(i)$ cette clé. Sort trie les enregistrements tels que $C(i) < C(i+1)$, pour $0 \leq i < N-1$.

Les problèmes de déplacement de données se résolvent par une suite d'appels *convenables* de ces procédures pour garantir que les bonnes données arrivent au bon endroit. En effet, en combinant les procédures Sort, Rank, Concentrate et Distribute, nous pouvons résoudre tout problème de routage aléatoire. L'aspect aléatoire signifie ici que les données sont stockées dans les mémoires des processeurs de façon irrégulière. Plus précisément, les données ne sont pas stockées initialement selon un schéma particulier, et on peut ainsi envoyer des données à leurs destinations sans avoir une connaissance globale du schéma de distribution des données a priori.

La mise en œuvre des mouvements de données, pour une simulation d'une opération RAW par exemple, est effectuée de la manière suivante. Supposons pour simplifier que toutes les destinations sont uniques. Dans un premier temps, on effectue un tri des enregistrements devant être routés selon leurs destinations en utilisant la procédure Sort. Une fois les enregistrements triés par destination, il suffit de les router vers leurs destinations finales en utilisant la procédure Distribute. Si les destinations sont monotones (i.e., l'ordre relatif des destinations est préservé), l'algorithme de routage peut être mis en œuvre également de la manière suivante. On commence par calculer le rang de chaque enregistrement avec la procédure Rank. Puis on effectue une concentration des enregistrements en utilisant la procédure Concentrate. On disperse ensuite les enregistrements en appliquant la procédure Distribute pour router chaque enregistrement vers sa destination finale. La figure 4.1

illustre un exemple d'une simulation d'une écriture à accès aléatoire. L'algorithme résultant s'exécute sans collision et tous les enregistrements sont routés vers leurs destinations exactes. La lecture à accès aléatoire est un peu plus complexe. Elle peut être implémentée comme deux écritures à accès aléatoire. Des exemples plus complexes d'opérations RAR et RAW seront expliqués et donnés un peu plus loin.

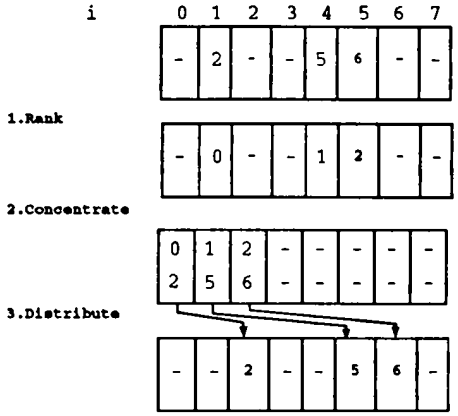

FIG. 4.1 - *Exemple d'une opération RAW. Le problème ici est celui de router les enregistrements de* $1 \rightarrow 2$, $4 \rightarrow 5$, $5 \rightarrow 6$. *Dans un premier temps, on calcule le rang de ces enregistrements en utilisant Rank. On les concentre avec Concentrate. On les distribue ensuite vers leurs destinations exactes avec Distribute. Si les destinations n'etaient pas monotones, on aurait d'abord triés les enregistrements par destination, puis il faut les router vers leurs destinations finales en utilisant Distribute.*

Les procédures de mouvement de données permettent en fait de résoudre de nombreux problèmes de routage. Le cas le plus simple des problèmes de routage est le routage injectif dans lequel chaque donnée n'a qu'une destination (un-vers-un). Les problèmes de concentration partielle et de diffusion partielle peuvent aussi être traités. Un problème de mouvement de données est une concentration partielle si plusieurs données ont la même destination (plusieurs-vers-un). Un problème de mouvement de données est une diffusion partielle si des données de certaines sources doivent être envoyées vers plusieurs destinations (un-vers-plusieurs).

Dans un problème de diffusion partielle, la principale difficulté est de créer toutes les données devant être envoyées. Si un processeur effectue lui même m copies d'une donnée, il lui faut au moins m étapes, ce qui peut être inefficace. Ce problème se résout ici à l'aide de la procédure Generalize qui distribue la tâche de création des copies afin d'être plus efficace.

Quand plusieurs données ont des destinations communes, c'est le cas de la concentration partielle, on se retrouve avec le problème des points de congestion. Par exemple, si à chaque étape, un enregistrement au plus, peut être reçu par un processeur destination, alors les enregistrements ayant une destination commune vont être retardés à cause du goulot

d'étranglement que présente la destination. Le problème est encore plus préoccupant si les points de congestion peuvent aussi retarder les enregistrements destinés à d'autres destinations. L'approche de [35] pour supprimer l'effet des points de congestion consiste à permettre aux données dont la destination est la même d'être fusionnées (ou combinées). La fusion de deux données ayant une même destination en une seule donnée, dont la taille peut être supérieure, a lieu quand les deux données se retrouvent dans un même processeur au même moment. Une autre solution envisagée dans [35] est celle de résoudre ces conflits d'écritures concurrentes en ne gardant, par exemple, que la valeur minimum des valeurs contenues dans les données ayant une même destination. Notons que la principale difficulté lors de la fusion est que les enregistrements devant subir une telle opération doivent se retrouver au même instant sur un même sommet. Nous allons voir plus loin la solution de [35] pour résoudre ce problème.

En fait, l'idée générale des mouvements de données de Nassimi et Sahni [35] est basé sur le fait qu'un problème quelconque de mouvement de données se décompose en un problème de tri et en un problème de routage monotone. En effet, un problème général de routage de $M = N$ enregistrements sur un réseau de N processeurs, où chaque enregistrement doit se rendre à une destination unique, se réduit en fait à un problème de tri de ces N enregistrements [81].

Dans le cas d'un problème de routage simple (un-vers-un) où toutes les destinations sont distinctes et $M < N$, il ne suffit pas de trier les M enregistrements en fonction de leurs destinations (rappelons que les $N - M$ enregistrements qui ne se déplacent pas ont la destination ∞). En effet, trier les M enregistrements revient à les concentrer sur les premiers processeurs de la machine dans l'ordre du tri, suivant des rangs consecutifs. Un enregistrement de destination i ne se retrouve pas nécessairement sur le processeur i si son rang est différent de i. Il est alors nécessaire ensuite de distribuer les enregistrements vers leurs destinations finales en utilisant la procédure Distribute.

Le cas d'un problème RAW où plusieurs enregistrements ont la même destination se résout de la manière suivante. Supposons qu'on peut fusionner tous les enregistrements ayant la même destination. Dans un premier temps, les enregistrements sont triés en fonction de leurs destinations. Tous les enregistrements ayant la même destination sont regroupés de façon contigue dans l'ordre du tri. Les rangs des enregistrements sont ensuite calculés. Les enregistrements ayant la même destination se feront attribuer le même rang. Tous ces enregistrements sont ensuite concentrés en utilisant la procédure Concentrate. Durant cette étape, les enregistrements de même rang rentrent en conflit puisqu'ils doivent se rendre dans un même processeur. Ces enregistrement seront alors combinés. Les enregistrements sont ensuite distribués au moyen de Distribute (voir la figure 4.2).

Si la solution de résoudre les conflits en gardant la donnée d'un seul enregistrement parmi ceux qui ont le même destination est envisagée, on commence également par trier les enregistrements en fonction de leurs destinations. Une fois les enregistrements triés, on calcule les rangs des enregistrements sélectionnés (i.e., les seuls enregistrements que l'on veut garder). La procédure Concentrate concentre ensuite ces enregistrements suivant leurs rangs. Il reste à router les enregistrement vers leurs destinations finales en utilisant la procédure Distribute (comme cela a été fait dans le premier exemple d'une opération RAW, voir la figure 4.1).

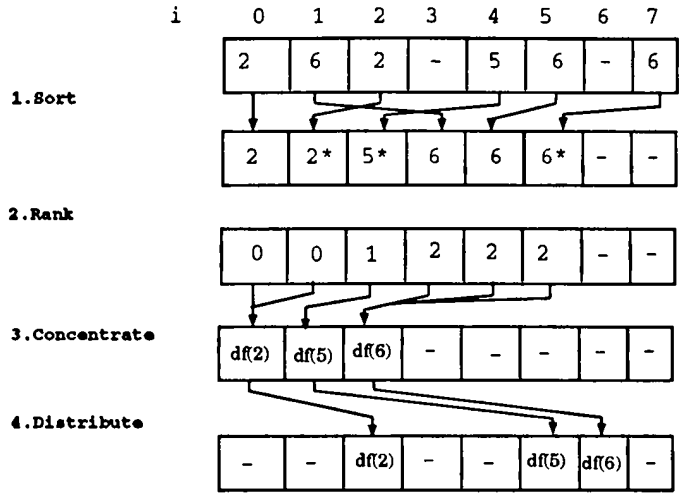

FIG. 4.2 - *Exemple d'une opération RAW. Dans cette opération plusieurs enregistrements ont la même destination. Sort trie les enregistrements en fonction de leurs destinations. Rank calcule les rangs. Les enregistrements de même destination se font attribuer le même rang. Concentrate concentre et fusionne les enregistrements de même rang. Distribute route dans une dernière étape les données vers leurs destinations finales.*

Dans les deux opérations RAW précédentes, on constate bien que ces opérations sont effectuées en deux vagues successives : on trie les enregistrements suivant leurs destinations puis on route les enregistrements vers leurs destinations finales en utilisant un routage monotone composé de Rank, Concentrate et Ditribute.

Le problème de RAR est un peu plus complexe qu'un problème RAW. L'opération RAR est implémentée également à l'aide du tri, et des autres procédures de bases. Notons que la difficulté de la création distribuée de copies dans le cas de la diffusion partielle est résolue à l'aide de Generalize.

Une opération RAR, où plusieurs processeurs veulent lire des données détenues par un seul processeur, est mise en œuvre de la manière suivante. Un appel de Sort dans un premier temps trie les enregistrements suivant leurs sources. A l'issue du tri, tous les enregistrements désirant lire dans un même processeur sont regroupés de façon contigue dans l'ordre du tri. Afin de pouvoir créer les copies, il faut d'abord disposer des données à copier. Un processeur parmi ceux désirant lire la même source est sélectionné. On se retrouve ainsi avec un problème de routage injectif et monotone (un-vers-un). A cette étape, on effectue des appels respectifs aux procédures Rank, Concentrate et Distribute comme cela a été fait lors de l'opération RAW de l'exemple précédent. Il reste maintenant à résoudre le problème de diffusion partielle des données lues (un-vers-plusieurs). Ces données sont concentrées en utilisant Concentrate pour permettre l'utilisation de la procédure Generalize. Un appel a Generalize permet de créer les copies nécessaires. La dernière étape de l'opération RAR consiste à envoyer les copies creées à leurs destinations finales. Cette étape est en fait un routage injectif (un-vers-un) dont les destinations sont les valeurs d'une variable T, ayant servi à stocker les indices de départ des processeurs voulant lire

(voir l'exemple 4.3). On utilise alors une fois de plus la procédure Sort en ayant comme clé les valeurs de cette variable T. Une opération RAR est illustrée par un exemple dans la figure 4.3. Dans l'exemple, on dispose de $N = 8$ processeurs. La configuration initiale du problème est la suivante. Les processeurs d'indices 0 et 2 veulent lire une donnée stockée dans l'enregistrement du processeur d'indice 2, le processeur 4 veut lire la donnée stockée dans le processeur 5, et les processeurs 1, 5 et 7 veulent lire celle du processeur 6.

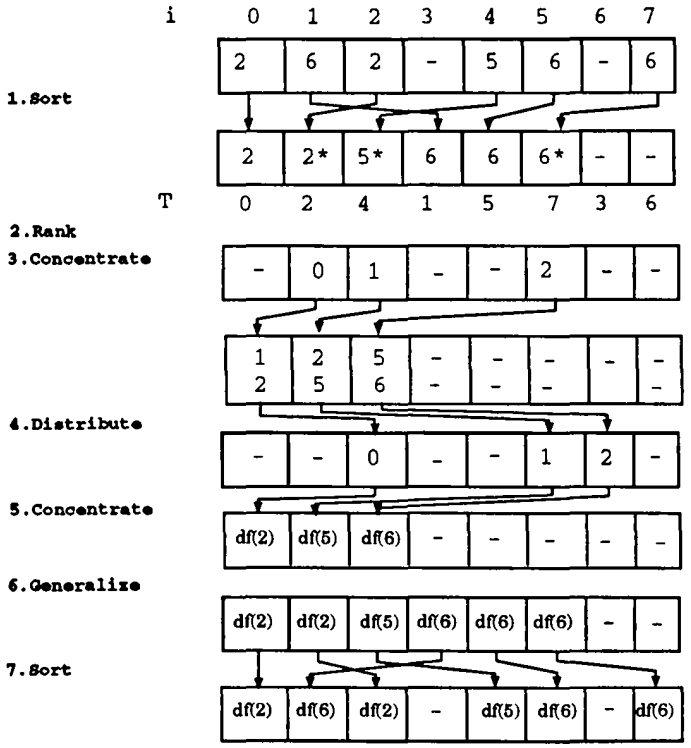

FIG. 4.3 - *Exemple d'une opération RAR. Dans cette opération, plusieurs processeurs veulent lire des données détenues par un seul processeur. On commence par trier les enregistrements par destination. Les indices des enregistrements avant le tri sont rangés dans une variable T. Un enregistrement parmi ceux qui ont la même destination est séléctionné (les enregistrements séléctionnés sont désignés par des astérisques). Une suite convenable d'appels des procédures Rank, Concentrate et Distribute ramène les données supposées contenues dans des registres $df(i)$. Ces données destinées à être copiées par la suite sont concentrées par Concentrate puis copiées en utilisant Generalize. Les copies creées ainsi sont envoyées ensuite vers leurs destinations finales en utilisant Sort suivant la clé donnée par T.*

Les deux opérations RAW et RAR permettent donc de résoudre les problèmes de déplacement et de copies des données dans une architecture distribuée. Ces deux opérations sont implémentées par une suite d'appels convenables des procédures bien définies. Nous allons dans la section suivante décrire les algorithmes correspondant aux procédures de mouvement de données et analyser leurs complexités.

4.2 Description des algorithmes des procédures du mouvement de données

Nassimi et Sahni ont considéré les architectures distribuées dotées des trois réseaux suivants. La grille de dimension quelconque, l'hypercube et le graphe mélange parfait.

Rappelons que la grille de dimension q et de côté n, $n_{q-1} \times n_{q-2} \times ... \times n_0$, est constituée de $N = n^q$ sommets. Son diamètre est $q(n-1)$. Un sommet $(i_{q-1}, i_{q-2}, ..., i_0)$ est connecté au sommet $(i_{q-1}, ..., i_j \pm 1, ..., i_0)$ dont il diffère exactement de 1 sur l'une des composantes (s'il existe). Les données envoyées d'un sommet à un autre doivent donc être acheminées à travers ces connexions. La grille de dimension q et de côté n est appelée communément grille q-dimensionnelle n-aire. Le cas particulier où $n = 2$ est l'hypercube de dimension $q = logN$ et ayant $N = 2^q$ processeurs. Son diamètre est $logN$ et chaque sommet possède $logN$ voisins.

Le graphe mélange-parfait de dimension q possède $N = 2^q$ sommets. On représente chaque sommet par un mot binaire de longueur q. Deux sommets u et v sont reliés par une arête s'ils diffèrent par leur dernier bit (l'arête est dite de type exchange), ou si u est obtenu par un décalage cyclique de v à gauche ou à droite (l'arête est dite de type décalage). Le graphe mélange parfait possède un diamètre $\Theta(logN)$. Notons que chaque sommet possède seulement 3 connexions [81]. Dans la suite, on va s'interesser tout particulièrement aux grilles de dimensions quelconques et aux hypercubes.

La complexité des algorithmes parallèles est généralement évaluée en termes de deux mesures: le temps de traitement T_{tment} et le temps de communication T_{comm}. La mesure T_{tment} reflète le temps maximum de traitement local exécuté par un processeur de façon séquentielle. Le temps de communication T_{comm} reflète la somme totale des temps de communications dépensés dans l'algorithme pour effectuer des transferts ou des échanges entre processeurs. Dans le cas des complexités des procédures de mouvement de données, on parlera de nombre d'unités de routages utilisées. Plus précisément, il s'agit du nombre d'étapes et du nombre de liens traversés nécessaires pour que chaque donnée atteigne sa destination.

Le coût total de la lecture à accès aléatoire RAR sur une grille $n \times n \times ... \times n$ de dimension q est $O(q^2n)$. Sur un hypercube ou un mélange parfait de N processeurs, le coût total est $O(log^2N)$. La puissance de deux dans ces complexités provient de la procédure Sort qui est la plus couteuse. La complexité de RAW est la même que celle de RAR quand chaque PE reçoit une seule donnée. Dans sa forme générale, si d données sont écrites dans un même PE, la complexié de RAW devient $O(q^2n + dqn)$ sur la grille de dimension q et $O(log^2N + dlogN)$ sur l'hypercube ou le mélange-parfait.

4.2.1 Notations

Pour décrire les algorithmes, on va utiliser les notations suivantes :

- pour tout entier i, on note par i_b le bit b de la représentation binaire de i (i_0 est le bit du poids le plus faible).

- $i_{s:r}$ représente le nombre dont la représentation binaire est $i_s i_{s-1} ... i_r$ avec $r \leq s$.
- $i^{(b)}$ est le nombre qui diffère de i au b-ième bit.
- "$\longleftarrow$" sera utilisé pour indiquer un transfert par un canal de communication et "$\longleftrightarrow$" pour indiquer un échange . Une affectation de ce type correspondra à une unité de routage.
- Un processeur est actif ou pas pour l'exécution d'une instruction selon les conditions données à la fin de cette instruction. Si aucune condition n'est indiquée, tous les PE executent l'instruction. Par exemple, la signification de l'instruction $A(k^{(b)}) \longleftarrow B(k), (k_b = 0)$ est la suivante: chaque PE dont le b-ième bit de son adresse k est 0 renvoie la donnée de sa cellule mémoire B à la cellule A du PE dont l'adresse est la même que celle de $PE(k)$ sauf que le b-ième bit est 1.

4.2.2 Descriptions des algorithmes

On va décrire ci-dessous les algorithmes relatifs aux procédures du mouvement de données et leurs complexités. Soit $N = 2^k$ le nombre de PE dans la grille ou l'hypercube. Les processeurs sont numérotés de 0 à $N - 1$. Chaque numéro est représenté par son écriture binaire, de longueur k. Les processeurs de la grille peuvent être numérotés suivant deux schémas: une *numérotation par ligne d'abord* ou une *numérotation par mélange de ligne d'abord.* L'exemple de la figure 4.4 illustre ces deux types de numérotations sur une grille 4×4. On désigne par *un bloc de taille b*, le bloc formé de 2^b processeurs dont les $k - b$ bits les plus significatifs sont identiques. La figure 4.4 illustre des blocs de taille 2^2 dans une grille dans le cas des deux numérotations (indiquées en trait gras). Tous les algorithmes décrits ci-dessous travaillent de la manière suivante : ils décomposent l'hypercube en blocs de sous-hypercubes ou la grille en blocs de sous-grilles puis opèrent en parallèle sur ces blocs.

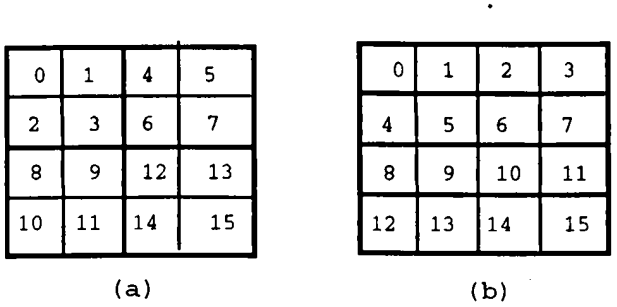

FIG. 4.4 - *(a) numérotation par mélange de ligne d'abord. (b) numérotation par ligne d'abord*

L'acheminement des enregistrements se fait de telle sorte qu'à chaque étape, au plus un enregistrement emprunte une arête et au plus un enregistrement peut être reçu par un processeur destination. On n'autorise pas en effet les enregistrements à être insérés dans des files d'attentes placées sur les processeurs. Ces algorithmes ont aussi comme propriété

que seules les arêtes d'une même dimension sont empruntées à chaque étape, et que des arêtes de dimensions consécutives sont empruntées au cours d'étapes consécutives.

Notons aussi que les algorithmes de routages de [35] s'effectuent en direct. Ceci implique que chaque processeur est capable de décider localement du devenir d'un enregistrement sans pré-traitement ni connaissance de la globalité du problème de routage. En effet, chaque processeur décide du devenir d'un enregistrement reçu en prenant en compte uniquement l'information representée par la destination attachée et progressant avec l'enregistrement et l'information locale dont il dispose représentée par son numéro.

4.2.2.1 La procédure Rank

La procédure Rank est une opération de calcul de préfixes en parallèle. Le calcul de ces préfixes (i.e., les rangs) ne concerne que les enregistrements séléctionnés (désirant lire ou écrire). L'idée de l'algorithme est que le rang d'un enregistrement dans un bloc de 2^b PE est le nombre d'enregistrements précédents celui-ci dans ce bloc. Un bloc 2^b se divise en deux sous-blocs 2^{b-1}. le $b-1$-ième bit du bloc de gauche est 0 et le $b-1$-ième bit du bloc de droite est 1. Soient $r(i)$ le rang de l'enregistrement (s'il existe) de $PE(i)$ dans le bloc 2^{b-1}, et $S(i)$ le nombre total des enregisrements selectionnés dans ce bloc. Le rang de l'enregistrement dans le bloc 2^b est $r(i)$ si le $b-1$-iéme bit est 0 (i.e., le bloc de gauche) et $r(i)+S(i)$ si le $b-1$-ième bit est 1.

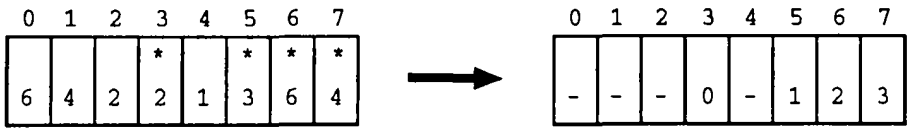

FIG. 4.5 - *Exemple d'emploi de la procédure Rank*

```
Procedure RANK
1  global r (le rang), S (est à 1 si l'enreg est sélectionné)
2  r(i) := 0
3  S(i) := 1 , (G(i) ≠ ∞)
4  S(i) := 0 , (G(i) = ∞)
5  for b := 0 to k − 1 do
6     T(i^(b)) ⟵ S(i)
7     r(i) := r(i) + T(i), (i_b = 1)
8     S(i) := S(i) + T(i)
9  end
end RANK
```

La procédure Rank décompose les $N = 2^k$ PE en blocs de taille b, pour $0 \leq b \leq k-1$. Elle utilise $k = logN$ unités de routage sur l'hypercube étant donné qu'une unité de routage est nécessaire par execution de la ligne 6 (la commmunication se fait via une ligne directe). Le temps de traitement sur chaque PE est en $O(k)$. Rank a donc une complexité optimale

et coûte $O(logN)$ sur l'hypercube.

Sur une grille $n \times n \times ... \times n$ de dimension q, dont le nombre de processeurs est $N = n^q = 2^k$, le temps de traitement sur chaque PE est également $O(k)$, mais le nombre total d'unités de routage est $2q(n-1)$ (les preuves sont données en détail dans [35]). Cette complexité est la même si les processeurs sont numérotés selon les schémas en ligne d'abord ou en mélange de ligne d'abord. Notons aussi que si deux PE ont des numéros successifs, ils ne sont pour autant voisins (par exemple, les processeurs 3 et 4 de la figure 4.4). Le voisinage est toujours celui donné par la topologie[1].

La complexité de la procédure Rank peut être résumée ainsi :

$$\begin{cases} T_{tment} = O(k); \\ T_{comm} = O(qn) \ pour \ la \ grille. \\ \quad\quad = O(logN) \ \ pour \ l'hypercube. \end{cases} \tag{1}$$

4.2.2.2 La procédure Concentrate

Procedure CONCENTRATE
1 global $G - d$ est un champ de l'enregistrement G
2 for $b := 0$ to $k - 1$ do
3 $G(i^{(b)}) \longleftrightarrow G(i)$, $(d(i) \neq null \ and \ d(i)_b \neq i_b)$
4 end
end CONCENTRATE

Le nombre d'unités de routage nécessaire pour la procédure Concentrate est analogue à celui de la procédure Rank. Le temps de calcul par PE est également le même. La procédure Concentrate déplace les enregistrements des $PE(i)$ au $PE(d(i))$ où les valeurs de $d(i)$ sont déterminées par la procédure Rank.

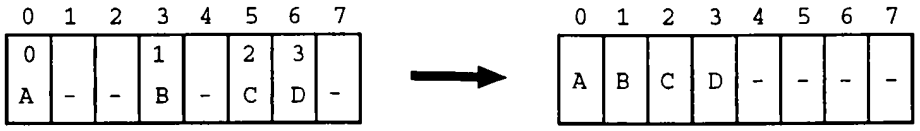

FIG. 4.6 - *Exemple d'emploi de la procédure Concentrate*

Dans la procédure Concentrate, chaque enregistrement emprunte un chemin pour atteindre sa destination calculée par Rank de la manière suivante. Au cours de chaque étape b, le bit

1. Considérons la numérotation en ligne d'abord. Le PE dont la position est le q-uplet $(i_{q-1}, i_{q-2}, ..., i_0)$ dans la grille possède le numéro $j = i_{q-1} i_{q-2} .. i_0$. Cet indice est aussi $j = j_{k-1} ... j_0$ où $j_{(d+1)logn-1:(d)logn} = i_d$, $0 \leq d < q$. La dimension correspondante au b-ième bit est en fait, la dimension $\lfloor b/log\ n \rfloor$ de la grille. La distance entre deux PE dont les num'eros diffèrent au b-ième bit (i.e., entre $PE(i)$ et $PE(i^{(b)})$ est 2^u, $u = b \ mod \ logn$. Ces deux PE diffère sur la dimension $\lfloor b/log\ n \rfloor$. Le nombre total d'unités de routage nécessaire est donc $2\lfloor k/log\ n \rfloor (n-1)$. Quand $k = log\ N = qlog\ n$, Le nombre total d'unités de routage nécessaire pour l'exécution de la ligne 6 est $2q(n-1)$.

b de l'indice du processeur en cours, $i_k...i_b...i_1i_0$, est comparé au b-ième bit de l'adresse de destination, $d_k...d_b...d_1d_0$. Si ces bits sont identiques, l'enregistrement reste sur place pour cette étape. Si ces bits sont différents, alors l'enregistrement est envoyé vers le processeur $i_k...d_b...i_1i_0$ de l'enregistrement. Au bout des k étapes, de 0 à $k-1$, tous les enregistrements rejoignent leurs destinations données par la procédure Rank. A titre d'exemple, supposons que l'enregistrement de $PE(110)$ a pour rang 3 et doit donc être envoyé vers $PE(011)$. Dans la première étape, il est envoyé vers $PE(111)$ qu'il ne quitte pas pendant la deuxième étape. Il est ensuite envoyé vers $PE(011)$ à la troisième étape qui représente sa destination finale.

4.2.2.3 La procédure Distribute

La procédure Distribute est l'opération inverse de Concentrate. La complexité de Distribute est donc la même que Concentrate.

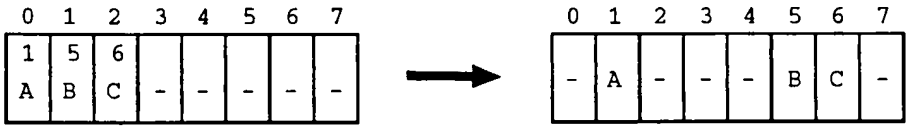

FIG. 4.7 - *Exemple d'emploi de la procédure Distribute*

```
Procedure DISTRIBUTE
1  global G
2  for b := k − 1 downto 0 do
3     G(i^(b)) ⟷ G(i), (H(i) ≠ null and H(i)_b ≠ i_b)
4  end
end DISTRIBUTE
```

La procédure distribute route les enregistrements de la même manière que Concentrate. La différence est que, dans la concentration, les bits des indices des processeurs et des adresses de destinations sont traités à partir de la droite (le poids faible d'abord). Alors que dans la distribution, les bits sont traités dans l'ordre inverse à partir de la gauche (le poids fort d'abord). Au bout des k étapes, de $k-1$ à 0, tous les enregistrements rejoignent leurs destinations exactes.

Notons que les deux algorithmes de routage Concentrate et Distribute s'exécutent en direct. En effet, ces algorithmes sont capables de traiter immédiatement les enregistrements dès qu'ils arrivent à un processeur en prenant en compte uniquement l'information locale dont il dispose (i.e., son indice) et l'information attachée et progressant avec l'enregistrement (i.e., la destination).

4.2.2.4 La procédure Generalize

La procédure Generalize déplace les enregistrements comme Distribute et les duplique si nécessaire. La complexité de Generalize est également la même que celle de la procédure Rank.

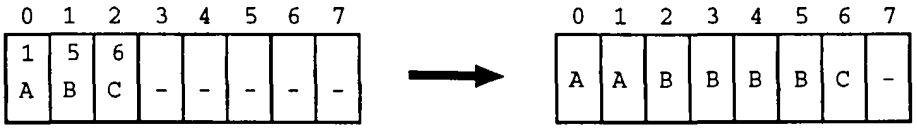

FIG. 4.8 - *Exemple d'emploi de la procédure Generalize*

Procedure Generalize
1 global G
2 for $b := k - 1$ downto 0 do
3 $G'(i^{(b)}) \longleftrightarrow G(i)$, $(H(i) \neq null \text{ and } H(i)_b \neq i_b)$
4 $H(i) := null$, $(H(i) < i - i_{b-1:0})$
5 $H'(i) := null$, $(H'(i) < i - i_{b-1:0})$
6 $G(i) := G'(i)$, $(H'(i) < H(i))$
7 end
end GENERALIZE

Dans un problème de diffusion partielle, si un processeur effectuait lui même M copies d'une donnée afin de l'envoyer vers M destinations différentes, il lui faudrait au moins M étapes, ce qui peut être inefficace. Une solution plus efficace résout ce problème de la manière suivante. On calcule le rang de chaque enregistrement. Puis on concentre ces enregistrements vers les processeurs correspondants aux rangs calculés. On effectue ensuite les copies nécessaires en utilisant la procédure Generalize. Il reste après à envoyer les copies créées aux destinations finales.

Generalize décompose, comme les autres procédures, la grille et l'hypercube en sous-blocs. Le bloc de 2^b processeurs se décompose en deux blocs 2^{b-1}. Le bloc 2^b contient des PE dont le bit $b-1$ est 0 et des PE dont le b bit est 1. Generalize s'exécute dans le bloc 2^b en envoyant les enregistrements des deux blocs $2^b - 1$, l'un vers l'autre, vers les processeurs correspondants (la ligne 3). Si une adresse de destination $d(i)$ est telle que $d(i) < i - i_{b-2:0}$, alors l'enregistrement $G(i)$ n'est pas dupliqué dans $PE(i)$. En effet, l'indice $d(i)$ est inférieur aux indices du bloc 2^b. La preuve de correction complète de la procédure est donnée dans [35].

4.2.2.5 La procédure Sort

En ce qui concerne le tri, les enregistrements peuvent être triés en $O(q^2 n)$ dans une grille de dimension q et de côté n en utilisant l'algorithme décrit dans [34] (avec les deux schémas

de numérotation possible). Dans l'hypercube, les enregistrements peuvent être triés en $O(log^2N)$ en utilisant le tri bitonique de [78].

4.3 Complexité des opérations de mouvement de données

Nous avons vu dans la section 5.3 que l'idée générale des mouvements de données de Nassimi et Sahni [35] est basé sur le fait qu'un problème quelconque de mouvement de données se décompose en un problème de tri et en un problème de routage monotone. Le problème de routage monotone se résout également en deux vagues. Une première étape qui consiste à concentrer les enregistrements dans les premiers processeurs de la machine suivant des rangs successifs. Cette étape est effectuée en utilisant Rank+Concentrate. La deuxième étape consiste à router les enregistrements vers leurs destinations exactes en utilisant Distribute ou Generalize si les données doivent être dupliquées.

Soit T le temps nécessaire pour trier N éléments sur un réseau de N processeurs. Les procédures de mouvements de données permettent de résoudre tout problème générale de routage de M enregistrements sur un réseau de N processeurs en $T + O(logN)$ étapes sur un hypercube de N sommets et $T + O(qn)$ sur une grille de $N = n^q$ sommets. En d'autres termes, le problème de routage se résout en $O(T)$ sur les deux structures puisque $T = log^2N = \Omega(logN)$ sur l'hypercube et $T = O(q^2n) = \Omega(qn)$ sur la grille.

Si on suppose que les destinations sont monotones (i.e., si $i < j$ alors $dest(i) < dest(j)$), alors il n'est pas nécessaire d'utiliser le tri. Il suffit en effet de résoudre un problème de routage monotone en utilisant les procédure Rank, Concentrate, Distribute et Generalize. Il est suffisant en effet de concentrer les enregistrements après avoir calculé leurs rangs, en utilisant Rank et Concentrate, avant de les distribuer vers leur destinations exactes en utilisant Distribute ou Generalize.

Ceci implique que sans la procédure Sort, si on admet que les enregistrements sont initialement triés (ou monotones), les algorithmes de RAR et RAW requièrent seulement $O(qn)$ sur la grille et $O(log\ N)$ sur l'hypercube. En effet, toutes les procédures, à l'exception du tri, ont une complexité $O(qn)$ et $O(logN))$ respectivement sur la grille et l'hypercube.

4.4 Extensions

Les algorithmes de mouvement de données tels qu'ils sont décrits dans [35] considèrent que le nombre d'enregistrements est inférieur ou égal au nombre de processeurs. L'acheminement des enregistrements se fait de telle sorte qu'à chaque étape, au plus un enregistrement emprunte une arête, et au plus un enregistrement peut être reçu par un processeur destination (i.e., au plus un enregistrement soit stocké dans un sommet).

Dans le cas où il se peut qu'il y ait plus d'enregistrements que de processeurs (i.e., on a M enregistrements tels que $M > N$), ce qui implique que plusieurs enregistrements peuvent être stockés sur un seul processeur, les algorithmes de mouvement de données de [35] sont utilisables en étant ralenti d'un facteur de $\lceil \frac{M}{N} \rceil$ (en supposant qu'il y a $\frac{M}{N}$ enregistrements par processeur). En effet, il suffit de décomposer le problème de mouvement de données

en $\frac{M}{N}$ problème de mouvement de donnée injectifs.

Nous devons donc étendre les procédures de mouvement de données pour qu'ils s'exécutent dans le cas où le nombre de processeurs est inférieur à la taille du problème à traiter (i.e., chaque processeur peut stocker plusieurs sommets). En d'autres termes, nous devons convertir les algorithmes conçu pour une grille de M processeurs pour s'exécuter dans une grille de N processeurs, où $N < M$. Le facteur de ralentissement de $\lceil \frac{M}{N} \rceil$ induit est normal puisqu'il y a $\frac{M}{N}$ fois moins de processeurs. Ce résultat est donc optimal.

Plus precisément, les algorithmes des mouvements de données qui s'exécutent en T étapes dans une grille de M sommets peuvent être simulés dans une grille N processeurs, où $N < M$, en $T\lceil \frac{M}{N} \rceil$ étapes.

La simulation est triviale. Il suffit de contracter (ou plonger) la grille de M sommets dans la grille de N sommets. Soient G une grille de dimension q dont le nombre de PE est $M = m^q = 2^k$, et g une grille de dimension q et dont le nombre de PE est $N = n^q = 2^p$, avec $N < M$. On définit la contraction (ou le plongement) π de la manière suivante. Chaque processeur de G est représenté par son mot binaire $(a_{k-1}...a_0)$. On pose $\pi(a_{k-1}...a_{p-1}a_{p-2}...a_0) = (a_{p-1}a_{p-2}...a_0)$. En d'autres termes, le processeur d'indice i de G est simulé par le processeur d'indice $i \ mod \ N$ de g. Chaque processeur de la grille g simule donc 2^{k-p} processeurs de la grille de départ G. De plus, on doit distinguer deux sortes de communications : des communications inter-processeurs et des communications intra-processeurs. Rappelons que deux processeurs communiquent dans une procédure du mouvement de données (ils sont voisins pour la procédure) s'ils diffèrent d'un bit.

- communications inter-processeurs : deux processeurs qui différent d'un bit parmi les p bits du poids faibles dans G sont simulés par deux processeurs dans g. Pour simuler une communication entre ces deux processeurs dans G, $\frac{M}{N}$ messages sont transmis sur un même lien de g.

- communications intra-processeurs : deux processeurs qui différent d'un bit parmi les $k - p$ bits du poids fort dans G sont simulés par le même processeur dans g. La simulation s'effectue donc en $\frac{M}{N}$ étapes (d'une manière séquentielle).

Quelques modifications aux algorithmes des procédures de mouvement de données décrits dans la section précédente doivent donc être apportées. Prenons pour exemple la procédure Concentrate.

```
Procedure CONCENTRATE(k,p)
- - communications inter-PE
1  for b := 0 to p − 1 do
2     for j := 0 to 2^(k−p) − 1 do
3        G_j(i^(b)) ⟷ G_j(i),   (H_j(i) ≠ null and H_j(i)_b ≠ i_b)
4     end
5  end
- - communications intra-PE
6  - - Traiter les 2^(k−p) sommets d'un même processeur séquentiellement
end CONCENTRATE
```

La complexité de la procédure est donc la suivante:

$$\begin{cases} T_{tment} = O(p\frac{M}{N}); \\ T_{comm} = O(pn\frac{M}{N}). \end{cases} \quad (2)$$

Les changements à apporter aux procédures Distribute, Generalize et Rank sont similaires à ceux de la procédure Concentrate, en ajoutant les lignes 2 et 6 et en modifiant la ligne 3. Les complexités sont également similaires à (2).

4.5 $O(\frac{M}{N}T)$ et $O(\frac{M}{N}+T)$

Nous avons vu qu'un algorithme de mouvement de données peut être implementé à l'aide des opéations de Nassimi et Sahni pour résoudre un problème quelconque de routage aléatoire d'au plus N enregistrements sur un réseau de N processeurs disposé en grille ou en hypercube.

Nous avons vu également que l'on peut étendre cet algorithme pour résoudre le problème de routage de M enregistrements sur un réseau de N processeurs, où $N < M$, avec un facteur de ralentissement égale à $\lceil M/N \rceil$.

Plus precisément, nous avons decomposé le problème de routage de M enregistrements sur un réseau de N processeurs en $O(M/N)$ problèmes de routage injectifs (en supposant que tout processeur commence et termine avec $O(M/N)$ enregistrements). Le problème se résout ainsi en $O(TM/N)$ étapes, atteignant ainsi un facteur d'acceleration linéaire.

Bien que cette solution produise un temps d'exécution optimal puisque toute solution pour résoudre le problème prendra un temps $\Omega(TM/N)$, on peut se demander s'il est possible d'améliorer ses performances et de trouver une autre soultion qui résout le problème en $O(M/N+T)$ au lieu de $O(TM/N)$ étapes?. La réponse réside dans l'utilisation du pipeline.

Dans les algorithmes tels que decrits jusqu'ici, chaque processeur peut communiquer en n'utilisant qu'un lien de communication à une étape donnée. Ces algorithmes s'appliquent parfaitement aux réseaux dans lesquels chaque processeur ne peut envoyer et/ou recevoir qu'une donnée à chaque étape. Ces processeurs sont dotés d'un seul port de communication. On parle alors du modèle de réseau à 1-port. Il est cependant plus efficace de permettre l'envoi simultané d'enregistrements sur plusieurs liens à partir d'un même processeur. Dans le modèle multi-port, chaque processeur peut envoyer et/ou recevoir une donnée de chacun de ses voisins en une seule étape. Un ($log\ p$)-model a été proposé dans [134] (via [95, 97] et [65]), appelé modèle d'hypercube pipeliné dans lequel chaque processeur peut envoyer et/ou recevoir une donnée de chacun de ses $logp$ voisins à une étape donnée.

Les opérations de base de Nassimi et Sahni peuvent être adaptées à ce modèle comme cela est mentionné dans [95, 97, 65]. Ainsi k opérations de mouvement de données peuvent être exécutées en $k + log\ p$ étapes sur le modèle hypercube pipeliné. Pourrions-nous alors changer les algorithmes en les pipelinant tout simplement afin d'aboutir au même temps d'exécution sur le modèle 1-port?. La réponse est négative sauf pour la procédure Rank. L'opération de calcul de préfixe Rank donnée par Nassimi et Sahni utilise tous les processeurs à chaque étape. Mais cette opération peut ête pipelinées sur le modèle 1-port. En

effet, il suffit de considerer l'algorithme de calcul des prefixes sur un arbre binaire. Dans cet algorithme, un seul niveau de l'arbre est actif à chaque étape. Cet algorithme est donc pipelinable et il suffit ensuite de plonger l'arbre binaire dans le modèle 1-port. Ainsi, k opérations de calcul de prefixe peuvent être exécutées en $O(k + log\ p)$ sur un hypercube de p processeurs.

Pour les deux opérations Concentrate et Distribute malheureusement, s'il est possible d'exécuter k de ces opérations en $k + log\ p$ étapes sur le modèle hypercube pipeliné, on ne peut atteindre cette borne sur le modèle 1-port. Une preuve élégante est donnée dans [48, 95, 97] où on montre que ces opérations requiert une borne inférieure de $\Omega(k\ log^{1/2}p)$ étapes. En effet, si on considère un ensemble de k routages monotones pour lesquels les processeurs sources sont exactement ceux ayant plus de 0 que de 1 dans leurs numéros, et les processeurs destinations sont ceux ayant plus de 1 que de 0. Dans ce cas, $\Omega(kp)$ enregistrements doivent traverser les $O(plog^{-1/2}p)$ processeurs ayant autant de 1 et de 0 (ou un 0 de plus que 1 si $log\ p$ est impair) dans leurs indices. Ceci implique une borne inférieure de $\Omega(k\ log^{1/2}p)$ étapes pour exécuter k routage monotones [95, 97].

4.6 Conclusion

Il est vrai que les opérations de mouvement de données de [35] constituent un cadre se prêtant bien à la résolution de problèmes généraux de routage. En effet, ces opérations permettent aux données injectées sur les processeurs de la machine parallèle, et ne possédant pas a priori une forme prédéfinie, d'être à la bonne place au bon moment. On peut se représenter ce type de problème et les relations entre les données pour lequel on n'a pas défini de forme de distribution comme un plat de "spaghettis". Ces opérations permettent en conséquence de simuler par exemple la machine abstraite parallèle et universelle, la machine PRAM [35, 75, 81].

Cependant, une meilleure compréhension de la forme de distribution des données et la nature des problèmes de routage nécessaires peut permettre d'améliorer l'utilisation des opérations de [35]. Comme le modèle est fondé sur une liste d'appels convenables des procédures, on peut alors imaginer qu'on peut réduire la liste de "choses-à-faire". Par exemple, serait-il toujours nécessaire d'appeler la procédure Rank et la procédure Concentrate avant d'utiliser Distribute?. On sait par ailleurs que l'appel ordonné de ces trois procédures fait que ces étapes devrait assurer l'acheminement des données aux destinations appropriées en évitant les conflits. C'est sur la présence de mauvais ou bons conflits autrement dit les preuves de corrections des procédures qu'ont va se baser pour aborder et discuter les problèmes d'optimisation.

Chapitre 5

Plongement d'arbres quelconques et dynamiques

5.1 Introduction

Dans le chapitre 3, nous avons étudié des méthodes de plongements d'arbres binaires complets, ou d'arbres quaternaires complets, dans un réseau de type grille bidimensionnelle. Nous avons comparé ces méthodes de plongements avec une méthode standard de plongement d'un arbre binaire complet dans une grille bidimensionnelle, le plongement en H. Plus précisément, nous avons montré comment plonger un arbre binaire complet de $N-1$ sommets dans une grille de $\Omega(N)$ sommets en améliorant le plongement en H en expansion et en dilatation. Ces méthodes de plongements ne sont toutefois valables que pour des arbres binaires, complets et statiques, c'est à dire dont on connaît a priori la taille et la forme.

Dans la suite, nous allons étudier le problème de plongement d'arbres quelconques. Les réseaux de l'architecture parallèle à mémoire distribuée considérés sont de type grille multi-dimensionnelle (i.e. la grille de dimension quelconque et l'hypercube).

La structure d'arbre intervient naturellement dans de nombreux algorithmes pratiques. Les algorithmes d'optimisation combinatoire, de type diviser-pour-régner, d'évaluation d'arbres de jeux et d'évaluation d'expressions fonctionnelles possèdent très souvent une structure intrinsèque en arbre dans laquelle les sommets représentent les tâches calculatoires et les arêtes les communications nécessaires aux calculs. Mais, il est souvent difficile de connaître a priori la forme de l'arbre ou sa taille avant que les calculs ne soient quasiment tous effectués [17, 81]. Mettre en œuvre de façon efficace ces algorithmes sur une grille de degré quelconques ou un hypercube requiert donc un plongement efficace de l'arbre de telle sorte que les traitements soient équitablement répartis sur les processeurs (i.e., de telle sorte que la charge soit faible), et de telle sorte que les communications soient locales (i.e., de telle sorte que la dilatation soit faible). Mais, il est préférable que, de plus, le plongement puisse s'effectuer de façon dynamique. En d'autres termes, il est préférable que chaque sommet de l'arbre soit plongé sans tenir compte de la forme du reste de l'arbre ni de sa taille.

Le problème du plongement dynamique d'arbres binaires quelconques dans un hypercube a été initialement proposé et étudié par Bhatt et Cai dans [17]. Le problème a été posé de la manière suivante [17, 81, 82]: on suppose pour simplifier que l'arbre binaire croît d'un sommet à chaque étape, en commençant par la racine. A chaque fois qu'un sommet est ajouté à l'arbre, il l'est en tant que fils d'un sommet déjà existant et il est immédiatement placé dans l'hypercube. La décision de savoir où placer le sommet se fait localement sans tenir compte des sommets qui pourront être ajoutés dans le futur. La décision du placement doit être implémentée dans le réseau d'une manière totalement distribuée sans recours à aucune information globale. De plus, une fois qu'un sommet est ajouté, il ne peut plus être déplacé par la suite. Ce paradigme proposé par Bhatt et Cai [17] n'autorise donc pas la migration de processus (i.e., les sommets de l'arbre). Bien sûr, permettre la migration peut potentiellement donner une charge et une dilatation meilleures, mais sa mise en œuvre peut aussi être extrêmement coûteuse dans la pratique selon [17, 82, 89].

Notons que le problème du plongement dynamique est plus difficile que le plongement statique étant donné que les nouveaux sommets (les fils) générés doivent être alloués aux proceseurs d'une manière incrémentielle, sans prédiction quant à la forme future de l'arbre. Bien sûr, si on peut avoir la connaissance globale de l'arbre avant d'effectuer le plongement, on peut utiliser des algorithmes de plongements déterministes dont la congestion, la dilatation et la charge sont constantes [81]. Dans [16], on a prouvé que tout arbre binaire statique se plonge dans un hypercube avec une dilatation et une congestion constante.

Bhatt et Cai décrivent dans [17] un algorithme plongeant dynamiquement un arbre binaire de M sommets dans un hypercube de N sommets ayant, avec une forte probabilité, une dilatation égale à $O(loglogN)$ et une charge en $O(\frac{M}{N}+1)$. La congestion du plongement n'est pas determinée mais elle est probablement de l'ordre de $log^{\alpha}N$ pour une constante α selon [82].

Ce résultat fut amélioré par Leighton, Newman, Ranade et Schwabe [81, 82] dont un premier algorithme plus simple plonge dynamiquement un arbre binaire quelconque ayant M sommets dans un hypercube de N sommets avec une dilatation 1, et avec une forte probabilité, une charge en $O(\frac{M}{N}+logN)$. Ils décrivent ensuite un autre plongement qui permet d'obtenir une dilatation constante en $O(1)$, une charge en $O(\frac{M}{N}+1)$ et une congestion en $O(\frac{M}{N}+1)$. Ce dernier algorithme, qui améliore d'un facteur $logN$, pour $M=\Theta(N)$, la borne sur la charge du premier algorithme est malheureusement assez complexe [81].

Il est important de noter que les algorithmes de plongement dynamiques décrits dans [17, 81, 82] sont des algorithmes probabilistes qui permettent de plonger tout arbre binaire avec une forte probabilité. L'aspect aléatoire de ces algorithmes est primordial car on donne dans [82] une borne inférieure pour tout algorithme déterministe de plongement dynamique dans un hypercube qui prouve que tout algorithme de plongement déterministe qui équilibre la charge a nécessairement une dilatation $\Omega(\sqrt{log\,N})$ [82]. On montre ainsi, par conséquent que tout algorithme de plongement dynamique qui optimise simultanement la charge et la dilatation doit être aléatoire [82].

Nous reviendrons sur les algorithmes aléatoires de [17, 81, 82, 89] dans le chapitre suivant dans lequel nous allons présenter une approche aléatoire de plongement d'arbres quelconques dans une topologie arbitraire.

Dans un travail récent, et à l'encontre du paradigme proposé par Bhatt et Cai [17], Heun et Mayr [53] ont proposé un algorithme déterministe de plongement dynamique d'arbres binaires dans un hypercube basé sur la migration (i.e., les sommets de l'arbre plongé sont déplaçables). Leur algorithme plonge un arbre binaire de M sommets dans un hypercube avec une dilatation constante, et une charge en $O(\frac{M}{N})$. Cependant, cet algorithme dépense un temps dû à la migration, appelé par les auteurs le temps d'amortissement, en $O(log^2 m)$ à chaque étape dans laquelle au plus m nouveaux sommets sont à ajouter dans l'arbre.

Rappelons que le plongement d'un *graphe logique* G (le graphe que l'on plonge) dans un *graphe hote* H (graphe dans lequel on effectue le plongement) consiste à placer les sommets de G sur des sommets de H et à affecter à chaque arête de G un chemin dans H. En d'autres termes, pour plonger le graphe G dans H, on s'autorise à *"étirer"* les arêtes de G et à les projeter sur les chemins de H. Les plongements ainsi basés sur *"l'étirement"* sont habituellement étudiés en fonction de paramètres comme la dilatation (qui correspond à l'étirement maximum nécessaire considéré sur toutes les arêtes pour effectuer le plongement), l'expansion (le rapport entre le nombre de sommets de H et le nombre de sommets de G), la congestion (le nombre maximum d'arêtes de G qui sont plongées sur une même arête de H) et la charge (le nombre maximum de sommets de G qui sont plongés dans un seul sommet de H). Ces paramètres mesurent l'efficacité avec laquelle le graphe hôte peut simuler le graphe logique. En effet, quand H simule des communications prévues dans G, la perte de temps provient du fait que la structure des liens est différente (ce que mesure en partie la dilatation), et du fait qu'il y a engorgement de ces liens (ce que mesure la congestion) [119]. Comme nous l'avons déjà signalé, le meilleur plongement est évidemment celui pour lequel la dilatation, l'expansion, la congestion et la charge sont faibles. Dans le cas général, on ne trouve pas de plongements qui satisfassent tous ces paramètres simultanément et on doit alors toujours trouver des compromis entre ces paramètres. Notons que les plongements qui nous intéressent ici sont des plongements non injectifs (i.e., le nombre de nœuds de l'arbre à plonger est supérieur au nombre de processeurs disponibles). Dans ce cas, la charge devient un paramètre très essentiel.

Supposons que l'on ait un plongement de G dans H avec une dilataion d et une congestion c. Considérons que lors d'une étape de communication dans H, une arête du réseau ne peut être empruntée que par un seul message. Le terme une étape de communication défini un ensemble de communications effectuées en parallèle au même moment. Pour simuler une étape de communication entre deux voisins dans G, au plus d étapes dans H sont nécessaires puisque chaque arête dans G s'étire sur au plus d arêtes dans H. De plus, puisque c messages doivent emprunter une même arête dans H, alors la simulation prend au moins c étapes. De même, chaque message peut être retardé au plus c étapes dans un processeur, et donc cd étapes sont suffisantes pour simuler une étape de G. On déduit donc que si T est le nombre d'étapes de communications nécessaires pour simuler sur H une étape de communication dans G, alors on a [16, 119]

$$max(c, d) \leq T \leq cd. \tag{5.1}$$

Leighton, Maggs et Rao donnent dans [86] une meilleure majoration du nombre T d'étapes de communication sur H en montrant qu'une étape de communication sur G peut être simulée par $O(c + d)$ étapes de communication sur H. Cependant, la preuve donnée dans [86] n'est qu'une preuve d'existence qui, de plus, suppose un pré-traitement (simulation

off-line ou pré-caclulée). En fait, le problème qui a été consideré au départ dans [86] et les travaux qui lui ont succédé [84, 83, 85] est le suivant: étant donné un réseau et un ensemble de chemins pré-déterminés, le but est de produire un ordonnancement qui minimise le temps total et la taille maximum des files d'attentes au niveau des processeurs nécessaires, pour router des paquets empruntant ces chemins, et sachant qu'un paquet au plus traverse chaque arête. On montre dans [86, 84, 83] que le routage peut être effectué en $O(c+d)$, d étant la distance du chemin le plus long et c la congestion des chemins. Ce résulat trouve des applications dans le problème de routage dans les machines parallèles, l'émulation de réseaux (i.e., plongement de graphes) et le job scheduling [84].

L'application au problème du plongement se traduit par le fait qu'à une communication sur une arête de G correspond par le plongement un chemin dans H, la dilation d correpond au chemin le plus long et c la congestion des chemins.

Un point essentiel de ce qui vient d'être présenté est que le graphe G doit être connu à l'avance pour pouvoir utiliser le résultat de [86, 84, 83, 85]. Plus précisément, les chemins doivent déjà être déterminés à l'avance et un pré-calcul est effectué afin de trouver un ordonancement des mouvements sur ces chemins pour réaliser le routage (i.e. le schedule specifie quel paquet doit avancer et quel paquet doit attendre à chaque étape). En d'autres termes, le chemin suivi par chaque paquet lors de chaque communication doit être pré-calculé et stocké dans le processeur d'origine.

Il est vrai que si la structure de communication de G reste la même, il suffit de pré-calculer les chemins de routage une fois pour toutes. Dans le cas contraire, une méthode qui détermine les chemins de routage en direct (i.e. sans caclul préliminaire), ce qui plus difficile, est préferable.

Notons aussi que les algorithmes off-line ont deux défauts majeurs: d'une part, le pré-calcul peut être d'une durée importante (éventuellement exponentielle, l'algorithme pour produire l'ordonnancement donnée dans [86] pouvait être exponentielle pour trouver un ordonnancement avec une file d'attente de taille constante, ce dernier a été amélioré récemment dans [85]) et d'autre part le résultat de ce pré-traitement doit être localement stocké sur chaque processeur ce qui sous-entend qu'il faut prévoir une mémoire d'une taille conséquente.

Notons par ailleurs que la borne donnée par l'encadrement 6.1 est obtenue à partir d'une simulation en direct ne nécessitant aucun traitement préalable.

On conclut donc de ce qui est présenté ci-dessus qu'en fait, ce qui compte dans la pratique, c'est l'efficacité avec laquelle un algorithme de routage (ce que mesurent en partie la dilatation et la congestion) permet de simuler dans le graphe hôte H, le graphe logique G. En d'autres termes, un algorithme de plongement induit un algorithme de routage.

Dans ce chapitre, nous considérons le modèle des réseaux synchrones. Supposons que dans ce modèle qui impose un mode de fonctionnement synchrone, lors d'une étape de communication dans H, une arête du réseau ne peut être empruntée que par un seul message et que seules les communications dans une direction uniforme peuvent s'exécuer en parallèle. Par exemple, si le réseau est une grille où chaque processur est connecté à ses 4 voisins immédiats (au Nord, Sud, Est et Ouest), alors à une étape donnée, seuls les messages envoyés dans la même direction, par exemple au Nord, peuvent s'exécuter en

parallèle. La grille bidimensionnelle X-net de la machine massivement parallèle MasPar est un exemple d'un tel réseau.

On s'apperçoit donc que les notions de dilatation et congestion sont insuffisantes dès lors que l'on s'intéresse à un ensemble de communications synchrones et uniformes: une dilation d par exemple signifie qu'une communication se fait en au plus d étapes mais ce nombre peut être supérieur pour des communications que l'on souhaite exécuter en parallèle. Par exemple, une communication suivant un parcours S-S-S-S (S pour Sud) et une autre communication suivant un parcours N-E-E (N pour Nord et E pour Est) exigera $4 + 3 = 7$ étapes.

Dans ce qui suit, l'approche que nous utilisons pour plonger de manière dynamique un arbre quelconque de M sommets dans une grille de dimension quelconque de N sommets est différente de l'approche classique des méthodes de plongement. En effet, et en conséquence de ce qui a été présenté ci-dessus, après avoir préciser la fonction de projection, nous allons utiliser un algorithme de routage, afin d'assurer les communications inter-nœuds dans l'arbre plongé. Plus précisément, nous montrons comment plonger d'une manière dynamique, un arbre quelconque de telle sorte que la charge soit bien équilibrée et, au lieu d'*étirer* les arêtes de l'arbre et les projeter sur les chemins du réseau comme dans l'approche classique des plongements, nous allons utiliser un mécanisme de routage basé sur la notion de mouvement de données, afin de faire communiquer les nœuds de l'arbre plongé. Ainsi, nous allons utiliser le concept de mouvement de données de [35], décrit dans le chapitre précédent.

Rappelons que ce concept couvre un ensemble d'outils permettant de copier et de déplacer des données à travers les processeurs d'une machine parallèle synchrone. De plus, nous allons montrer qu'il est possible d'optimiser l'utilisation de ces outils si certaines contraintes sont vérifiées. Nous montrons dans le chapitre suivant, que ces contraintes peuvent être facile à satisfaire. Rappelons aussi que les algorithmes de routages de [35] s'effectuent en direct (on-line). En effet, chaque processeur est capable de décider localement du devenir d'un enregistrement sans pré-traitement. Ainsi, chaque processeur décide du devenir d'un enregistrement reçu en prenant en compte uniquement l'information representée par la destination attachée et progressant avec l'enregistrement, et l'information locale dont il dispose représentée par son numéro.

Il faut aussi noter de plus, pour comprendre les raisons pour lesquelles on a opté pour le routage au lieu de l'étirement, que l'on veut plonger des arbres quelconques et pas nécessairement binaires, comme c'est le cas souvent dans la littérature [81, 17]. De plus les réseaux considérés sont de type grille multi-dimensionnelle incluant la grille de dimension 2 et l'hypercube souvent privilégiés pour les plongements des arbres binaires complets ou quelconques. A première vue, il semble qu'il suffit de vérifier simplement que les algorithmes de plongement décrits dans la littérature et destinés aux hypercubes peuvent être mis en œuvre sur les grilles de dimension quelconque directement ou après de légères modifications. Il est vrai que l'hypercube est capable de simuler toute grille et tout arbre binaire avec un facteur de ralentissement constant d'après les plongements de grilles et d'arbres binaires décrits dans [81] et de ce fait on déduit immédiatement que l'hypercube est au moins aussi puissant que les grilles et les arbres réunis. Malheureusement, les algorithmes de plongement d'arbres déjà décrits pour l'hypercube concernent essentiellement

les arbres binaires et ils sont fondés sur des relations intéressantes entre les arbres binaires et les hypercubes d'une part, et d'autre part sur les propriétes remarquables de l'hypercube. Par exemple, l'arbre binaire complet à double racine, une légère modification de l'arbre binaire complet, est un sous graphe de l'hypercube, ou bien encore, une numérotation infixée d'un arbre binaire complet de N sommets induit un plongement de dilatation 2 dans un hypercube de N sommets. Parmi les propriétes remarquables de l'hypercube qui possède toutes les bonnes propriétés généralement souhaitées d'où sa popularité citons, son faible diamètre en $logN$, chaque sommet posséde $logN$ voisins dans un hypercube de N sommets, une large bissection, une structure récursive, ...

Ce chapitre est divisé en 4 sections. Dans la section 5.2, nous commençons par analyser tout d'abord le comportement des procédures de mouvement de données et nous verrons comment optimiser l'utilisation de ces procédures pour résoudre des problèmes de routage particuliers sans pour autant altérer le résultat. Nous définirons ensuite dans la section 5.3 deux méthodes de plongement dynamique d'arbres. La première méthode de plongement est une méthode déterministe et qui autorise le mécanisme de migration de processus (i.e., les sommets de l'arbre). La deuxième méthode de plongement est une méthode probabiliste et sera présentée dans le chapitre suivant. Les deux méthodes utilisent les procédures de mouvement de données, dans une architecture synchrone, afin de répondre au problème du routage dans les algorithmes de plongement que nous voulons construire pour assurer les communications inter-nœuds dans les arbres plongés. Dans la section 5.4, nous allons analyser l'impact des algorithmes de plongement sur l'utilisation des procédures de mouvement de données par les opérations de base de manipulation des arbres. Plus précisément, nous allons établir les contraintes à vérifier par les algorithmes de plongement afin de rendre posssible l'utilisation optimisée des procédures de mouvement de données pour les communications inter-sommets dans l'arbre plongé.

Notons que puisque dans ce chapitre, le plongement d'arbres est vu au travers des communications parallèles entre sommets des arbres plongés à l'aide des opérations de mouvement de données, les résultats démontrés dans ce chapitre concernent autant le plongement d'arbres statiques que celui du plongement d'arbres dynamiques.

5.2 Optimisation des mouvements de données

Utiliser directement l'approche de Nassimi et Sahni [35] décrite dans le chapitre précédent pour construire à partir des procédures de mouvement des données un algorithme de routage, nécessiterait un appel respectif de ces procédures dans un ordre bien défini. Ceci dans le but d'assurer l'arrivée des données à leurs bonnes destinations finales.

En effet, comme il a été dit dans le chapitre précédent, Nassimi et Sahni [35] utilisent, pour résoudre un problème de routage aléatoire, un algorithme de routage qui se décompose dans sa forme générale en deux étapes. Lors de la première étape, les enregistrements sont triés en fonction de leurs destinations. Une fois les enregistrements triés, on se retrouve avec un problème de routage monotone. Un problème de routage monotone se résout lui même en deux phases. La première phase effectue une opération de concentration des enregistrements. Une opération de concentration comme elle est définie communément dans [35, 81]

consiste à router les enregistrements vers les premiers processeurs contigus de la machine. Les indices de ces premiers processeurs sont calculés en utilisant la procédure Rank. L'opération de concentration est donc composée des deux procédures Rank+Concentrate. La deuxième phase suivante effectue une opération de distribution afin de redistribuer les enregistrements. Une opération de distribution comme il est défini dans [35, 81] consiste à router l'ensemble des enregistrements stockés dans les premiers processeurs contigus de la machine vers leurs destinations finales. L'opération de distribution est constituée de la procédure Distribute ou de la procédure Generalize. Le clé du succés de cet algorithme de routage est qu'il n'y a jamais de conflits entre les enregistrements. Cela est dû à la phase de tri qui convertit le problème de routage initial en un problème de routage monotone, et aux phases suivantes constituées de deux vagues successives, une vague de concentration et une vague de redistribution.

L'interêt et le respect de ces trois étapes sont donc nécessaires pour router toutes les données vers leurs destinations finales. Cet argument est vrai dans le cas général de routage aléatoire. Cependant, les enregistrements peuvent être placés sur les processeurs selon une stratégie bien définie (par exemple, des enregistrements qui représentent les nœuds d'un arbre sont répartis sur les processeurs selon la fonction de projection bien définie du plongement). Faut-il donc effectuer systématiquement toutes ces étapes de routage?

En analysant soigneusement le protocole suivi par un algorithme de routage selon l'approche de [35] pour la résolution des conflits, et en examinant les preuves de corrections des procédures de mouvement de données [36], nous allons montrer dans ce qui suit qu'il est possible de réduire la liste des étapes à faire pour effectuer un routage.

Commonçons par rappeler comment les algorithmes des procédures de mouvement de données, et particulièrement ceux des procédures Concentrate et Distribute, sont mises en œuvre pour envoyer des données vers leurs destinations en parallèle dans une grille multi-dimensionnelle. L'acheminement des enregistrements se fait de telle sorte qu'à chaque étape, au plus un enregistrement emprunte une arête. De plus, seules les arêtes d'une même dimension sont empruntées à chaque étape, et que des arêtes de dimensions consécutives sont empruntées au cours d'étapes consécutives. Rappelons que deux processeurs sont dits *dans la même dimension k* si les repésentations binaires de leurs numéros ne diffèrent que par leur k-ième bit.

Notons que cette notion de voisinage au sens des algorithmes de mouvement de données de [35] se matérialise par une liaison physique dans le cas d'un hypercube (i.e., le cas particulier d'une grille multi-dimensionnelle de côté 2). Tandis que pour une grille de manière générale, une connexion entre un couple de sommets au sens des algorithmes de [35] ne traduit pas forcément un voisinage physique. Notons aussi que cette notion de voisinage logique permet d'utiliser les mêmes algorithmes sans modifications sur un hypercube et sur une grille de dimension quelconque. En effet, ces algorithmes sont mis en œuvre sur une grille en numérotant les processeurs suivant la numérotation en ligne d'abord ou en mélange de ligne d'abord, comme cela a été déjà présenté dans le chapitre 4. Bien sûr, le coût d'exécution n'est pas le même sur la grille et l'hypercube, et il est en fonction du diamètre du réseau utilisé.

Nous résumons cette propriété dans la définition suivante afin d'y faire référence par la

suite.

Définition 2 *Une grille multi-dimensionnelle est dite bien numérotée si pour tout couple de sommet i et j, il existe une connection entre i et j dans la grille si et seulement si i et j ne diffèrent que d'un seul bit.*

Comme cela a été déjà mentionné dans le chapitre précédent, la procédure Concentrate est mise en œuvre de la manière suivante. Soit $r(i)$ le rang calculé par la procédure Rank de l'enregistrement se trouvant dans $P(i)$. Ce numéro de processeur indique que cet enregistrement doit être concentré vers le processeur $P(r(i))$. Formellement, la procédure Concentrate commence par déplacer les enregistrements aux processeurs dont le bit du poids faible 0 de l'indice est identique à celui de $r(i)$. Ensuite, aux processeurs dont les bits 0 et 1 de son indice sont identiques à ceux de $r(i)$ et ainsi de suite jusqu'à ce que les enregistrements rejoignent leurs processeurs destinataires (i.e., la progression continue jusqu'à ce que les enregistrement rejoignent les processeurs d'indice $r(i)$). Par exemple, pour que l'enregistrement de $P(6 = 110)$ rejoint son processeur destination $P(3 = 011)$, il est d'abord router vers $P(111)$ puis vers $P(011)$.

La procédure Distribute effectue l'opération inverse de Concentrate et s'exécute de la manière suivante. Soit $d(i)$ la destination de l'enregistrement se trouvant dans le processeur $P(i)$. Ce numéro de processeur indique que cet enregistrement doit être routé vers le processeur $P(d(i))$. Formellement, la procédure Distribute commence par déplacer les enregistrements aux processeurs dont le bit du poids fort $\log N - 1$ de l'indice est identique à celui de $d(i)$. Ensuite, aux processeurs dont les bits $\log N - 1$ et $\log N - 2$ de son indice sont identiques à ceux de $d(i)$ et ainsi de suite jusqu'à ce que les enregistrements rejoignent leurs processeurs destinataires (i.e., la progression continue jusqu' à ce que les enregistrement rejoignent les processeurs d'indice $d(i)$). Par exemple, pour que l'enregistrement de $P(3 = 011)$ rejoint son processeur destination $P(6 = 110)$, il est d'abord router vers $P(111)$ puis vers $P(110)$.

5.2.1 Les collisions dans une grille

Il y a deux situations possibles dans une grille, indiquées dans la figure 5.1, dans lesquelles deux enregistrements peuvent se trouver dans un même processeur au même moment. Dans la première situation, une collision a lieu si à une étape donnée du routage, un enregistrement qui ne se déplace pas durant cette étape, est écrasé par un autre envoyé dans le même processeur. Dans la deuxième situation, deux enregistrements se déplacent et se rejoignent dans un même processeur.

En reprenant ce qui a été présenté ci-dessus, on s'aperçoit que la situation (2) ne peut survenir pour les raisons suivantes:

1. du point de vue numérotation: dans une grille bien numérotée, un processeur P_k ne peut avoir en même temps deux processeurs P_i et P_j, avec $i \neq j$, dont les représentations binaires diffèrent de celle de P_k au même bit. La situation (2) ne peut donc survenir si on utilise un algorithme (Concentrate par exemple) dans lequel à chaque

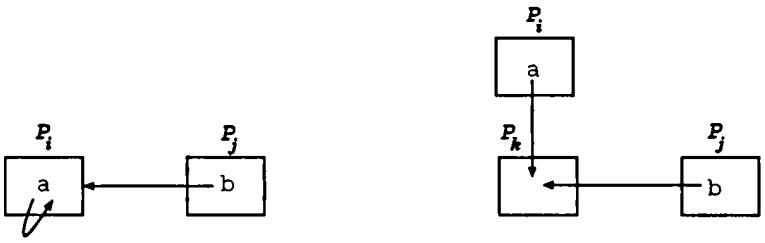

FIG. 5.1 - *Deux situations de collisions possibles durant une étape donnée. Dans la situation (1), l'enregistrement a reste sur place et l'enregistrement b va se dépalcer vers le processeur P_i. Dans la situation (2), a et b se déplacent tous les deux vers un processeur P_k.*

étape ℓ, un enregistremnt n'est autorisé à traverser une arête que si les représentations binaires des numéros des processeurs au deux bouts diffèrent du même bit ℓ.

2. du point de vue réseau uniforme: lors d'un routage synchrone, la situation (2) ne peut survenir puisque les deux communications possèdent deux directions différentes et elles ne peuvent donc être effectuées simultanément.

On déduit donc que dans l'un ou l'autre cas, le seul cas possible de collision est la situation (1).

Nous avons besoin pour la suite du lemme élémentaire suivant qui fournit les bases de notre analyse. Ce lemme indique le fait qu'étant donné deux mots binaires u et v de longeur k (k bits), si les derniers b bits de u et v sont identiques alors on a $|u-v| \geq 2^{b+1}$. Par contre, si c'est les premiers b bits de u et v qui sont identiques, alors on a $|u-v| < 2^{b+1}$.

Lemme 5 *Soient deux processeurs u et v où*
$u = u_{k-1} \ldots u_{b+1} u_k u_{b-1} \ldots u_1 u_0$ *et* $v = v_{k-1} \ldots v_{b+1} v_k v_{b-1} \ldots v_1 v_0$

1. *si* $u_i = v_i$, $\forall\ 0 \leq i \leq b$, *alors on a* $|u-v| \geq 2^{b+1}$.
2. *si* $u_i = v_i$, $\forall\ b \leq i \leq k-1$, *alors on a* $|u-v| < 2^{b+1}$.

Nous considérons dans ce qui suit deux types particuliers de problème de routage. Soient $S = \{ i_1, \ldots, i_p \}$ l'ensemble des processeurs sources et $D = \{ d(i_1), \ldots, d(i_p) \}$ l'ensemble des processeurs destinations du problème de routage. On note par $C = \{ i_1 \rightarrow d(i_1), i_2 \rightarrow d(i_2), \ldots, i_p \rightarrow d(i_p) \}$ l'ensemble de communications simultanées à effectuer pour résoudre le problème, sachant que si $i_k \neq i_\ell$ alors $d(i_k) \neq d(i_\ell)$. Les deux types particuliers de problème de routage sont énoncés dans les deux définitions suivantes.

Définition 3 *On dit qu'un ensemble de communications C est réducteur si*

$$\forall\ i, j \in S \quad |i-j| \geq |d(i) - d(j)|.$$

Définition 4 *On dit qu'un ensemble de communications C est expansif si*

$$\forall\ i,j \in S \quad |i-j| \leq |d(i)-d(j)|.$$

On désigne par R^+ l'algorithme de routage correspondant à celui de la procédure Concentrate. R^+ est donc mise en œuvre de la manière suivante. Un enregistrement originaire du processeur i est envoyé au cours d'une étape b, dans l'ordre $b = 0, 1, \ldots, \log N - 1$, vers le processeur $i^{(b)}$, si i et $d(i)$ diffèrent par leur b-ième bit.

Définition 5 *On note R^+ l'algorithme de routage qui traite les bits des numéros de processeurs de la droite vers la gauche (de 0 à $\log N - 1$).*

De la même manière, on designe par R^- l'algorithme correspondant à celui de la procedure Distribute. Rappelons que la procédure Distribute effectue l'opération inverse de Concentrate. Un enregistrement progresse au cours des étapes b allant de $\log N - 1, \ldots, 0$.

Définition 6 *On note R^- l'algorithme de routage qui traite les bits des numéros de processeurs de la façon inverse (de $\log N - 1$ à 0).*

Soit un problème de routage dont l'ensemble de communication est C. Nous allons montrer dans ce qui suit que si C est réducteur, alors il suffit d'utiliser le routage R^+ pour résoudre le problème. Plus précisément, nous allons montrer que le routage R^+ s'effectue sans collision et que chaque enregistrement atteint sa bonne destination finale.

Theorème 2 *Le routage par R^+ d'un ensemble de communications réducteur est sans collision*

preuve.

Soient $i = i_{k-1} \ldots i_0$ et $j = j_{k-1} \ldots j_0$ deux processeurs quelconques de la grille appartenant à S. i et j contiennent deux enregistrements à destination des processeurs $d(i)$ et $d(j)$. A l'étape b du routage R^+, les deux communications sont arrivées respectivement dans les deux processeurs intermediares p et q telsque

$$p = (i_{k-1:b+1}, d(i)_{b:0}) \text{ et } q = (j_{k-1:b+1}, d(j)_{b:0}).$$

En effet, après avoir été deplacé si nécessaire au cours des étapes $0, 1, \ldots, b-1$, un enregistrement originaire d'un processeur ℓ et de destination $d(\ell)$ arrive dans un processeur intermediaire r, telsque les b premiers bits de r sont identiques au b premiers bits de sa destination $d(\ell)$.

On a donc la présence de collisions dans un processeur à cette étape si $p = q$ et par conséquent on a

$$\begin{array}{ll} i) & i_{k-1:b+1} = j_{k-1:b+1} \\ ii) & d(i)_{b:0} = d(j)_{b:0} \end{array}$$

D'après le lemme 5, (i) implique que $|i-j| < 2^{b+1}$ et (ii) implique que $|d(i)-d(j)| \geq 2^{b+1}$. Par conséquent $|i-j| < |d(i)-d(j)|$. Or ceci contredit l'hypothèse d'un ensemble de communications réducteurs pour lequel $\forall\ i,j \in S,\ |i-j| \geq |d(i)-d(j)|$. On déduit donc

que le routage R^+ s'effectue sans collision. La figure 5.2 illustre un exemple d'emploi du routage R^+ (i.e., la procédure Concentrate) qui s'exécute sans conflits pour résoudre le problème de routage $C = \{ 3 \rightarrow 1, 6 \rightarrow 3\}$.

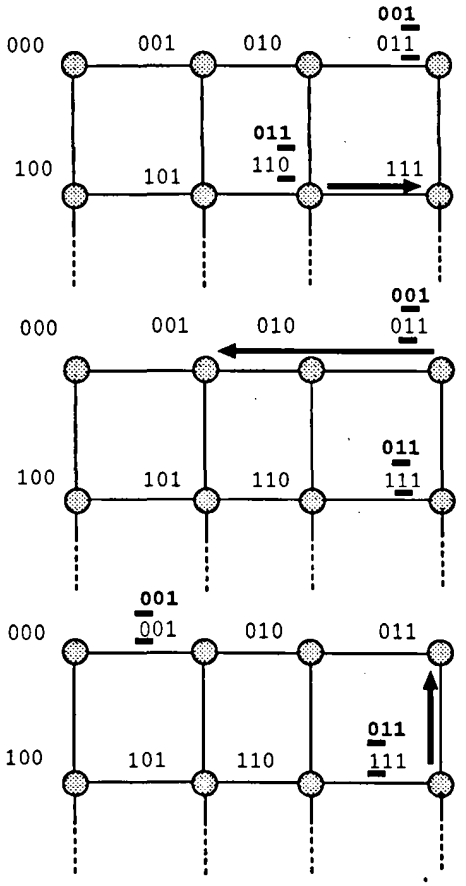

FIG. 5.2 - *Un exemple de routage R^+. Chaque enregistrement dans i emprunte un chemin pour rejoindre sa destination $d(i)$ de la manière suivante. Durant l'itération b, le b-ième bit de l'indice de son processeur initial i est remplacé par le b-ième bit de $d(i)$. Si les deux bits sont identiques, l'enregistrement reste sur place durant cette itération. Par contre, si les deux bits sont différents, l'enregistrement traverse l'arête vers le processeur dont le b-ième est complémentaire de celui de i. Notons qu'il n'y a jamais de collision entre deux enregistrements dans un même sommet.*

Le corollaire au théorème 2 ci-dessous traduit l'absence de conflits quand un algorithme suivant le protocole de [35] est utilisé.

Corollaire 1 *Dans les mouvements de données de [35], quand on applique la procédure Concentrate après la procédure Rank pour des destinations ordonnées, on est dans la situation suivante :*

- *l'ensemble des communications est réducteur car les destinations sont* $0, 1, \ldots, r$

- *la procédure Concentrate utilise le routage* R^+

Un routage s'effectue donc sans collision.

preuve.

Rappelons qu'un problème de routage selon la méthode de Nasimi et Sahni se réduit à un problème de routage monotone qui se résout en appliquant une opération de concentration suivie d'une opération de distribution. Ainsi, un algorithme de routage monotone qui permet d'envoyer des données vers leurs destinations en parallèle (une opération RAW par exemple) est mis en œuvre de la manière suivante. Dans un premier temps, on calcule le rang des données en utilisant la procédure Rank. Puis on concentre ces données vers les premiers processeurs de la machine en utilisant la procédure Concentrate de façon à ce que les données de rang i se retrouve sur le processeur i. La procédure Concentrate utilise ensuite un routage R^+. Une fois les données sont concentrées, il suffit alors de les router vers leurs destinations finales en utilisant la procédure Distribute. On ne peut avoir de collisions lors de l'opération de concentration si on procède de la sorte puisque les destiniations $d(i)$ et $d(j)$, pour tout i, j de S, sont donnés par la procédure Rank et donc $0 \leq d(i) < d(j) < r \leq N$ *et* $0 \leq i < j \leq N$. L'ensemble de communication lors de l'opération de concentration est réducteur puisqu'on a forcément la contrainte $|i - j| \geq |d(i) - d(j)|$.

Si maintenant on considère que la contrainte reste vérifiée, on n'aura pas à appeler la procédure Rank et par suite la procédure Distribute, Concentrate sera suffisante à elle seule. Plus précisément, si l'ensemble des communcations est réducteur, on a la garantie qu'il n'y a jamais de collision entre deux enregistrements dans un processeur en utilisant un routage R^+. On peut alors réduire l'algorithme de routage en invoquant uniquement la procédure Concentrate.

Nous allons maintenant montrer que si l'ensemble de communication C est expansif, alors il suffit d'utiliser le routage R^- pour résoudre le problème de routage. Plus précisément, montrons que le routage R^- s'effectue sans colision et que chaque enregistrement rejoint sa bonne destination finale.

Theorème 3 *Le routage par* R^- *d'un ensemble de communications expansif est sans collision*

preuve.

La preuve est similaire au cas du routage R^+. Si un enregistrement, originaire d'un processeur ℓ et de destination $d(\ell)$, rejoint un processeur intermediaire r, il le fait après avoir été deplacé, si nécessaire, au cours des étapes $log\, N - 1, log\, N - 2, \ldots, b - 1$. Les b premiers bits de r doivent être identiques au b premiers bits de sa destination $d(\ell)$.

Soient i et j deux processeurs originaires de deux enregistrements à destination des processeurs $d(i)$ et $d(j)$. A l'étape b du routage R^-, les deux communications arrivent respectivement dans les processeurs intermédiaires

$$p = (d(i)_{k-1:b}, i_{b-1:0}) \text{ et } q = (d(j)_{k-1:b}, j_{b-1:0}).$$

On aura la présence de collision durant l'itération b si $p = q$:

$$i)\ d(i)_{k-1:b} = d(j)_{k-1:b}$$
$$ii)\ i_{b-1:0} = j_{b-1:0}$$

D'après le lemme 5, (i) signifie que $|i - j| \geq 2^b$ et (ii) signifie que $|R(i) - R(j)| < 2^b$. On a alors $|R(i) - R(j)| < |i - j|$ ce qui mène à une contradiction puisque l'ensemble des communications est expansif. La figure 5.3 illustre un exemple d'emploi de la procédure Distribute.

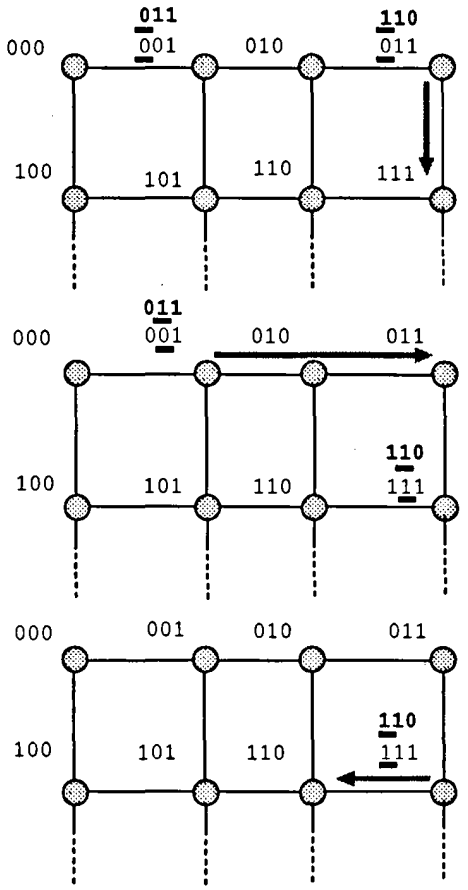

FIG. 5.3 - *Un exemple de Distribution. Chaque enregistrement dans i emprunte un chemin pour rejoindre sa destination d(i) de la manière suivante. Durant l'itération b, le b-ième bit de l'indice de son processeur initial i est remplacé par le b-ième bit de d(i). Si les deux bits sont identiques, l'enregistrement reste sur place durant cette itération. Par contre, si les deux bits sont différents, l'enregistrement traverse l'arête vers le processeur dont le b-ième est complémentaire de celui de i. Notons qu'il n'y a jamais de collision entre deux enregistrements dans un même sommet.*

Le corollaire au théorème 3 ci-dessous traduit l'absence de conflits lors d'une opération de

distibution quand on utilise un algorithme suivant le protocole de [35] est utilisé.

Corollaire 2 *Dans les mouvements de données de [35], la procédure Distribute s'applique après la procédure Concentrate :*

- *elle utilise le routage* R^-
- *l'ensemble des communications est expansif puisque les sources sont* $0, 1, \ldots, r$ *et sont ordonnées selon les destinations.*

En conséquence, le routage se fait sans collision.

preuve.

Dans un algorithme de routage suivant la méthode de Nassimi et Sahni, la procédure Distribute est employée une fois les enregistrements sont concentrés sur les premiers processeurs de la machine d'une manière consecutive, suite à l'appel de la procédure Rank et à l'appel de la procédure Concentrate ou bien siute à l'appel de la procédure Sort. Plus précisément, dans les processeurs ℓ, $0 \leq \ell < N$, les valeurs des destinations des enregistrements $d(\ell)$ sont telles que $0 \leq d(0) < d(1) < \ldots < d(r) \leq N - 1$. On a alors $|d(i) - d(j)| \geq |i - j|$. On ne peut donc y avoir de collisions.

Basé sur les observations réunies dans les deux théorèmes ci-dessus, on conclut donc qu'au lieu d'appeler une suite convenable des procédures de mouvement de données pour résoudr un problème de routage quelconque, on peut se contenter de faire appel uniquement à la procédure Concentrate (i.e. le routage R^+) si l'ensemble de communication est réducteur ou bien utiliser uniquement la procédure Distribute (i.e. le routage R^-) si l'ensemble de communication est expansif.

En d'autres termes, afin de pouvoir bénéficier d'une utilisation optimisée des procédures de mouvement de données, lors d'une opération d'acheminement d'enregistrements depuis des processeurs sources vers leurs proceseurs destinataires, il suffit de distribuer judicieusement ces enregistrements au départ dans les processeurs dans le but de rendre tout ensemble de communication à effectuer reducteur ou expansif.

Notre étude sur l'utilisation optimisée de l'approche de Nassimi et Sahni terminée, nous revenons au problème de plongement d'arbres quelconaues dans une grille multi-dimensionnelles.

5.3 Les stratégies de plongement

Rappelons que le plongement d'un arbre T de M sommets dans une machine parallèle S ayant N processeurs revient à la construction d'une fonction de projection qui affecte les sommets de T aux N processeurs. En d'autres termes, on est conduit à associer chaque sommet ou un groupe de sommets de cet arbre à un processeur donné en s'appuyant sur une forme que l'on spécifie en précisant la dite fonction de projection. De plus, le plongement doit être effectué en direct ou on-line (i.e., sans calcul préliminaire une fois pour toute) sinon l'arbre doit être connu au départ. Plus précisément, nous voulons ici que

le placement des sommets de l'arbre sur les processeurs puisse s'effectuer d'une manière dynamique sans tenir compte ni de la forme ni de la taille de l'arbre.

Un plongement dynamique peut être vu comme une succession de plongements dont chacun est une extension de son précédent. Nous ferons la différence entre deux stratégies différentes de plongement. La première tolère le principe de migration et procède donc en effectuant des pré-traitements intermédiaires avant que chaque nouveau plongement ne démarre afin de calculer les nouvelles positions des sommets à réarranger. La seconde procède en plongeant directement l'arbre sans effectuer des pré-calculs intermédiaires.

Dans ce qui suit, nous allons examiner les deux méthodes de plongement basés sur les deux types de fonctions de projection suivantes :

- ***fonction de projection irrégulière*** : Une manière de plonger un arbre est de numéroter les nœuds, suivant l'ordonnancement par niveau par exemple, et de placer le i-ième nœud de l'arbre sur le $i \bmod N$-ième processeur de la machine. Un algorithme de plongement utilisant cette fonction de projection irrégulière est basé sur le principe de migration puisque les numéros assignés aux sommets ne sont pas définitifs.

- ***fonction de projection régulière*** : la fonction de projection répartie les sommets de T dans des processeurs dont les numéros sont définis à l'avance et ils sont définitifs. Un algorithme de plongement utilisant cette fonction de projection régulière respecte le paradigme proposé par Bhatt et Cai [17] qui n'autorise pas la migration.

En utilisant l'une ou l'autre des deux méthodes de plongement, on remarque que les sommets de l'arbre sont distribués selon une fonction de projection bien définie. Plus précisément, les sommets ne sont pas distribués aléatoirement et nous sommes alors capable de ce fait, comme nous allons le voir ci-après, de vérifier s'il est possible de bénéficier d'une utilisation optimisée des opérations de mouvement des données.

Rappelons que puisqu'ici on a choisi de ne pas *étirer* les arêtes de l'arbre pour effectuer le plongement, la dilatation de ce dernier ne correspond pas à l'étirement maximum nécessaire pour effectuer le plongement mais correspond au coût du routage que nous allons utiliser pour mettre en œuvre les communications inter-sommets des arbres plongés. Rappelons aussi que le coût de ce routage, basé sur les procédures du mouvement de données, est mesuré en nombre d'étapes nécessaires pour router des données envoyées de leurs nœuds d'origines à leurs nœuds de destinations. La meilleure méthode de plongement est donc celle pour laquelle il est possible de réduire le nombre d'étapes de routage et de telle sorte que les sommets de l'arbre soient équitablement répartis sur les processeurs.

5.4 Application de l'optimisation au problème de plongement

Avant d'analyser dans ce chapitre la première stratégie de plongement, nous allons tout d'abord définir quelques opérations de bases nécessaires à la manipulation d'un arbre.

5.4.1 Les opérations de communication inter-sommets

On définit une opération primitive dans un arbre T comme étant un pas de communication (ou un pas de calcul) d'une unité de temps dans cet arbre. L'implémentation d'une primitive afin de manipuler l'arbre, dépend bien entendu de la topologie du graphe hôte sur lequel on plonge l'arbre et de la fonction de projection. Nous allons donc étudier les opérations primitives selon la fonction de projection choisie, et fournir ainsi les implémentations adaptées aux stratégies de ces fonctions de projection.

Les primitives de base, d'utilité intrinsèque pour manipuler un arbre T sont ceux qui permettent de réaliser les communications inter-sommets. Ces communications sont dictées par la structure de l'arbre elle-même. Il s'agit du routage d'informations depuis un fils vers son père *fils→père*, du père à un fils *père→fils*, de tous les fils à leur père *enfants→père* ou du père à ses enfants *père→enfants*. Notons que ces opérations s'exécutent soit dans un sens ascendant soit dans un sens descendant.

A première vue, il semble qu'il suffit d'implémenter ces opérations en utilisant une suite convenable des procédures de mouvement de données, par exemple Rank+Concentrate+Distribute ou bien Rank+Concentrate+Generalize. En utilisant les résultats de la section 5.2, nous allons voir dans ce qui suit qu'on peut utiliser, dans certains cas, une seule procédure sur les trois. Les coûts de ces opérations nous permettront donc de comparer les deux approches du plongement irrégulier et du plongement régulier.

5.4.2 analyse du plongement irrégulier

Une manière de plonger un arbre quelconque (i.e., binaire ou non) de M sommets dans un hypercube ou une grille de N processeurs, avec $M \leq N$, est de numéroter ses sommets, suivant l'ordonnancement par niveau par exemple, et de placer le i-ième sommet de l'arbre sur le i-ième processeur de la grille ou de l'hypercube. Bien entendu, on peut numéroter les sommets de l'arbre dans un autre ordre. Mais la numérotation suivant l'ordonnancement par niveau est choisie pour des raisons liées aux procédures des mouvements de données.

En effet, dans une numérotation d'un arbre de $M \leq N$ sommets suivant l'ordonnancement par niveau, un sommet est étiqueté après que tous les sommets gauches appartenant au même niveau l'aient été, mais avant que leurs descendants ne le soient. Les étiquettes sont données dans l'ordre 0, 1, ..., $N-1$. On appelera le processeur dans lequel est stocké un sommet, le *représentant* de ce sommet. Cette numérotation implique qu'implémenter une des opérations de bases décrites ci-dessus consiste à résoudre un problème de routage monotone étant donné qu'il n'est pas utile d'effectuer de tri. De plus, les fils d'un même sommet de l'arbre sont placés sur des processeurs de numéros successifs ce qui rend plus aisée la résolution des problèmes de concentration partielle et de diffusion partielle.

Lorsque le nombre M de sommets de l'arbre est plus grand que le nombre N des processeurs, alors le i-ième sommet de l'arbre est placé sur le i modulo N-ième processeur. Ainsi, puisque la numérotation par niveaux distribue les sommets de l'arbre sur les processeurs suivant l'ordre 0, 1, ..., $N-1$ et d'une manière cyclique en utilisant l'opérateur modulo, chaque processeur prend donc en charge $\lceil \frac{M}{N} \rceil$ sommets. Il doit être donc requis de pouvoir utiliser les procédures de mouvement de données quand $M \geq N$ (i.e., chaque processeur

contient $\lceil M/N \rceil$ sommets). Comme cela a été mentionné dans le chapitre précédent, la méthode consiste à utiliser les procédures de mouvement de données en transformant le problème de routage en $\frac{M}{N}$ problèmes de routage injectif.

L'algorithme parallèle de Branch and Bound (B&B) dans un hypercube à grain fin qui a été proposé dans [43, 2] représente un exemple d'utilisation de la numérotation suivant l'ordonnancement par niveau. En effet, l'arbre de l'algorithme B&B de N sommets a été implémenté sur l'hypercube de N sommets en numérotant les sommets de l'arbre suivant l'ordonnancement par niveau. Les auteurs utilisent également les procédures de mouvement de données de [35] pour la mise à jour de l'arbre au cours de l'agorithme. Seulement, ils utilisent ces procédures comme un modèle ou un outil monolithique alors que nous ici nous cherchons à analyser ce modèle afin d'optimiser son utilisation. De plus, les auteurs se sont restreints au cas où le nombre M de sommets de l'arbre est inférieure ou égale au nombre N des processeurs. En fait, le problème de la mise en œuvre de l'algorithme B&B sur un hypercube SIMD à été traité par les auteurs comme étant un problème de simulation d'une PRAM sur une machine à mémoire distribuée. Ainsi, le fait que le nombre M de sommets de l'arbre dépasse le nombre de processeurs disponibles est traduit par une insuffisance ménoire.

Le plongement irrégulier se caractérisent par les trois propriétés suivantes:

- un ordre fixe est établi sur les sommets de l'arbre de gauche à droite en partant de la racine.
- un sommet n'a cependant pas un numéro fixe et définitif. Si des sommets sont ajoutés ou supprimés dans l'arbre, un réaménagement de cet arbre s'impose. Les adresses des nouveaux représentants sont calculées, et des sommets doivent alors être déplacés vers leurs nouveaux représentants. Ce réarrangement a pour but de préserver la relation d'ordre due à la fonction de projection basée sur la numérotation suivant l'ordonnancement par niveau.
- la charge des processeurs est identique à un près puisque chaque processeur prend en charge $\lceil M/N \rceil$ sommets, elle est donc optimale. Ceci est l'effet de la numérotation par niveaux et du principe de réaménagement des sommets (i.e., migration des sommets) selon le schéma du stockage de la fonction de projection.

Dans ce qui suit, nous explorons la relation existante entre l'optimisation des opérations de mouvements de données et la fonction de projection du plongement irrégulier. Plus précisément, nous décrivons la mise en œuvre des opérations de base en montrant les situations qui permettent d'optimiser leur implémentation. Afin de bien analyser ce plongement, on va distinguer deux cas, le cas des arbres complets et celui des arbres quelconques.

5.4.2.1 Cas des arbres complets

Soient N processeurs numérotés de 0 à $N-1$. Un arbre d-aire complet de M sommets peut être plongé dans les N processeurs de la manière suivante: le sommet racine est dans le processeur d'indice 1, et pour chaque sommet dans i, ses fils de la gauche vers la droite

sont respectivement dans $d \times i \, mod \, N, (d \times i + 1) mod \, N, \ldots, (d \times i + d - 1) mod \, N$. La figure 5.4 illustre un exemple de plongement d'un arbre binaire complet.

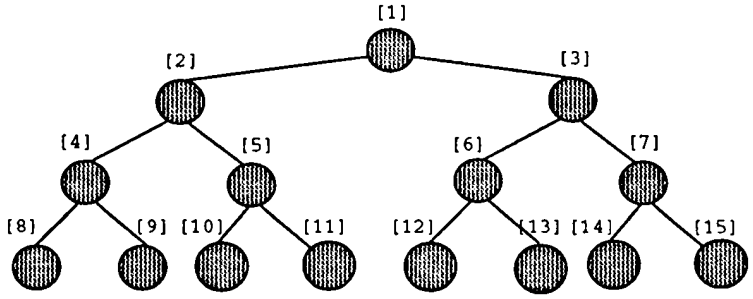

FIG. 5.4 - *Plongement d'un arbre binaire complet de $N-1$ sommets dans une grille ou un hypercube de N processeurs. Les étiquettes des sommets de l'arbre désignent les processeurs dans lesquels ils sont projetés. Plus précisément, le i-ième sommet de l'arbre est projeté sur le i-ième processeur, et chaque sommet interne i a ses fils gauche et droit dans $2 \times i$ et $2 \times i + 1$, respectivement.*

En appliquant directement l'approche de Nassimi et Sahni [35], l'appel respectif des trois procédures Rank, Concentrate et Distribute est nécessaire pour effectuer le routage d'informations, par exemple, lors des opérations ascendantes (i.e., fils→père et enfants→père). En reprenant ce que l'on a vu dans la section 5.2, nous allons montré cependant qu'il n'est pas nécessaire d'utiliser impérativement ces trois procédures. En effet, pour la mise en œuvre des opérations ascendantes, un appel à la procédure Concentrate suffit quand le plongement est utilisé dans le cas d'un arbre complet.

Plus précisément, montrons qu'aucune collision ne peut parvenir lors de ces opérations. Ceci revient à montrer que l'ensemble de communication induit par les opérations est réducteur et que lorsque l'algorithme R^+ est employé, le problème de routage se résout sans conflits.

Theorème 4 *Dans le cas d'un plongement irrégulier dans une grille multi-dimensionnelle d'un arbre complet, le routage R^+ s'effectue sans collision. En conséquence, un seul appel à ce dernier suffit pour la mise en œuvre des opérations ascendantes.*

Pour simplifier, et sans perte de généralité, nous supposons que l'arbre complet est binaire. Nous supposons aussi que le nombre de sommets est inférieur au nombre de processeurs (nous allons voir l'extension plus loin dans la section 5.4.2.5). D'après le plongement que l'on vient de décrire de l'arbre complet, un sommet v à ses 2 fils respectifs dans $2 \times v$ et $2 \times v + 1$ pour $v > 0$.

Soient i et j les représentants de deux sommets données de l'arbre. Notons par $d(i)$ et $d(j)$ les deux représentants de leurs parents respectifs (i.e., les parents des sommets se trouvant dans i et j sont dans les processeur $d(i)$ et $d(j)$ respectivement). On sait que le père du

nœud se trouvant dans i est dans $\lfloor \frac{i}{2} \rfloor$. On a donc :

$$|d(i) - d(j)| = |\lfloor \frac{i}{2} \rfloor - \lfloor \frac{j}{2} \rfloor| \leq \frac{1}{2}|i - j|.$$

On a donc bien $|d(i) - d(j)| \leq |i - j|$. Cette inégalité est vérifiée, on déduit que l'ensemble de communication induit par une opération fils→père est réducteur.

D'après le théorème 2, l'exécution d'un routage R^+ se fera donc sans conflit. On peut donc appliquer directement la procédure Concentrate pour la mise en œuvre de l'opération. Notons par ailleurs qu'aucune hypothèse n'a été posée sur les sommets présents dans i et j qui ne sont pas forcément dans le même niveau dans l'arbre.

Dans le cas de l'opération enfants→père, on peut également se contenter de faire appel à la procédure Concentrate. Durant l'exécution de la procédure, des "bons" conflits seront déclenchés comme il y a des enregistrements qui sont routés vers le même processeur. Ainsi, quand deux enregistrements arrivent à un même processeur, ils seront combinés au moyen d'une opération de cumulation associative donnée comme cela a été montionné dans le chapitre précédent.

Dans le cas des opérations descendantes inverses, montrons également qu'un appel à la procédure Distribute est suffisant pour mettre en œuvre ces opérations. Plus précisément, montrons que l'ensemble de communication induit par ces opérations est expansif. Ainsi, lorsque l'algorithme R^- est employé, le problème de routage se résout sans conflits.

Theorème 5 *Dans le cas d'un plongement irrégulier dans une grille multi-dimensionnelle d'un arbre complet, le routage R^- s'effectue sans collision. En conséquence, un seul appel à ce dernier suffit pour la mise en œuvre des opérations descendantes.*

Pour simplifier, et sans perte de généralité, nous supposons ici aussi que l'arbre complet est binaire, et que le nombre de sommets est inférieur au nombre de processeurs. On sait que le fils gauche d'un sommet se trouvant dans i est dans $d(2i)$ et que son fils droit est dans $d(2i+1)$. On a donc :

$$|d(i) - d(j)| = |2i + \tau - 2j - \tau'| \ \ avec\ \tau, \tau' \in \{0, 1\}.$$

on a alors

$$|d(i) - d(j)| = |2(i - j) + a| \ \ avec\ a \in \{-1, 0, 1\}.$$

On en déduit que $|d(i) - d(j)| \geq |i - j|$. L'ensemble de communication est donc expansif. D'après le théorème 3, l'exécution d'un routage R^- se fera sans conflit. On conclut donc que lors d'une opération père→fils, on peut se contenter de faire appel uniquement à la procedure Distribute dans le cas d'un arbre binaire complet.

Dans le cas de l'opération de routage père→enfants, on peut également se contenter de faire appel à la procédure Generalize. Et comme cela a été montré dans le chapitre précédent, cette procédure procède comme la procédure Distribute (i.e, effectue un routage R^- mais en plus, distribue la tâche de la copie de la donnée qu'un sommet père désire diffuser à tous ses fils sachant que ces derniers sont regroupés dans des processeurs consécutifs. Cette propriété est assurée par le plongement irrégulier qui , par construction, numérote consécutivemnet les fils d'un même sommet.

5.4.2.2 Cas des arbres quelconques

Nous avons montré que dans le cas des arbres binaires complets, on peut réduire le nombre d'étapes nécessaires pour réaliser les opérations de communication inter-nœuds de l'arbre plongé dans un grille de dimension quelconque. Nous allons examiner dans cette section le cas des arbres binaires quelconques.

Notons qu'un ordre fixe est défini sur les sommets de l'arbre de gauche à droite en partant de la racine. Cette relation d'ordre est dictée par la fonction de projection du plongement irrégulier. La figure 5.5 montre un exemple du plongement irrégulier d'un arbre binaire quelconque.

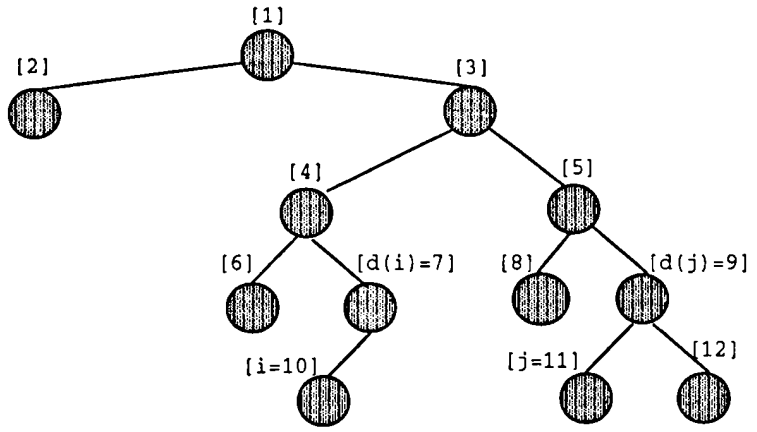

FIG. 5.5 - *Plongement d'un arbre binaire quelconque. Les étiquettes des sommets de l'arbre désignent les processeurs dans lesquels ils sont projetés. Notons que les fils gauche et droit s'ils existent d'un sommet interne i ne se reçoivent pas nécessairement Les étiquettes $2i$ et $2i+1$.*

Considérons le cas des opérations ascendantes, fils→père et enfants→père. Soient d(i) et d(j) les deux représentants des parents respectifs des sommets se trouvant dans i et j. On sait que cette opération peut être mis en œuvre d'une manière optimisé en utilisant seulement un routage R^+ (i.e, un appel à la procédure Distribute) si l'ensemble de communication est expansif. En d'autres termes, on sait que l'on a absence de "mauvais" conflits lors du routage R^+ si la contrainte $|d(i) - d(j)| \leq |i - j|$ est vérifiée.

Supposons pour simplifier que $i < j$ et $d(i) < d(j)$. Supposons aussi sans perte de généralité que l'arbre est binaire. Tout sommet v appartenant à l'intervalle $[d(i)..d(j)]$ à ses fils gauche ou droit, s'ils existent, dans l'intervalle $[i..j]$ (les numéros des processeurs sont des entiers naturels de l'ensemble $I\!N$). D'après la fonction de projection qui numérote les nœuds par niveau, on sait que $d(j) = d(i) + n$, où n est le nombre de nœuds présents dans les processeurs entre $d(i)$ et $d(j)$. On sait aussi que $j = i + f$, où f est le nombre de fils des nœuds situés entre $d(i)$ et $d(j)$. Observons tout d'abord le cas particulier suivant dans lequel chaque sommet de l'intervalle $[d(i) \ldots d(j)]$ possède exactement un fils. Ces fils appartiennent à l'intervalle $[i \ldots j]$. Puisqu'on a associé à chaque élement de $[d(i) \ldots d(j)]$

un élement et un seul dans $[i \ldots j]$, il est manifeste que $d(j) - d(i) = i - j$. Si on a alors $d(j) - d(i) > j - i$ cela signifie qu'il existe un certain nombre de sommets entre $d(i)$ et $d(j)$ qui n'ont ni un fils droit ni un fils gauche et auquel cas ces sommets sont des feuilles. Par contre, si chaque sommet de l'intervalle $[d(i) \ldots d(j)]$ a au moins un fils, on aura alors forcément $d(j) - d(i) \leq j - i$.

La figure 5.5 illustre un exemple du cas où une collision a lieu quand les sommets présents dans 10 et 11 veulent communiquer avec leurs pères respectifs en 7 et 9. En effet, durant le routage R^+ (i.e., la procédure concentration), le nœud de $10 = 1010$ ayant pour destination $7 = 0111$ va se dépalcer, à la première itération, vers $11 = 1011$ et va écraser le nœud déja présent et qui ne se dépalce pas durant la même itération.

D'après ce qui vient d'être présenté ci-dessus, la condition d'absence de collision concerne le nombre de fils. Ainsi, si entre deux sommets $d(i)$ et $d(j)$, il n'existe pas de feuille (i.e., un sommet sommet qui n'ait pas d'enfants) alors on est assuré que des collisions ne peuvent se produire. Bien entendu, il peut arriver qu'une feuille dans $[d(i) \ldots d(j)]$ qui induit l'absence d'un ou deux sommets dans $[i \ldots j]$ soit compensé par un sommet ayant deux fils.

Mais d'une manière générale, on conclut que la condition d'absence de collision est toujours satisfite lors d'une opération ascendante fils→père (ou enfants→père) mise en œuvre par le routage R^+, si tout sommet de l'arbre appartenant à l'ensemble des destinations de l'ensemble de communication a au moins un fils.

La preuve est similaire dans le cas des opérations descendantes, père→fils et père→enfants. Rappelons que la procédure Distribute effectue l'opération inverse de Concentrate. Si d(i) et d(j) sont les deux fils respectifs des sommets se trouvant dans i et j. On sait que l'on a absence de mauvais conflits si la contrainte $|d(i) - d(j)| \geq |i - j|$ est vérifiée. Cette contrainte ne peut être assurée dans le cas d'arbre quelconques. Il suffit de reprendre la preuve précédente en intervertissant les rôles de $d(i)$ par i et de $d(j)$ par j.

On conclut donc qu'on peut réduire l'appel aux procédures de mouvement de données de [35]

Les appels aux procédures suivant la nature des opérations sont résumés dans les tableaux suivants :

TAB. 5.1 - *optimisation des appels aux procédures de mouvement de données dans le cas des opérations fils→père et père→fils.*

appel non optimisé	appel optimisé	
fils↔père	fils→père	père→fils
Rank	–	–
Concentrate	Concentrate	–
Distribute	–	Distribute

TAB. 5.2 - *optimisation des appels aux procédures de mouvement de données dans le cas des opérations enfants→père et père→enfants.*

appel non optimisé	appel optimisé	
enfants↔père	enfants→père	père→enfants
Rank	-	-
Concentrate	Concentrate	-
Generalize	-	Generalize

5.4.2.3 Le test de collision

Pour se rendre compte de la présence de collisions, on peut mettre en œuvre une procédure de test. Ce dernier va introduire bien entendu un coût supplémentaire qu'il faut réduire au minimum possible.

Notons que l'on peut choisir entre deux façons de mise en œuvre d'un test. Un test de décision nous permet de déterminer si l'on peut optimiser les appels aux procédures de mouvement de données, avant de lancer une opération. Un test de vérification, intégré à l'algorithme de routage, permet de surveiller si aucune collision ne se produit lors de l'exécution d'une opération d'une manière optimisée. Dans le cas contraire, on reprend l'exécution de l'opération sans recourir à l'optimisation.

Plus précisément. un test de vérification peut être implémenté de la manière suivante. Suivant l'opération à effectuer, on lance l'exécution du routage approprié R^+ ou R^-. Les enregistrements commence à se déplacer en empruntant les chemins correspondants selon les algorithmes déjà décrits. Rappelons que ces algorithmes ont comme propriété qu'à chaque étape, au plus un enregistrement utilise une arête donnée. Une collision a lieu si deux enregistrements se situent sur un même processeur au même instant comme cela a été illustré dans la figure 5.1.

Plus formellement, rappelons les notations, i_b qui signifie le b-ième bit de i, et $i^{(b)}$ qui est identique à i, sauf à la b-ième position où son bit est complémentaire de celui de i. Supposons qu'au début d'une itération b, on ait $d(i) \neq \infty$, $d(i^{(b)}) \neq \infty$, $d(i)_b = i_b$ et $d(i^{(b)})_b \neq i_b^{(b)}$. Durant cette itération, l'enregistrement de i va être écrasé par l'enregistrement provenant de $i^{(b)}$. En effet, l'enregistrement de i reste sur place durant cette itération puisque les deux bits $d(i)_b$ et i_b sont identiques. Par contre, l'enregistrement se trouvant dans le processeur voisin de i, soit $i^{(b)}$, va se déplacer vers i puisque les deux bits $d(i^{(b)})_b$ et $i_b^{(b)}$ sont différents.

Le test consiste donc à surveiller à chaque étape de l'exécution du routage R^+ ou R^-, qu'aucune collision ne se produit. Plus peécisément, que la condition suivante

$$d(i) \neq \infty,\ d(i^{(b)}) \neq \infty,\ d(i)_b = i_b \text{ et } d(i^{(b)})_b \neq i_b^{(b)}.$$

n'est pas vérifiée et ce à chaque étape b de l'algorithme R^+ ou R^-.

Si le test s'avère toujours négatif jusqu'à la fin de l'algorithme de routage, alors cela

veut dire qu'il n'y a pas eu de conflits et que tous les enregistrements sont arrivés à leurs bonnes destinations. L'opération de départ s'est donc convenablement exécutée en utilisant un appel optimisé. Par contre si le test s'avère positif, alors on reprend l'exécution de l'opération en utilisant l'appel non optimisé des procédures de mouvement de données.

Nous avons préferé le choix d'un test de vérification au lieu d'un test de décision, à cause de son coût plus réduit. En effet, il est difficile d'implémenter un test de décision qui vérifie que l'emsemble de communication introduit par l'opération à exécuter, est réducteur ou expansif, sans introduire un coût important étant donné que le problème de routage n'est pas toujours le même et n'est pas connu d'avance.

Pour s'en convaincre, examinons par exemple l'implémentation d'une procédure de test de décision qui traite un cas particulier simple dans lequel des nœuds d'un niveau k dans l'arbre veulent communiquer avec leurs enfants dans le niveau $k+1$. D'après l'analyse précédente, on sait que si tous les nœuds du niveau k ont au moins un fils, la condition d'absence de collision est toujours assurée. Le test peut donc être implémenté de la manière suivanate. Soit $C = \{ i_1 \rightarrow d(i_1), i_2 \rightarrow d(i_2), \ldots, i_p \rightarrow d(i_p) \}$ l'ensemble de communication à effectuer. Chaque sommet stocké dans un processeur i_ℓ dispose de la valeur 1 ou 0. Cette valeur est 1 s'il ne possède pas d'enfants, et si son sommet voisin de droite (s'il appartient à l'ensemble des sources S) et son sommet voisin de gauche (s'il appartient à S) possèdent des enfants. Il suffit d'effectuer une opération de réduction globale pour décider si l'on peut optimiser les appels aux procédures de mouvement de données. Il est important de noter cependant que pour qu'un nœud puisse communiquer avec son voisin de droite ou de gauche, il faudrait faire appel aux procédure de mouvement de données. En effet, ces sommets ne sont pas forcément placés sur des processeurs voisins. Le coût de ce test, même pour un cas particulier, devient donc très important.

5.4.2.4 La migration

Comme cela a déjà été mentionné dans la section 5.4.2, le plongement irrégulier ne permet pas à un sommet de l'arbre de posséder un numéro fixe et définitif. Ainsi, si des sommets sont ajoutés ou supprimés dans l'arbre, un réaménagement de cet arbre s'impose. Les adresses des nouveaux représentants doivent être calculées, et des sommets doivent alors être déplacés vers leurs nouveaux représentants. Ce réarrangement a pour but de préserver la relation d'ordre due à la fonction de projection basée sur la numérotation suivant l'ordonnancement par niveau. Notons cependant que s'il s'agit d'un arbre qui demeure complet (par exemple, s'il ne subit pas de suppression de sommets), tout réarrangement de sommets est inutile.

Pour supprimer des nœud et en rajouter d'autres dans leur arbre B&B, les auteurs de l'algorithme de B&B de [43, 2] ont décrit un algorithme de mise à jour de cet arbre. Plus précisément, l'arbre croît et décroît au cours de l'algorithme. Durant une étape de croissance, un sommet de l'arbre peut engendrer une feuille dont il faut calculer l'adresse de son représentant. Durant une étape de décroissance, des feuilles sont supprimées de l'arbre. L'algorithme de mise à jour de [43, 2] utilise des appels successifs aux procédures de mouvement de données suivant l'approche de [35]. Nous ne reprenons pas ici en détail la mise en œuvre du procesus de migration. En effet, il suffit de reprendre le même algorithme

de mise à jour de [43, 2] à quelques transformations près.

Il est important de noter que ce réarrangement (i.e., le processus de migration) introduit bien entendu un coût supplémentaire ce qui n'est pas souhaitable, d'autant plus que les conditions d'une utilisation optimisée des procédures de mouvement de données ne sont pas garanties.

Dans le chapitre suivant, nous allons voir qu'avec le plongement régulier, on n'aura pas besoin de s'assurer s'il est toujours possible d'optimiser l'utilisation des procédures de mouvement de données de [35]. En effet, dans le plongement régulier, une fois qu'un nœud est placé dans un processeur défini à l'avance, il ne peut être déplacé par la suite. Nous éliminons ainsi les inconvénients dus à l'irrégularité du placement des sommets de l'arbre. Malheureusement, nous éliminons dans le même temps l'avantage bénéfique de la charge optimale qu'apporte une telle irrégularité. Pour contourner ce problème, nous allons recourir à des algorithmes de plongement probabilistes.

5.4.2.5 Extensions

Rappelons que le sommet i d'un arbre de M sommets est projeté sur le processeur $i \ mod \ N$. Dans le cas où $M > N$, nous utiliserons les procédures de mouvement de données étendues comme cela a été indiqué dans le chapitre précédent. En effet, il suffit de décomposer le problème de mouvement de données en $\frac{M}{N}$ problèmes de mouvement de données injectifs. Les contraintes sur les arbres quant à la réduction du nombres d'étapes pour réaliser les opérations de parcours et consultations de l'arbre reste inchangées.

5.4.2.6 Expérimentations

Nous avons implémenté, sur la machine massivement parallèle MasPar MP-1 doté de 16.384 processeurs disposés en une grille bidimensionnelle 128 × 128, les opérations de mouvement de données de [35]. Nous avons considéré le cas d'un arbre binaire complet qui croît au fur et à mesure de la progression de l'algorithme. L'algorithme est mis en œuvre de la manière suivante. L'arbre croît niveau par niveau. Durant une étape de croissance, un sommet de l'arbre engendre ses deux sommets fils en invoquant l'opération père→fils. Durant une étape de décroissance, l'opération fils→père est invoquée pour permettre à un sommet père de recevoir une valeur de ses deux fils. Le plongement irrégulier de l'arbre binaire complet implique qu'un routage R^+ ou R^- s'exécute sans collisions comme cela a été montré dans la section 5.4.2.1. Cela signifie que les opérations de communication inter-sommets peuvent être implémentées d'une manière optimisée. La figure 5.6 montre le résulat experimental obtenu sur la MasPar quand un appel optimisé aux procédures de mouvement de données est utilisé (resp. n'est pas utilisé) sur la machine MasPar.

5.5 Conclusion

La propriété la plus intéressante du plongement irrégulier est que sa charge est optimale. Cette propriété est due à la fonction de projection qui établit une relation d'ordre sur les

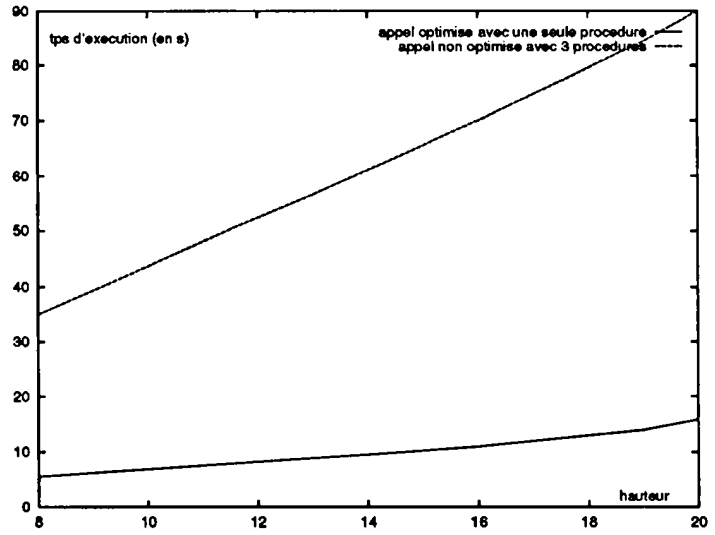

FIG. 5.6 - *L'effet de l'optimisation sur le temps d'exécution dans le cas d'un arbre binaire complet.*

sommets des arbres. Cela impose cependant une réorganisation de l'arbre (i.e. processus de migration) si sa forme ou sa taille varie. Nous avons montré que les opérations de communication entre les sommets de l'arbre plongé peuvent être implémentées d'une façon optimisée si l'arbre est complet. Dans le cas où l'arbre est dynamique et quelconque, l'absence de conflits lors d'une utilisation optimisée n'est pas garantie.

Dans le chapitre suivant, nous allons présenter la stratégie de plongement régulier qui permet de surmonter les difficultés du plogement irrégulier. En effet, cette stratégie qui plonge dynamiquement un arbre quelconque, permet d'une part de bénéficier des possibilités d'optimisation des opérations de mouvement de données quel que soit cet arbre. D'autre part, on n'aura pas à réorganiser l'arbre puisque les représentants affectés aux sommets de l'arbre sont fixes et définitifs. En d'autres termes, le principe de migration n'est pas autorisé. On reprend ainsi les mêmes hypothèses que le problème du plongement dynamique qui a été posé dans [17, 81, 82].

Partie 4

Chapitre 6

Plongement aléatoire d'arbres quelconques et dynamiques

6.1 Introduction

Dans le chapitre précédent, nous avons étudié une méthode de plongement dynamique d'un arbre quelconque de M sommets dans une grille multi-dimensionnelle de N processeurs. Comme nous l'avons vu, la charge obtenue par cette méthode est $\lceil \frac{M}{N} \rceil$, ce qui est le mieux que l'on puisse faire. Malheureusement, cette méthode de plongement est inefficace pour deux raisons. La première est qu'on ne peut pas bénéficier de l'optimisation des opérations de mouvement de données puisque les contraintes à satisfaire ne sont pas toujours vérifiées. La deuxième raison est que bien que la charge obtenue soit très bonne, la méthode mise en œuvre pour l'atteindre est chère dans la pratique. En effet, les représentants assignés aux sommets de l'arbre ne sont pas définitifs et doivent être recalculés si la forme de l'arbre varie. Par exemple, considérons le cas des arbres pouvant croître ou décroître au cours d'un algorithme. A chaque fois qu'une feuille est ajoutée à l'arbre durant une étape de croissance, elle n'est pas placée immédiatement, il faut d'abord calculer son représentant en fonction des sommets déja présents dans l'arbre et des autres feuilles à ajouter éventuellement. Si des feuilles sont supprimées dans l'arbre durant une étape de décroissance, il faut également recalculer les représentants des sommets restants de l'arbre et les déplacer vers leurs nouveaux représentants afin de respecter la numérotation dictée par la fonction de projection.

Dans ce qui suit, nous allons étudier un autre type de plongement. Afin de surmonter les problèmes de réarrangements des sommets de l'arbre évoqués ci-dessus, nous allons construire des plongements basés sur une fonction de projection *régulière*. Plus précisément, les représentants sont assignés aux sommets des arbres à plonger d'une manière fixe et définitive.

Notons que cette approche se situe bien dans le cadre du problème du plongement dynamique définie dans [17, 81, 122, 82]. Rappelons que ce problème a été posé dans le cadre du plongement dynamique d'arbres binaires quelconques dans un hypercube de la manière suivante [17, 81, 82]: on suppose pour simplifier que l'arbre binaire croît d'un sommet

à chaque étape, en commençant par la racine. A chaque fois qu'un sommet est ajouté à l'arbre, il l'est en tant que fils d'un sommet déjà existant et est immédiatement placé dans l'hypercube. La décision de savoir où placer le sommet se fait *localement* sans tenir compte des sommets qui pourront être ajoutés dans le futur. La décision du placement est implémentée dans le réseau d'une manière totalement distribuée sans recours à aucune information globale. Une fois qu'un sommet est ajouté, il ne peut plus être déplacé par la suite. Ce paradigme proposé par Bhatt et Cai [17] n'autorise pas la migration des sommets de l'arbre. La migration peut permettre potentiellement de donner une charge et une dilatation meilleures, mais sa mise en œuvre peut aussi être extrêmement chère dans la pratique [82, 89].

Nous commençons par décrire dans la section 6.2 les algorithmes de plongement dynamique d'arbres binaires dues à Leighton [81] et à Bhatt et Cai [17]. Nous définissons ensuite un modèle de plongement dynamique et nous présentons les objectifs que nous voulons atteindre. Dans la section 6.4, nous décrivons une méthode de plongement qui permet de construire des algorithmes probabilistes de plongement et nous montrons en particulier comment plonger en direct un graphe quelconque de M sommets dans une topologie quelconque de N sommets. Afin d'analyser le comportement aléatoire de ces algorithmes, nous allons utiliser des outils mathématiques issus de la théorie des chaînes de Markov et des résultats d'analyse numérique sur les itérations des matrices carrées. Cette analyse permettra de mettre en évidence le résultat décrit dans la section 6.5 et qui montre que l'approche de plongement permet de plonger un arbre quelconques G de M sommets dans un graphe hôte H de N sommet de telle sorte qu'un sommet de H ne représente qu'au plus $O(M/N + \varepsilon)$ sommets de l'arbre, où ε dépend de la fonction de projection utilisée pour effectuer le plongement.

Dans le cas des réseaux synchrones, et en particulier, l'hypercube et la grille, les procédures de mouvement de données peuvent toujours être utilisées afin de répondre au problème du routage inter-sommets dans les arbres plongés comme cela a été montré dans le chapitre précédent. Les contraintes à vérifier pour une utilisation optimisée restent toujours valables. Cependant, il est important de noter que les résultats de ce chapitre s'étendent au problème de placement d'un graphe de processus dans une topologie quelconque.

6.2 Ce qui a été fait

Bhatt et Cai qui ont étudié initialement le problème du plongement dynamique d'arbres binaires quelconques dans un hypercube décrivent dans [17, 122] un algorithme qui plonge dynamiquement un arbre binaire de M sommets dans un hypercube de N sommets ayant, avec une forte probabilité, une dilatation égale à $O(log\, log\, N)$ et une charge en $O(\frac{M}{N}+1)$. La congestion du plongement n'est pas determinée mais elle est probablement de l'ordre de $log^{\alpha}N$ pour une constante α selon [81]. Bhatt et Cai ont employé la théorie de la marche aléatoire pour construire leur plongement. Ils ont obtenu ainsi leur résultat probabiliste en utilisant une analyse de marche aléatoire. Le principe est le suivant. Si un sommet x de l'arbre est placé dans un sommet v d'un hypercube et si x a un fils y, alors un parcours aléatoire d'une longueur donnée est effectué au traves de l'hypercube pour placer y, en empruntant à chaque étape une arêtre aléatoire pour progresser. L'idée sous-

jacente au plongement de Bhatt et Cai est la suivante. La stratégie de plongement d'un nouveau sommet est divisée en deux étapes. Dans la première étape, un nouveau sommet à placer effectue une marche aléatoire de longueur $O(log\,log\,N)$ et arrive dans un sous-hypercube de taille $O(logN)$. A l'intérieur de ce sous-hypercube, le sommet est assigné, d'une manière déterministe et uniforme, à un des $O(log\,N)$ processeurs le moins chargé de ce sous-hypercube. Etant donné que chaque sous-hypercube de taille $O(log\,N)$ possède un diamètre de $O(log\,log\,N)$, la dilatation totale du plongement est donc $O(log\,log\,N)$. Dans leur article [17], Bhat et Cai se sont contentés d'analyser seulement la première étape de leur algorithme mais ils n'ont pas détaillé davantage la deuxième étape. Plus précisément, Bhatt and Cai ont calculé la plus petite longueur de la marche aléatoire à devoir effectuer pour assurer que leur algorithme, composé des deux étapes ci-dessus, plonge dynaniquement un arbre quelconque avec une charge en $O(\frac{M}{N}+1)$.

Dans [81, 82], Leighton, Newman, Ranade et Schwabe ont amélioré le résultat de [17] et ils ont décrit un algorithme de plongement plus simple qui permet de plonger un arbre binaire quelconque de M sommets dans un hypercube de N sommets avec une dilatation 1 et une charge en $O(\frac{M}{N}+logN)$ avec une forte probabilité. Pour obtenir ce résultat, Leighton introduit le hasard dans son algorithme en utilisant une technique qu'il nomme *principe du bit oscillant.*

Leighton utilise cette technique de la manière suivante. A chaque sommet v de l'arbre binaire est associé de façon aléatoire un *bit d'oscillation* $b(v)$ qui a la même probabilité d'être égale à 1 ou 0. Supposons que x soit le sommet plongé sur un certain processeur i. Soit $i^{t(x)}$ le processeur voisin de i dans la $t(x)$-ième dimension de l'hypercube. Les fils de x, s'ils existent, sont plongés de la façon suivante. Si le bit $b(x)=0$, alors le fils gauche de x (s'il existe) est plongé avec x sur le même processeur i et le fils droit (s'il existe) sur $i^{t(x)}$. Si $b(x)=1$, alors les fils de x (s'ils existent) sont plongés de façon inverse. Les valeurs de $t(x)$ sont déterminées de la façon suivante. A chaque sommet x de l'arbre binaire est associé un numéro de trace $t(x)$ appartenant à l'intervalle $[1, logN]$ et congru à son niveau dans l'arbre modulo $logN$. Au départ, la trace de la racine est 1, et la racine est plongée sur le sommet 0 de l'hypercube. Si y est un fils de x dans l'arbre, alors $t(y)=t(x)+1$ (sauf si $t(x)=logN$, et dans ce cas $t(y)=1$). Ce plongement est illustré sur la figure 6.1 appliqué à un arbre binaire complet et pour $N=8$. Leighton montre que le comportement aléatoire du bit oscillant, utilisé dans son algorithme de plongement, permet de donner les charges décrites ci-dessus[1].

Leighton donne dans [81] un deuxième algorithme de plongement qui plonge un arbre binaire quelconque de M sommets dans un hypercube de N sommets avec une dilatation en $O(1)$ et une charge en $O(\frac{M}{N}+1)$. Il améliore ainsi d'un facteur $logN$, pour $M=\Theta(N)$, la borne sur la charge du premier algorithme. Notons qu'en améliorant la borne sur la charge, la dilatation augmente d'un facteur constant.

Ce deuxième algorithme de plongement est mis en oeuvre en combinant un plongement de l'arbre binaire dans un réseau appelé *hypercube de cliques*, avec un plongement de ce dernier dans l'hypercube. Un hypercube de cliques P_r de dimension r possède 2^r sommets

1. Leighton introduit d'ailleurs le hasard dans plusieurs endroits dans son livre [81]. On retrouve ainsi la description de plusieurs algorithmes dont le comportement est aléatoire, dans des contextes différents du plongement (voire par exemple, les chapitres 7, 16, 21, 24, 25, 26).

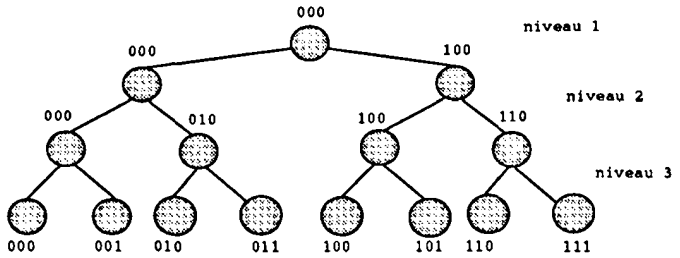

FIG. 6.1 - *Le premier algorithme de plongement décrit par Leighton et al. appliqué à un arbre binaire complet et pour $N = 8$. Les étiquettes sur les sommets de l'arbre désignent les processeurs de l'hypercube sur lesquels ils sont projetés. La i-ème feuille est projetée sur le i-ème sommet de l'hypercube, et chaque sommet interne est projeté sur le même sommet que la feuille la plus à gauche de son arbre, $0 \leq i \leq N-1$. Les valeurs des bits d'oscillation ne font aucune différence dans ce cas d'un arbre binaire complet. Mais pour un arbre binaire quelconque, Leighton montre que le comportement aléatoire permet de plonger l'arbre avec une charge en $O(M/N + logN)$ avec une forte probabilité.*

et se construit à partir d'un hypercube de dimension r en ajoutant une arrête entre les couples de sommets de l'hypercube qui diffèrent sur les $\lfloor logr \rfloor$ derniers bits. La taille des cliques est $2^{\lfloor loglogN \rfloor} = \Theta(logN)$. Un hypercube de cliques de dimension 4 est illustré sur la figure 6.2.

Leighton utilise encore l'idée du bit oscillant pour plonger un arbre binaire quelconque dans l'hypercube de cliques de la manière suivante. A l'aide du bit oscillant, la clique dans laquelle un sommet de l'arbre est plongé est déterminée de la même manière que dans le premier plongement, où chaque clique joue le rôle d'un sommet de l'hypercube. On projette ensuite le sommet de l'arbre à l'intérieur de la clique sur le sommet de cette clique ayant la plus petite charge. Ceci a pour effet de répartir la charge des sommets de chaque clique. Leighton montre ainsi que cet algorithme plonge un arbre binaire quelconque de M sommets dans un hypercube de cliques $P_{log\,N}$ avec une dilatation $O(1)$, une charge en $O(\frac{M}{N}+1)$ et une congestion en $O(\frac{M}{N}+1)$.

Leighton montre ensuite en utilisant la théorie des codes 1-correcteurs [81] qu'un hypercube de cliques de N sommets se plonge dans un hypercube de N sommets avec une charge, une dilatation et une congestion constantes. En combinant le plongement de l'arbre dans un hypercube de cliques et le plongement de ce dernier dans un hypercube, il montre que le plongement d'un arbre binaire quelconque de M sommets dans un hypercube de N sommets possède une charge en $O(\frac{M}{N}+1)$, une congestion en $O(\frac{M}{N}+1)$ et une dilatation en $O(1)$.

6.3 Définition du modèle de plongement dynamique

Un algorithme de plongement possède en entrée les trois données suivantes : le graphe logique G que l'on plonge, la fonction de projection, et le graphe hôte H dans lequel on effectue le plongement. Le rôle de l'algorithme de plongement est de placer les sommets

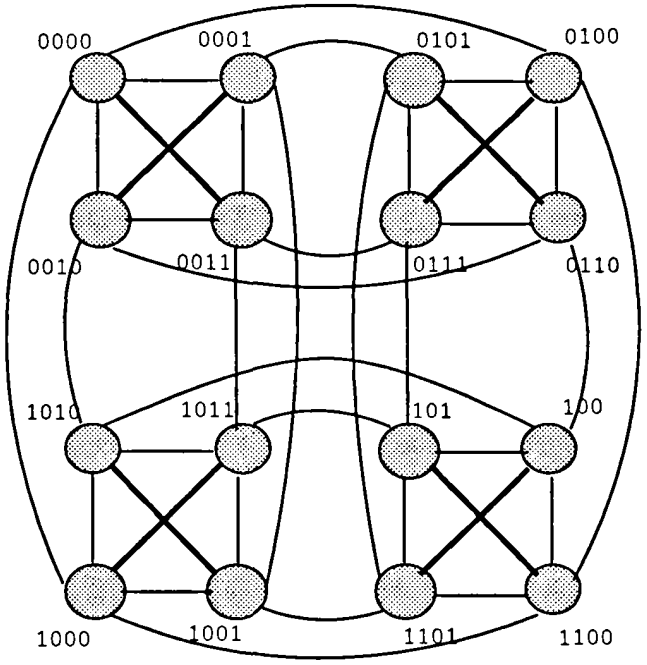

FIG. 6.2 - *Un hypercube de cliques P_r de dimension $r = 4$ et possédant 2^4 sommets. Les arêtes ajoutés à l'hypercube de 16 sommets pour former l'hypercube de cliques de 16 sommets sont représentés en gras. Deux sommets sont voisins s'ils diffèrent d'un seul bit, ou s'ils diffèrent sur leur $\lfloor logr \rfloor$ derniers bits. La taille des cliques ici est 4.*

de G sur les sommets du graphe H en utilisant la fonction de projection. Dans notre cas, le graphe logique est un arbre quelconque que l'on doit plonger sans connaître a priori sa forme ou sa taille. Il existe bien sur de nombreux choix pour la fonction de projection ϕ. L'un des plus naturels est de considérer, comme cela a été fait dans le chapitre précédent, la numérotation des sommets de l'arbre en largeur d'abord et d'affecter le sommet de l'arbre de numéro i au sommet i du graphe hôte (modulo le nombre de sommets présents dans ce graphe).

Considérons le même problème de plongement dynamique définie dans [17, 81, 82]. Rappelons que ce problème a été posé dans le cadre du plongement dynamique d'arbres binaires quelconque dans un hypercube alors que dans notre cas, nous voulons plonger des arbres d'arités quelconques. Le problème de plongement dynamique d'un arbre quelconque d'arité d est défini alors de la manière suivante : on suppose pour simplifier que l'arbre à plonger croît d'un sommet à chaque étape, en commençant par la racine. A chaque instant, un sommet quelconque de l'arbre alloué à un certain sommet u et ne possédant pas d fils peut générer un nouveau sommet. Les nouveaux sommets générés doivent être projetés sur des sommets du graphe hôte en respectant les contraintes suivantes :

- sans tenir compte des sommets qui pourront être ajoutés dans le futur,

- sans accéder à aucune information globale. En conséquence, la décision de placement des sommets doit être implémentée d'une manière distribuée dans le réseau,

- Une fois qu'un sommet de l'arbre est placé dans un sommet particulier de G, il ne peut plus être déplacé. En d'autres termes, le processus de migration n'est pas autorisé,

- la fonction de projection doit être facile à calculer.

Il est important de noter les deux points suivants nécessaires pour déterminer la fonction de projection ϕ. Le premier est qu'il n'existe pas de mécanisme de contrôle centralisé (i.e., une structure de données globale qui contient l'état de charge des processeurs et qui permet de centraliser le processus de placement des sommets de l'arbre). Le deuxième point est que la fonction de projection ϕ doit être facile à calculer. En fait, la fonction de projection qui distribue les sommets de l'arbre peut être calculée conjointement par tous les processeurs d'une façon totalement distibuée comme cela a été proposé dans le chapitre précédent. Cette stratégie est cependant inefficace à cause du temps dépensé pour le calcul de la fonction de projection ce qui induit un surcoût additionnel au plongement. Il est préférable donc que la fonction de projection puisse être calculée facilement et que la décision de placement d'un sommet puisse s'effectuer d'une manière locale et presque instantanée. L'algorithme de plongement doit être mis en œuvre de la manière suivante. Quand un sommet u placé dans un processeur p_i génère un nouveau fils v, u choisit un processeur $p_j = \phi(v)$ et envoie v dans ce processeur p_j. Cette décision est effectuée localement dans p_i sans consulter les autres processeurs. En conséquence, $\phi(v)$ dépend uniquement du processeur p_i qui l'a caclulé.

De plus, comme cela a été montré dans le chapitre précedent, quand le graphe hôte représente une grille ou un hypercbe, il faut aussi choisir soigneusement la fonction de projection de façon à satisfaire toutes les contraintes afin de s'assurer qu'il est toujours possible de bénéficier de l'utilisation optimisée des opérations de mouvement de données.

D'une manière générale, les connexions entre les couples de sommets du graphe hôte, qui représente le réseau distribué sur lequel l'algorithme opère, varient selon le réseau représenté. Ainsi dans une grille de processeurs, chaque processeur ne dispose que d'un nombre constant de connexions avec d'autres processeurs (i.e., un réseau de degré borné). Dans un hypercube, chaque processeur est connecté à $log\, N$ autres processeurs. Dans un réseau de stations de travail, chaque station est connecté à???.

Cependant, il est important de noter que la stratégie de plongement que nous allons décrire ne dépend pas du graphe hôte. Plus précisément, nous ne considérons aucun réseau particulier. Le graphe hôte peut représenter par exemple un réseau de stations de travail ou une grille de processeurs. En fait, la structure du réseau n'affecte pas l'analyse du plongement que nous allons présenter. Tout au long de ce chapitre, nous allons nous intéresser particulierement à résoudre le probème de l'équilibrage de charge et qui est d'exiger que la fonction de projection plonge d'une manière équitable les sommets de l'arbre sur les sommets du graphe hôte. .

6.4 Description de l'algorithme de plongement aléatoire

Considérons les notations suivantes. Soient P_j un sommet du graphe hôte (i.e., un processeur ou une station) de numéro j et d un entier donné. On définit, pour tout sommet P_j, l'ensemble

$$\mathcal{V}(P_j) = \{P_{n(j,1)}, P_{n(j,2)}, ..., P_{n(j,d)}\}.$$

Cet ensemble définit plus précisément *un voisinage* logique pour tout sommet P_j. $n(j,\delta)$ désigne le voisin logique numéro δ de P_j. Les termes *voisin logique* signifie qu'un élement de l'ensemble $\mathcal{V}(P_j)$ n'est pas nécessairement un voisin physique de P_j dans le réseau distribué représenté par le graphe hôte.

L'algorithme de plongement que nous allons décrire est aléatoire et opère de la façon suivante. Supposons qu'un sommet u de l'arbre est placé dans le sommet P_j. Les fils du u sont placés aléatoirement sur les différents sommets de l'ensemble $\mathcal{V}(P_j)$. Plus précisément, supposons par exemple que le sommet u génère un nouveau fils v. Le sommet u choisi aléatoirement par l'intermediaire de la fonction de projection ϕ un élément P_k de $\mathcal{V}(P_j)$ et envoie son fils v sur ce sommet $P_k = \phi(v)$.

Nous appelons les fonctions de projection utilisées selon cette stratégie de plongement, des fonctions de projection *régulières* étant données que les valeurs de ϕ sont calculées d'une manière fixe et définitive (i.e., une fois qu'un sommet u de l'arbre est placé dans le sommet $\phi(u)$ de G, il ne peut plus être déplacé).

L'aspect aléatoire est essentiel pour le succés de l'algorithme comme cela va être montré dans son analyse. En effet, l'introduction du hasard au niveau de la fonction de projection régulière nous permet d'éviter des conséquences désastreuses au niveau de la charge des processeurs. Cet aspect aléatoire peut être implémenté en s'inspirant de l'idée du *principe du bit oscillation* décrit par Leighton dans [81]. Notons qu'au départ, la racine de l'arbre à plonger est placée sur un sommet initial choisi d'une manière aléatoire ou déterministe.

La décision distribuée du placement est mis en œuvre en s'inspirant de l'idée du principe du bit oscillant de la manière suivante. Soit T l'arbre d-aire quelconque. A chaque sommet v de l'arbre est associé une valeur aléatoire $a(v) \in [0,1[$. Soit $b(v) = a(v) \times d$. $b(v) \in [0,d[$ a la même probabilité d'être dans $[0,1[$, $[1,2[$, ... ou $[d-1,d[$. Les fils du sommet v sont placés selon la valeur de $b(v)$ de la façon suivante. Supposons que le sommet v est plongé sur un certain processeur P_j, et que ses fils doivent être plongés dans les sommets éléments de $\mathcal{V}(P_j)$. Si $b(v) \in [0,1[$, alors le fils le plus à gauche de v (s'ils existe) sera plongé dans le sommet $P_{n(j,1)}$, le fils suivant (s'il existe) est placé dans $P_{n(j,2)}$, ..., et le d-ième fils (s'il existe) est placé dans $P_{n(j,d)}$. Si $b(v) \in [1,2[$, alors les fils de v seront plongés de la même manière dans les sommets $P_{n(j,2)}, P_{n(j,3)}, \ldots, P_{n(j,d)}, P_{n(j,1)}$. On peut généraliser ainsi. Si $b(v) \in [p,q[$, alors le fils le plus à gauche de v (s'il existe) est plongé dans le sommet $P_{n(j,q)}$, le fils suivant (s'il existe) est placé dans $P_{n(j,q+1)}$, le $(d-q+1)$-ième fils (s'il existe) est placé dans $P_{n(j,d)}$ et le d-ième fils (s'il existe) est placé dans $P_{n(j,q-1)}$. Les fils de v sont ainsi plongés suivant le comportement aléatoire de $b(v)$.

A titre d'exemple, considérons l'algorithme de plongement suivant d'un arbre quelconque d'arité d dans N sommets. On définit le voisinage logique d'un sommet P_j en posant $n(j,\delta) = (jd+\delta) \bmod N$ pour $0 \leq j \leq N-1$ et $1 \leq \delta \leq d$. Les positions des sommets

de l'arbre sont déterminées de la manière suivante. Si un sommet u de l'arbre est placé dans un sommet P_j, alors à chaque fils de ce sommet est affecté un sommet aléatoire appartenant à l'intervalle $\mathcal{V}(P_j) = [P_{(jd+1) \bmod N}, P_{(jd+2) \bmod N}, \ldots, P_{(jd+d) \bmod N}]$ suivant la valeur de $b(u)$.

Si on applique cet algorithme de plongement à un arbre binaire, on a $d = 2$ et $\mathcal{V}(P_j) = [P_{(2j+1) \bmod N}, P_{(2j+2) \bmod N}]$ pour un sommet quelconque P_j. La décision de placement probabiliste est implementée de la manière suivante. A chaque nœud v est affecté une valeur aléatoire $a(v) \in [0, 1[$. Soit $b(v) = a(v) \times 2$ (étant donné qu'on considère un arbre binaire), $b(v) \in [0, 2[$ et a la même probabilité d'être dans $[0, 1[$ ou $[1, 2[$. Les enfants de v sont plongés selon cette valeur. Supposons que v est plongé dans un sommet P_j. Si $b(v) \in [0, 1[$ alors le fils gauche de v (s'il existe) est plongé dans le sommet $2j + 1$ et le fils droit (s'il existe) dans le sommet $2j + 2$. Si $b(v) \in [1, 2[$ alors les fils de v sont plongés de façon inverse. Notons que dans le cas des arbres binaires, on peut se contenter des valeurs alátoires $a(v)$ comme dans [81], mais on utilise $b(v)$ comme nous l'avons proposé dans le cas général).

6.5 Le résultat principal

Nous nous intéressons ici à determiner la charge de l'algorithme de plongement, i.e. le nombre maximum de nœuds de l'arbre qui sont projetés sur un même sommet du réseau.

Afin d'analyser le comportement aléatoire de l'algorithme de plongement, nous allons utiliser des outils mathématiques issus de la théorie des chaînes de Markov et des résultats d'analyse numérique sur les itérations des matrices carrées.

En effet, la distribution des nœuds de l'arbre peut être vue comme étant une distribution de probabilité, et le placement des sommets générés devient mathématiquement identique à l'évolution d'une chaîne de Markov.

En conséquence, nous allons utiliser une technique de preuve basée sur une analyse d'un processus de Markov en modélisant l'algorithme probabiliste de plongement par sa matrice stochastique. Le résultat principal que nous allons obtenir est le suivant :

Result *étant donné un algorithme de plongement d'un arbre de processus quelconque dans une topologie arbitraire, le nombre de nœuds qui sont placés sur un même sommet du réseau ne dépasse pas $O(\frac{M}{N} + \varepsilon)$, où M est le nombre de nœuds, N le nombre de sommets du réseau et ε, qui sera donné par la suite, est une caracteristique de la fonction de projection choisie.*

Nous commençons l'analyse par une digression sur la théorie des chaînes de Markov et sur l'analyse numérique des matrices stochastiques. Dans la section 6.6, nous allons rappeler quelques résultats relatifs au calcul matriciel [118, 22, 90] et déduire quelques remarques dont on aura besoin par la suite. Nous représentons ensuite dans la section 6.8 brièvement le concept des chaînes de Markov [76, 118, 10, 29, 22, 90] (voir aussi [100, 103, 49]).

6.6 Matrices stochastiques

Les matrices stochastiques[2] constituent une classe importante des matrices carrées ayant des propriétés particulières. Nous allons rappeler ci-dessous certaines de ces propriétés que l'on peut trouver aussi dans [118, 22, 90].

Soit $A = (a_{ij})$ une matrice réelle $n \times n$. La matrice A est dite *non négative* si tous ses éléments a_{ij} sont réels et non négatifs (i.e., positifs ou nuls). Une matrice est dite *positive* si tous ses éléments sont réels et positifs.

Une matrice non négative est dite *stochastique*, si toutes ses sommes en lignes ou en colonnes sont égales à 1. Elle est doublement stochastique si toutes ses sommes en lignes et en colonnes sont égales à 1.

6.6.1 Valeurs propres et vecteurs propres

Rappelons les définitions des valeurs propres et des vecteurs propres à droite et à gauche d'une matrice carrée.

Un vecteur colonne non nul q est un vecteur propre, à droite, d'une matrice carrée A, s'il existe un scalaire λ tels que

$$Aq = \lambda q$$

λ est alors une valeur propre de A qui peut être un réel ou un complexe.

Le vecteur ligne non nul p est un vecteur propre, à gauche[3], de A, associé à la valeur propre λ, si on a

$$pA = \lambda p$$

Les valeurs propres λ de la matrice A de taille $n \times n$ sont les racines de l'équation caractéristique en λ de A suivante (*det* pour déterminant),

$$det(A - \lambda I) = 0$$

Cette équation est un polynome de degré n, et ses racines λ peuvent être réelles, complexes ou multiples.

Le rayon spectral d'une matrice carrée représente la plus grande valeur propre en valeur absolue de cette matrice, et il existe au moins une valeur propre pour laquelle il s'agit d'une égalité. On sait que si une matrice carrée A $n \times n$ est non négative et si les sommes en lignes (ou en colonnes) sont égales à une constante k, alors son rayon spectral $\rho(A)$ est égal à k. Il existe alors un vecteur propre à droite et un vecteur propre à gauche correspondant à cette valeur propre $\rho(A)$ dont toutes les composantes sont non négatives.

Il en résulte donc que pour une matrice stochastique[4] A, 1 est toujours une valeur propre,

2. Une suite finie d'expérience dont chacune a un nombre fini de résultats possibles avec des probabilités données est appelée un processus stochastique fini.

3. Notons qu'en parlant de vecteur à droite, on parle de vecteur colonne, et en parlant de vecteur gauche, on parle de vecteur ligne.

4. Pour dégager toute ambigüité et pour simplifier l'écriture par la suite, on appelera ici une matrice stochastique la matrice dont toutes les sommes en lignes sont égales à 1. Notons par ailleurs qu'une matrice et sa transposé ont toujours les mêmes valeurs propres.

et il existe un vecteur propre à droite et un vecteur propre à gauche correspondant à cette valeur propre $\lambda = 1$

De plus, toutes les valeurs propres de A sont dans le disque-unité du plan complexe. Plus précisément, si λ est une valeur propre de A, alors $|\lambda| \leq 1$. En d'autres termes, toute valeur propre complexe de A a un module inférieur ou égale à 1, et une valeur propre de A de module 1, distincte de 1, est nécessairement une racine $q$ième de l'unité, $1 \leq q \leq n$. Cette remarque nous sera utile pour le calcul de la puissance $k$ième d'une matrice stochastique. Des démonstrations de ces caractéristiques des valeurs propres d'une matrice stochastique sont données dans [29]. Une autre démonstration du fait que les valeurs propres de A sont dans le disque-unité du plan complexe, peut être déduite du théorème de Gerschgorin[5] donné dans [22].

Notons de plus que comme toutes les sommes en lignes (resp. en colonnes) de la matrice stochastique sont égales à 1, alors le vecteur propre à droite (resp. à gauche) correspondant à la valeur propre $\lambda = 1$ a toutes ses composantes *égales* et *positives*. Ses composantes sont positives puisque un vecteur propre ne peut être le vecteur nul par contre les valeurs propres peuvent être nulles.

En effet, pour une matrice stochastique dont les sommes en lignes sont égales à 1, considérons le vecteur colonne

$$q = \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{pmatrix}$$

Ce vecteur est un vecteur propre[6] à droite associé à la valeur propre $\lambda = 1$ puisque

$$q = A\,q$$

Notons que le vecteur propre à gauche p associé à $\lambda = 1$ dans ce cas (i.e., $p = pA$), n'est pas forcément égal au transposé de q, et ne possède pas nécessairement des composantes égales.

De même, quand toutes les sommes en colonnes de la matrice A sont égales à 1, alors le vecteur suivant, transposé de q, soit

$$q^T = \begin{pmatrix} 1, 1, ..., 1 \end{pmatrix}$$

est un vecteur propre à gauche associé à $\lambda = 1$ puisque

$$q^T = q^T A$$

5. Chaque ligne d'une matrice carrée engendre un disque de Gerschgorin, limité par un cercle dont le centre est l'élément diagonal de la ligne et le rayon la somme des valeurs absolues de tous les autres éléments de cette ligne. D'après le théorème de Gerschgorin, toute valeur propre d'une matrice réelle ou complexe se trouve à l'intérieur de l'un de ces disques de Gerschgorin. Ce théorème est utilisé pour trouver une valeur approchée des valeurs propres d'une matrice. Le théorème de Gerschgorin peut être appliqué à la transposée d'une matrice pour déterminer un second ensemble d'approximation des valeurs propres de la matrice, puisque les valeurs propres se conservent par transposition.

6. Si v est un vecteur propre d'une matrice, alors kv l'est aussi quelque soit la constante k non nulle, et v et kv correspondent à la même valeur propre.

On déduit donc que dans le cas où la matrice est doublement stochastique, les deux vecteurs propres à gauche et à droite ont toutes leurs composantes positives et égales. Ces deux vecteurs sont bien entendu q et son transposé q^T. Cette remarque nous sera utile par la suite pour trouver facilement le vecteur propre associé à la valeur propre 1 quand la matrice est doublement stochastique.

Notons que si la détermination des valeurs propres ne présente aucune difficulté en théorie, ce n'est pas le cas en pratique. Les valeurs propres λ d'une matrice A $n \times n$ sont les racines de l'équation caractéristique en λ de A suivante, $det(A - \lambda I) = 0$. La recherche des racines de ce polynome caractéristique de degré n qui représentent les valeurs propres est un problème algébrique très difficile en général. Pour calculer donc les valeurs et les vecteurs propres, on doit recourir à des méthodes numériques (des méhodes numériques comme la méthode de la puissance sont données dans [22]) et à utiliser des logiciels numériques (comme le logiciel Matlab ou Maple), surtout quand les matrices sont de grandes tailles.

Dans ce qui suit, nous allons utiliser les renseignements qui ont été cités ci-dessous pour trouver facilement le vecteur propre associé à la valeur propre $\lambda = 1$.

Nous résumons les propriétés d'une matrice stochastique dans le théorème suivant :

Theorème 6 *Soit A une matrice stochastique, on a,*
(a) A possède 1 *comme valeur propre*
(b) toutes les valeurs propres de A sont majorées par 1
(c) une valeur propre de A de module 1, *distincte de* 1 *est nécessairement une racine qième de l'unité pour* $1 \leq q \leq n$
(d) si de plus, A est doublement stochastique, alors toutes les composantes du vecteur propre p associé à la valeur propre 1 *sont égales et positives.*

6.6.2 Calcul de la limite d'une suite de matrices stochastiques

Une matrice stochastique est dite *ergodigue* si la seule valeur propre dont la valeur absolue soit égale à 1 est 1 lui même. Cette valeur propre $\lambda = 1$ peut être de multiplicité k. Une matrice stochastique ergodique admet une limite $\widetilde{A} = \lim_{\ell \to \infty} A^{\ell}$.

On dit aussi qu'une matrice stochastique est *régulière* si l'une des ses puissances est une matrice positive (i.e., tous les éléments sont réels et > 0). Une matrice stochastique *régulière* est une matrice ergodique avec la valeur propre $\lambda = 1$ de multiplicité 1. Une matrice régulière possède donc une matrice limite $\widetilde{A}$.

Les matrices positives sont régulières. Une condition suffisante qui assure donc l'existence d'une limite pour une matrice stochastique A est qu'elle soit positive (i.e., il existe un réel $\epsilon > 0$ qui minore tous les coefficients de A).

Dans le cas d'une matrice régulière, la matrice limite $\widetilde{A}$ a une forme simple. Toutes les sommes en lignes de $\widetilde{A}$ sont égales à un et toutes les lignes de $\widetilde{A}$ sont identiques. Chacune des lignes est constituée du seul vecteur propre à gauche correspondant à la valeur propre $\lambda = 1$ dont la somme des composantes soit égale à 1. Des démonstrations sont données dans [29, 22, 90].

La forme de la matrice est plus complexe pour une matrice ergodique non régulière. Soit k la multiplicité de $\lambda = 1$. Si on place les k vecteurs propres à droite linéairement indépendants correspondants à cette valeur propre dans les k premières colonnes de la matrice modale[7] M, alors $\widetilde{A} = MDM^{-1}$, où D est une matrice diagonale dont les k premièrs éléments diagonaux sont égaux à un et les autres à zéro [22]. Des détails sur les matrices modales et le calcul de la matrice $\widetilde{A}$ quand A est ergodique mais non régulière peuvent être trouvés dans [22].

Nous résumons ces notions dans les définitions suivantes:

Définition 7 *Une matrice stochastique A est dite ergodique si la seule valeur propre dont le module soit égale à 1 est 1 lui même. La matrice limite $\lim_{t\to\infty} A^t$ existe alors et elle est stochastique.*

Définition 8 *Une matrice stochastique A est dite régulière s'il existe $k > 0$ tels que tous les éléments de A^k, la kième puissance de A, sont positives. De plus A est ergodique et sa valeur propre 1 est simple. La matrice A admet donc une matrice limite $L = \lim_{t\to\infty} A^t$.*

Nous résumons aussi les notions de convergence de séquences de matrices régulières dont on aura besoin par la suite dans les définitions suivantes:

Theorème 7 *Si A est une matrice régulière, alors $\lim_{t\to\infty} A^t = \widetilde{A}$ existe et elle est une matrice stochastique dont toutes les lignes sont identiques au même vecteur $\widetilde{p}$. Ce vecteur $\widetilde{p}$ est le vecteur propre à gauche correspondant à la valeur propre 1, $\widetilde{p} = \widetilde{p}A$, dont la somme des composantes est égale à 1.*

Il est important de noter que de plus, si A est doublement stochastique, alors la matrice limite $\widetilde{A}$ possède une forme particulière; tous ses éléments sont égaux à $1/n$. En effet, considérons le vecteur ligne dont tous les n éléments sont égaux à 1. Ce vecteur peut être normalisé en divisant chaque élément par n. On obtient alors un vecteur unique, $\widetilde{p}$, dont tous les éléments sont égaux à $1/n$. Etant donné que toutes les sommes des lignes de A sont égales à 1 alors on a $\widetilde{p} = \widetilde{p}A$. $\widetilde{p}$ est donc le vecteur propre à gauche correspondant à la valeur propre 1 dont la somme des composantes est égale à 1. D'après le théorème 7, on construit la matrice limite $\widetilde{A}$ dans sa forme uniforme.

Theorème 8 *Si A est régulière et doublement stochastique, alors $\widetilde{A}$ est une matrice $n \times n$ dont tous les éléments sont égaux à $1/n$.*

6.7 Distribution de probabilité

Un vecteur est un vecteur de probabilités si toutes ses composantes sont positives ou nulles et si la sommes de ses composantes est égale à un[8].

7. Une matrice modale M pour A est une matrice de même ordre que A ayant comme colonnes tous les vecteurs d'une base canonique de A. Une base canonique est un ensemble de vecteurs linéairement indépendants.

8. On peut dire d'ailleurs qu'une matrice est stochastique si toutes ses lignes (ou colonnes) sont des vecteurs de probabilités

Pour obtenir un vecteur de probabilité à partir d'un vecteur v quelconque, sachant que ses composantes non nulles sont positives, il suffit de diviser chaque composante de ce vecteur par la somme de ses composantes [90]. Le vecteur ainsi obtenu est un vecteur de probabilité puisque la somme de ses composantes est égales à 1. Ce vecteur de probabilité, soit αv, est en fait un multiple scalaire de v, il est donc unique.

A titre d'exemple, considérons $v = (1, 1, 1, 1)$, le vecteur propre à gauche d'une matrice doublement stochastique 4×4, associé à la valeur propre 1. v a toutes ses composantes positives. Il existe un et un seul vecteur multiple scalaire de v, $\frac{1}{4}v = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ qui soit un vecteur de probabilités.

6.8 Chaînes de Markov finies

Une chaîne de Markov à n états consiste en un ensemble d'objets et un ensemble fini de n états différents, n étant un entier positif, telle que

1. A un instant donné, chaque objet se trouve dans l'un des n états,
2. La probabilité qu'un objet passe d'un état à un autre, ou reste dans le même état, ne dépend que de ces états de départ et d'arrivée[9].

La probabilité qu'un objet passe à un état j à partir d'un état i, notée P_{ij}, est appelée *probabilité de transition d'ordre* 1. Les valeurs P_{ij}, $0 \leq i, j < n$, vérifient

$$P_{ij} \geq 0 \; et \sum_j P_{ij} = 1$$

Ces probabilités de transitions conditionnelles p_{ij} peuvent être rangées sous la forme d'une matrice carrée A, appelée matrice de transition ou matrice de Markov, dont le nombre de lignes est égale au nombre d'états

$$A = \begin{pmatrix} P_{00} & P_{01} & \cdots & P_{0n-1} \\ P_{10} & P_{11} & \cdots & P_{1n-1} \\ \cdots & & & \\ P_{n-1\,0} & P_{n-1\,1} & \cdots & P_{n-1n-1} \end{pmatrix}$$

La matrice carrée $A = (P_{ij})_{0 \leq i,j < n}$ est une matrice stochastique puisque chaque élément de A est positif ou nul et la somme des éléments de chaque ligne est 1.

Notons qu'à chaque état i, correspond la i-ième ligne

$$\left(P_{i0}, P_{i1}, P_{i2}, ..., P_{in-1} \right)$$

de la matrice de transition. Ce vecteur ligne de probabilité représente les probabilités de tous les résultats de transitions possibles à partir de l'état i à l'étape suivante.

9. En d'autres termes, son état à l'étape ℓ résume toute l'information utile pour son évolution future et une connaissance supplémentaire du passé du processus ne peut être utile pour améliorer cette prévision [10, 118, 90, 29].

L'élément P_{ij}^{ℓ} de la matrice de transition A^{ℓ}, la ℓ-ième puissance de A, représente la probabilité de passer de l'état i à l'état j au bout de t transitions.

Une chaîne de Markov est complétement définie par la donnée, d'une part des P_{ij}, $0 \leq i,j < n$ (i.e., la matrice de transition), et d'autre part des probabilités initiales $p_0(i)$, $0 \leq i < n$.

Soit

$$p_0 = \left(p_0(0), p_0(1), \ldots, p_0(n) \right)$$

le vecteur de distribution des probabilités initialles. Chaque composante $p_0(i)$ du vecteur p_0 représente la proportion d'objets dans l'état i au début du processus, et soit

$$p_\ell = \left(p_\ell(0), p_\ell(1), \ldots, p_\ell(n) \right)$$

le vecteur de distribution au bout de ℓ étapes. p_ℓ représente la proportion d'objets dans chaque état au bout de ℓ étapes. On a $\forall \ell \geq 0$,

$$p_\ell = p_{\ell-1} A = p_0 A^{\ell} \tag{6.1}$$

On sait que si A est une matrice *régulière*, alors la suite de matrices $(A^{\ell})_{\ell \geq 0}$ converge, quand $\ell \to \infty$, vers une matrice limite $\tilde{A}$, dont toutes les lignes sont identiques à un même vecteur de distribution $\tilde{p}$, et on a

$$\tilde{p} = \lim_{\ell \to \infty} p_\ell = p_0 \tilde{A} \tag{6.2}$$

Ce qui signifie que la suite $(p_\ell)_{\ell \geq 0}$ converge aussi vers un vecteur de distribution constant $\tilde{p}$. Ce vecteur est l'unique vecteur propre à gauche de A correspondant à la valeur propre $\lambda = 1$ dont les éléments sont positifs et leur somme est égale à 1.

Notons que, quand on dit que A^{ℓ} converge vers $\tilde{A}$, cela signifie que chaque élément de A^{ℓ} converge vers l'élément correspondant de $\tilde{A}$. De même, quand le vecteur p^{ℓ} converge vers $\tilde{p}$, cela signifie que chaque composante de p^{t} converge vers la composante correspondante de $\tilde{p}$.

La i-ème composante de $\tilde{p}$ représente la proportion approximative d'objets dans l'état i asymptotiquement, cette valeur limite étant *indépendante* de la distribution initiale p_0.

En fait, en général, les transitions de probabilités p_{ij} dépendent de l'état de départ, de l'état final et du nombre de transitions effectuées. Si $p_{ij}^{(\ell)}$ converge quand $\ell \to \infty$, alors les transitions de probabilités deviennent independantes du nombre variable ℓ, et on dit alors que la chaîne de Markov possède des transitions de probabilités stationnaires.

Ainsi, quand la matrice A est régulière, la probabilité $P_{ij}^{(\ell)}$ d'être dans un état j après ℓ transitions est approximativement égale à la $j$ème composante de $\tilde{p}$ et ceci indépendamment de l'état initial.

En effet, le fait que tous les coefficients de la matrice limite $\tilde{A}$ soient positifs, assure une certaine "homogénéisation" de la distribution des probabilités entre les différents états au cours du temps, indépendamment de la distribution initiale [29]. En conséquence, la

probabilité P_{ij}^t pour qu'un état j finisse par être réalisé pour un t suffisamment grand est indépendante de l'état initial i et elle est approximativement égale à la $j$ème composante de $\tilde{p}$. Notons que ceci signifie aussi que toute suite de distribution de probabilités converge vers la distribution stationnaire $\tilde{p}$. La distribution $\tilde{p}$ est donc l'unique distribution de probabilités stationnaire par rapport à la matrice stochastique régulière A.

Si la matrice A et ergodique mais non régulière, on peut encore utiliser 6.2 pour obtenir la distribution d'états limite, mais celle-ci dépend de la valeur de la distribution initiale $p^{(0)}$.

Pour finir, il est important de noter que si A est doublement stochastique et régulière alors le vecteur $\tilde{p}$ est un vecteur uniforme ; tous ses éléments sont positifs et égaux à $1/n$. D'après les théorèmes 7 et 8, on a le théorème suivant

Theorème 9 *Si A est **régulière et doublement stochastique** alors*

$$\tilde{p} = \lim_{t\to\infty} p_0 A^t = \left(\tfrac{1}{N}, \tfrac{1}{N}, \ldots, \tfrac{1}{N} \right).$$

et

$$\tilde{A} = \begin{pmatrix} \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \\ \vdots & & & \\ \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \end{pmatrix}$$

Nous pouvons maintenant revenir à l'algorithme du plongement régulier d'arbres quelconques de M sommets dans un réseau de N processeurs, pour montrer que la charge de ce plongement est en $O(\frac{M}{N} + \varepsilon)$, ce qui est optimal pour $M = \Omega(N\varepsilon)$, puisque la charge d'un plongement est toujours au moins $\Omega(M/N)$.

6.9 Analyse du plongement

Soit T un arbre quelconque d'arité maximale d que l'on veut plonger sur un graphe hôte de N sommets. L'algorithme aléatoire de plongement opère de la manière suivante. Supposons qu'un sommet u de l'arbre est placé dans P_j. A chacun des nœuds fils de u générés, on lui affecte aléatoirement un sommet de l'ensemble $\mathcal{V}(P_j)$ de cardinalité d comme cela a déjà été décrit. La probabilité qu'un élément de $\mathcal{V}(P_j)$ soit choisi est $1/d$. Le choix des destinations est effectué localement dans P_j et ne dépend que de P_j. En conséquence, l'état du système à une étape ℓ ne dépend que de l'étape précédent $\ell - 1$. Un processus stochastique qui satisfait cette propriété est une chaîne de Markov finie dont l'espace des états est l'ensemble des N sommets du graphe hôte.

En effet, nous voulons étudier un système évoluant de façon aléatoire entre un nombre fini d'états. On connaît par le biais de la fonction de projection régulière, pour chaque paire d'états i, j et pour chaque étape ℓ, la probabilité a_{ij} pour que l'on passe à l'étape suivante $\ell + 1$ à l'état j, si on se trouvait initialement dans l'état i.

On construit sa matrice de transition $A = (a_{ij})_{0 \leq i,j < N}$ sachant que

$$a_{ij} = \begin{cases} \frac{1}{d} & si\ j \in \mathcal{V}(i) = \{n(i,1), \ldots, n(i,d)\} \\ 0 & sinon \end{cases}$$

La distribution des nœuds de l'arbre peut être considérée comme étant une distribution de probabilités, et le placement des nouveaux sommets générés devient mathématiquement identique à l'évolution d'une chaîne de Markov.

Soit p_t le vecteur de distribution au bout de t étapes. Chacune de ses N entrées $p_t(i)$ représente la proportion d'objets (i.e., des nœuds de l'arbre) dans l'état i (i.e., le sommet i du graphe hôte) à l'étape t. Soit p_0 le vecteur de distribution des probabilités initial. D'après (6.1), à l'étape t, on a

$$p_t = p_0 A^t,\ \forall t \geq 1.$$

D'après l'analyse faite dans la section 6.8, on sait que si la matrice A est regulière, alors la chaîne de Markov possède des probabilités de transition stationnaires étant donné que $\lim_{t \to \infty} A^t$ existe.

Ceci signifie que pour des valeurs suffisamment grandes de t, la probabilité d'être dans un état i après t transitions est approximativement égale à $\frac{1}{N}$ (la $i$ème entrée de $\widetilde{p}$) et ceci indépendamment de l'état initial. En d'autres termes, comme conséquence du comportement asymptotique d'un tel processus de Markov, la distribution converge vers une distribution uniforme qui alloue approximativement le même nombre de nœuds à chaque sommet asymptotiquement. On conclut donc que la charge est $O(\frac{M}{N})$ pour M suffisamment grande.

Pour le montrer, soit la variable aléatoire X_i telle que $X_i = 1$ (resp. $X_i = 0$) si le sommet i est (resp. n'est pas) choisi pour recevoir un noeud de l'arbre. X_i est une variable aléatoire de Bernoulli et suit la loi de Bernoulli i.e., $X_i \sim B(1, \frac{1}{N})$ (Une variable aléatoire de Bernoulli est une variable aléatoire prenant les valeurs 0 ou 1 et plus précisément, X_i est une variable aléatoire de Bernoulli ssi $Prob[X_i = 0] + Prob[X_i = 1] = 1$).

Soit X le nombre de nœuds placés dans un processeur i. Etant donné que l'espérance mathématique de X_i est $E(X_i) = \frac{1}{N}$, on a en conséquence

$$E(X) = E(\sum_{i=1}^{M} E(X_i)) = M \times \frac{1}{N}$$

ce qui sgnifie que le nombre moyen de nœuds $E(X)$ placés dans i est $\frac{M}{N}$.

Malheureusement, l'analyse que nous venons de présenter non seulement demande à M d'être très grand pour que la probabilité de succés de X_i soit proche de $1/N$ mais de plus elle est imcomplète et peut nous induire en erreur. En effet, considérons l'algorithme de plongement pour lequel la matrice de transition correspondante est la matrice unité. Par exemple, pour $N = 4$, on a:

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Cette matrice est doublement stochastique. Puisque $\forall t \geq 1\ I^t = I$, la suite de matrices $(I^t)_{t\geq 0}$ admet la matrice I_4 elle même comme matrice limite. En appliquant 6.2, on a

$$\tilde{p} = \lim_{t\to\infty} p^t = p_0 I$$

Cependant I n'est pas régulière étant donné qu'il n'existe pas de matrice puissance de I dont toutes les entrées sont positives (de plus, les valeurs propres de la matrice diagnonale I sont les éléments de sa diagonale, et on en déduit que 1 est valeur propre de I de multiplicité N). La distribution limite $\tilde{p}$ dépend donc de la distribution initiale p_0. Supposons qu'au départ, la racine v de l'arbre est placée dans le sommet de numéro i. D'après la matrice de transition, tous les descendants de v seront placés également dans le même sommet i. Ceci signifie que la charge du plongement est égale à M. Cet exemple illustre donc bien l'intêret que la matrice de transition soit régulière.

Nous arrivons maintenant à la deuxième partie de l'analyse. En fait, la chaîne de Markov peut être divisée en deux phases, une phase transitoire suivie d'une phase stationaire. L'analyse précédente ne tient pas compte de la première phase. En fait, nous allons montrer qu'aucun sommet ne peut recevoir plus que $O(\frac{M}{N} + \varepsilon)$, où ε dépend de la vitesse de convergence du processus de Markov.

D'après le théorème 6, si A est regulière, alors 1 est une valeur propre simple et toutes les autres valeurs propres de A ont leurs modules majorés par 1. A peut être écrite sous la forme $A = PDP^{-1}$, où D est une matrice diagonale avec les valeurs propres de A sur sa diagonale. On a $A^k = PD^kP^{-1}$. Les entrées de D^k sont les $k$ièmes puissances des valeurs propres de A. La valeur sous-dominante γ d'une matrice est sa seconde valeur propre la plus grande après la valeur propre 1 (i.e., dont le module est différent de 1). Intuitivement, A^t converge d'autant plus rapidement que sa seconde valeur propre γ est proche de 0. D'après Cybenko [49], si γ est la valeur propre sous-dominante de A, alors on a

$$\|pA^\ell - \bar{p}\|^2 \leq \gamma^{2\ell}\|p - \bar{p}\|^2 \quad . \tag{6.3}$$

Montrons maintenant le résultat annoncé dans la section 6.5.

Soit p_ℓ la distribution de probabilité à l'étape ℓ. En appliquant (6.1), on sait que

$$p_\ell = p_{\ell-1}A$$

et en appliquant le théorème 9, on a

$$\tilde{p} = \lim_{t\to\infty} p_t = \left(\tfrac{1}{N}, \tfrac{1}{N}, \ldots, \tfrac{1}{N} \right)$$

Soit $c(v)$ le nombre de fois où le sommet v est choisi. On suppose que la chaîne de Markov converge à partir de l'étape t (i.e., on choisit le plus petit entier t tels que les composantes de p_t sont égales à $1/N$. Notons par $k^{(\ell)}$ le nombre de nœuds générés en parallèle à une

étape quelconque ℓ. Nous avons $M = \sum_{\ell=0}^{\bar{\ell}} k^{(\ell)}$, avec $\bar{\ell} \in [t, \infty[$. On a

$$\begin{aligned}
c(v) &= \sum_{\ell=0}^{\bar{\ell}} p_\ell(v) k^{(\ell)} \\
&= \sum_{\ell=0}^{t} p_\ell(v) k^{(\ell)} + \sum_{\ell=t}^{\bar{\ell}} p_\ell(v) k^{(\ell)} \\
&= \sum_{\ell=0}^{t} p_\ell(v) k^{(\ell)} + \frac{1}{N} \sum_{\ell=t}^{\bar{\ell}} k^{(\ell)} \\
&= \sum_{\ell=0}^{t} (p_\ell(v) - \frac{1}{N}) k^{(\ell)} + \frac{1}{N} \sum_{\ell=0}^{\bar{\ell}} k^{(\ell)} \\
&= \sum_{\ell=0}^{t} (p_\ell(v) - \frac{1}{N}) k^{(\ell)} + \frac{M}{N} \\
&= \underbrace{\sum_{\ell=0}^{t} (p_\ell(v) - \bar{p}_\ell(v)) k^{(\ell)}}_{\alpha(v)} + \frac{M}{N}
\end{aligned}$$

Nous obtenons donc le vecteur α tel que

$$\alpha = \sum_{\ell=0}^{t} (p_\ell - \bar{p}) k^{(\ell)}$$

Nous avons besoin maintenant de majorer α. D'après l'équation (6.3), nous savons que

$$\| \underbrace{A^\ell p_0}_{p_\ell} - \bar{p} \|^2 \leq \gamma^{2\ell} \| p_0 - \bar{p} \|^2$$

et donc

$$\| p_\ell - \tilde{p} \| \leq \gamma^\ell \| p_0 - \tilde{p} \|$$

et

$$\begin{aligned}
\| \alpha \| &= \| \sum_{\ell=0}^{t} k^{(\ell)} (p_\ell - \tilde{p}) \| \\
&\leq \sum_{\ell=0}^{t} k^{(\ell)} \| p_\ell - \tilde{p} \| \\
&\leq \sum_{\ell=0}^{t} k^{(\ell)} \gamma^\ell \| p_0 - \tilde{p} \|
\end{aligned}$$

Donc

$$\| \alpha \| \leq \| p_0 - \tilde{p} \| \sum_{\ell=0}^{t} k^\ell \gamma^\ell$$

Notons que nous avons utilisé le fait que le nombre $k^{(\ell)}$ de nœuds générés à une étape quelconque ℓ ne peut dépasser k^{ℓ}

on a alors

$$\|\alpha\| \leq \|p_0 - \tilde{p}\| \frac{(k\gamma)^{t+1} - 1}{k\gamma - 1}$$

Rappelons que la norme $\|.\|_\infty$ d'un vecteur désigne sa composante maximale. Désignons par $\varepsilon = \|\alpha\|_\infty$ la composante maximale du vecteur α.

Puisque

$$p_0 - \tilde{p} = \left(1 - \tfrac{1}{N}, -\tfrac{1}{N}, \ldots \right)$$

$$\|p_0 - \tilde{p}\|_\infty = 1 - \frac{1}{N}$$

on déduit que

$$\varepsilon \leq (1 - \frac{1}{N}) \frac{1 - (k\gamma)^{t+1}}{1 - k\gamma} \tag{6.4}$$

Theorème 10 *Soit un algorithme aléatoire de plongement régulier d'un arbre k-aire quelconque dans une topologie arbitraire, pour lequel la matrice de Markov correspondante à l'ensemble de visinage est régulière. Le nombre de nœuds placés sur un même sommet du réseau est au plus* $O(\frac{M}{N} + \varepsilon)$, *où* $\varepsilon \leq (1 - \frac{1}{N}) \frac{1 - (k\gamma)^{t+1}}{1 - k\gamma}$, γ *est la valeur propre sous-dominante de la matrice stochastique du plongement, t est le nombre d'étapes nécessaire pour la convergence, M est le nombre de nœuds et N le nombre de sommets du réseau.*

Notons que ε représente l'écart qui sépare la charge du plongement de la charge optimale $O(M/N)$.

D'après l'inégalité (6.4), pour des petites valeurs de γ, on obtient de faibles valeurs pour ε. Rappelons que le module de $\gamma appartient] - 1, 1]$. Examinons la valeur de ε dans le cas particulier pour lequel γ a des valeurs proche de $\frac{1}{k}$. Soit $\gamma = \frac{1}{k} + \alpha$. On calcule le developpement en série de Taylor de ε autour de $\frac{1}{k}$. Ceci révèle que $\varepsilon \leq t + 1$, où t est la vitesse de convergence de A (notons que t est borné).

Remarque importante

Il est important de noter que le résultat présenté dans le théorème 10 s'applique non seulement aux arbres de degré borné mais s'applique également pour les graphes de degré borné.

En fait le travail que nous avons effectué est le suivant. On dispose d'un réseau (i.e., le graphe hôte H) dont chaque sommet dispose d'un voisinage physique donné. Nous avons construit pour chaque sommet du réseau un voisinage logique. Nous avons donc construit au dessus du graphe hôte un autre graphe "logique" H_L. La matrice de transition que nous avons étudié correspond à une marche alátoire dans ce graphe H_L (i.e., à partir d'un sommet de H_L, on peut aller dans un de ses voisins avec une probabilité uniforme).

Comme cela a déjà été précisé, une chaine de Markov converge vers une distribution stationnaire limite *unique* si la matrice de transition A est ergodique. Deux conditions sont nécessaire pour que A soit ergodique [27]:

(i) *irréductibilité*: $\forall i,\ j, \exists\ \ell$ tel que $A^\ell(i,j) > 0$.

(ii) *apériodicité*: le p.g.d.c de l'ensemble $\{\ell, A^\ell(i,j) > 0\}$ est égale à 1.

Les deux conditions d'ergodicité de la matrice de transition A sont équivalentes à deux conditions sur le graphe sous-jacent H_L. en effet:

- (i) est équivalente au fait que le graphe est connecté. En d'autres termes, le graphe ne peut être décomposé sous forme de composantes connexes indépendantes. Cette condition est équivalente au fait que 1 est une valeur propre simple.

- (ii) est équivalente au fait que le graphe est non-biparti[10]. Cette condition est équivalente au fait que -1 ne peut être une valeur propre de A. Un exemple illustrant ce cas est donné plus loin (voir Exemple 3, cas 2).

Notons que les deux conditions ensembles (i) et (ii) traduisent le fait que la matrice est régulière (i.e., $\exists\ \ell > 0$ telque tous les éléments de A^ℓ sont positifs). Rappelons aussi que si de plus la matrice de transition est doubelement stochastique alors la distribution limite est uniforme. Notons que A est doubelement stochastique revient à dire que le graphe H_L est régulier.

6.10 Exemples

Le théorème 10 offre un aspect pratique qui permet de comparer les performances, au niveau de la charge, de différents choix de fonctions de projections. Pour illuster cet intérêt pratique, nous allons donner dans ce qui suit des exemples d'algorithmes de plongement qui peuvent être formulé sous forme de chaînes de Markov. Nous allons construire pour chaque exemple la matrice stochastique correspondante et calculer sa valeur propre sous-dominante. Pour simplifier les écritures nous supposons que les arbres à plonger sont binaires.

Exemple 1

Pour le premier algorithme, on considère le cas particulier dans lequel, à partir de chaque sommet du graphe hôte, on peut accéder, avec la même probabilité, à n'importe quel autre sommet du graphe. Plus précisément, on pose $\mathcal{V}(P_j) = N$, pour chaque sommet P_j d'un graphe hôte ayant N sommets. En d'autres termes, chaque sommet P_j peut accéder à tout autre sommet P_i parmi les N sommets du graphe hôte, avec une probabilité $1/N$. On construit la matrice de transition correspondante qui possède la forme suivante, pour

10. Un graphe $G(U,V,E)$ est biparti s'il est formé à partir des deux ensembles de sommets U et V connectés par des arêtes de E, et les sommets de U ne sont connectés qu'à des sommets de V et vice versa [81].

On veut plonger un arbre binaire quelconque dans un réseau de N processeurs. On suppose que les processeurs sont disposés en anneau, le processer i est connecté aux processeurs $(i-1)\ mod\ N$ et $(i+1)\ mod\ N$ (i.e., ces deux voisins à gauche et à droite). On peut choisir une fonction de projection de la manière suivante. Si un sommet est placé dans le processur i, alors ses fils gauche et droit s'il existent seraient placés, avec une probabilité uniforme 1/2, dans les processeurs voisins. Ce plongement possède évidemment une dilation 1. Nous sommes ici aussi en présence d'une chaîne de Markov dont la matrice de transition s'écrit pour $N = 4$:

$$A_5 = \begin{pmatrix} 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \end{pmatrix}$$

La matrice de transition A_5 que l'on obtient ne possède pas de limite. En effet, cette dernière est une matrice périodique. Cela s'explique par le fait que si on se trouve dans un état d'indice pair à une étape donnée, on se trouvera dans un état impair à l'étape suivante. Le comportement de l'algorithme est un comportement alternatif entre les indices pairs et les indices impairs. Notons que le graphe sous-jacent est biparti. La figure 6.3 illustre ce graphe.

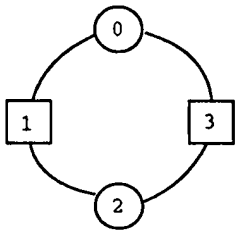

FIG. 6.3 - *Le graphe H_L correspondant au plongement du cas 2.*

On vérifie, à l'aide du logiciel numérique Matlab, que -1 est une valeur propre de A_5. On ne peut donc analyser le comportement de cet algorithme de plongement à l'aide du théorème 10.

Notons que la suite $(A_5^\ell)_{\ell \geq 0}$ possède deux matrices limites.

$$A_5^{2.\ell} = \begin{pmatrix} 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \end{pmatrix} \quad A_5^{2.\ell+1} = \begin{pmatrix} 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \end{pmatrix}$$

Il n'existe donc pas de distribution limite, en général. Bien sûr, si on suppose que la distribution des probabilités initiale est

$$p_0 = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$$

on obtient un état stationnaire $p = pA$, puisque p_0 est un vecteur propre associé à la valeur propre 1. Cet état stationnaire ne peut être atteint quelle que soit la distribution initiale. Il peut être atteint par exemple avec $p_0 = \left(0\,,\,\frac{1}{2}\,,\,\frac{1}{2}\,,\,0\right)$, mais pas avec $p_0 = (1\,,\,0\,,\,0\,,\,0)$, ce qui est plus réaliste (on commence par exemple par mettre la racine dans le processeur 0). Cet exemple souligne une fois de plus, l'importance du fait que la matrice de transition soit régulière pour assurer l'existence d'une distribution limite unique atteinte quelque soit la distribution initiale.

cas 3 :

Dans cet exemple, nous illusrons le cas où la matrice ne peut être régulière à cause d'un état *absorbant*. Dans une chaine de Markov, un état i est dit état absorbant si sa ligne dans la matrice de transition contient 1. la matrice suivante est une exemple de ce cas :

$$A_6 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

A l'opposé du cas où la matrice est régulière et dans lequel une répartition uniforme de la probabilité s'opère au cours du temps, dans le cas de la matrice A_6, la distribution limite possède la forme $p = (1\,,\,0\,,\,0\,,\,0)$. Plus précisément, asymptotiquement, le système se trouvera et restera dans l'état de numéro 0 avec une probabilité 1.

6.11 L'utilisation des opérations de mouvement de données

Lorsque nous avons considéré le problème de plongement d'arbres quelconques dans les réseaux synchrones de type hypercube et de type grille, nous avons transformé ce problème en un problème de mouvement de données. Dans le chapitre pécédent, nous avons montré que l'on peut réduire à une constante multiplicative près le temps nécessaire pour assurer les communications inter-nœuds de l'arbre plongé, en choisissant soigneusement la fonction de projection. Rappelons que nous avons utilisé des fonctions de projection irrégulières ; les valeurs de ce type de fonctions sont calculées conjointement par tous les processeurs d'une façon distribuée. Ceci induit un surcoût additionnel introduit au plongement ce qui rend cette stratégie inefficace. On peut alors appliquer la méthode de plongement aléatoire pour obtenir des algorithmes de plongement basés sur des fonctions de projection régulières et utilisant les mouvement de données.

A titre d'exemple, reprenons l'exemple donnée dans la section 6.4. Pour plonger un arbre quelconque d'arité d sur un graphe G de N sommets, nous avons défini le voisinage logique d'un sommet P_j en posant $n(j,\delta) = (jd+\delta) \bmod N$ pour $0 \leq j \leq N-1$ et $1 \leq \delta \leq d$.

Afin de mieux observer le comportement de cet algorithme de plongement, nous avons verifié que les matrices de transition correspondantes pour $d = 2$ (i.e., arbres binaires) et $d = 4$ (i.e., arbres quaternaires) sont régulières, et nous avons calculé les valeurs propres sous-dominantes de ces matrices de transition pour différentes valeurs de N.

Ces valeurs listées dans la table suivante montre que la convergence des deux matrices est

$N = 4$:

$$A_1 = \begin{pmatrix} 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{pmatrix}$$

La matrice A_1 est doublement stochastique et régulière étant donné que toutes ses entrées sont positives. Dans le cas de matrices de cette forme, on peut verifier que 0 est une valeur propre de multiplicité $N - 1$. En effet, 1 doit être une valeur propre simple pour A_1. Notons aussi que $A_1 = \widetilde{A}$ et on déduit donc que quel que soit le vecteur de distribution p, on a $\widetilde{p} = pA_1$. En appliquant le théorème 10, on déduit que la charge du plongement est $O(M/N + 0.75)$ ($\varepsilon = 0.75$ puisque $\gamma = 0$).

Exemple 2

Nous considérons trois algorithmes de plongement d'arbres binaires pour lesquels les deux premiers ensembles de voisinage logique sont respectivement $\mathcal{V}_1(P_j) = \{P_{2\times j+1}, P_{2\times j+2}\}$ et $\mathcal{V}_2(P_j) = \{P_j, P_{j+1}\}$. Pour le troisième algorithme, nous avons determiné l'ensemble de voisinage aléatoirement sachant que $card(\mathcal{V}_3) = 2$ (nous avons laissé la machine déterminer cet ensemble à l'aide d'un petit programme en C).

Les deux matrices de transistion correspondantes aux deux premiers exemples sont les suivantes. On les écrit ci-dessous pour $N = 4^{11}$:

$$A_1 = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0.5 \end{pmatrix} A_2 = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0.5 \end{pmatrix}$$

Pour $N = 16$, et à l'aide du logiciel numérique Matlab, nous avons calculé les valeurs propres sous-dominantes des trois matrices.

λ_1	λ_2	λ_3
4.74×10^{-5}	0.706	0.980

Ces matrices sont doublement stochastiques et régulières. En conséquence, la distribution stationnaire est uniforme. On peut donc appliquer le théorème 10.

Soit v le sommet le plus chargé du réseau. Nous avons montré que la chagre $c(v)$ ne peut dépasser $O(M/N + \varepsilon)$. Plus précisément, nous avons montré que l'écart qui sépare $c(v)$ de la charge optimale $O(M/N)$ est

$$\varepsilon = O((k\lambda)^t).$$

Rappelons que t représente le nombre d'étapes nécessaires à la chaine de Markov pour qu'une distribution p_ℓ rejoint la distribution stationaire uniforme, en partant d'une distribution initiale quelconque p_0.

11. Nous avons écrit les deux premières matrices pour $N = 4$ juste pour les illustrer. Dans le reste de l'exemple, les calculs sont effectués pour $N = 16$. On ne peut donc écrire ici la troisième matrice 16×16.

Malheureusement, nous ne pouvons pas déterminer précisément la valeur de t. Pour les deux premiers exemples d'algorithmes, et pour $N = 16$, nous avons calculé la valeur de t expérimentallement en exécutant un programe en C qui calcule la suite A^ℓ, jusquà ce qu'à atteindre la matrice limite $\widetilde{A}$. Pour les deux matrices A_1 et A_2, la valeur de de t obtenue est respectivement 4 et 201.

Nous avons calculé des valeurs de ε, listées dans la table suivante, pour chacun des trois algorithmes (réferencées par $\varepsilon_1, \varepsilon_2, \varepsilon_3$), et pour $3 \leq s < t$.

	λ_1	λ_2	λ_3
	4.74×10^{-5}	0.706	0.980
t	ε_1	ε_2	ε_3
3	0.93	4.13	6.38
4	0.93	6.77	13.45
5	0.93	10.50	27.33
6	0.93	15.77	54.56
7	0.93	23.21	107.97
8	0.93	33.73	212.73
9	0.93	48.58	418.22
10	0.93	69.55	821.32
11	0.93	99.17	1612.02

On déduit donc que l'écart ε est d'autant petite si la seconde valeur propre est proche de 0. Ainsi, l'algorithme de plongement correspondant à la matrice A_1 est meileur que ceux correspondants aux matrice A_2 et A_3, en terme de charge.

Exemple 3

Dans cet exemple, nous avons illustré les trois cas pour lequels le théorème 10 ne s'applique pas. Plus précisément, il s'agit des cas où la distribution limite uniforme n'existe pas.

cas 1 :

considérons un algorithme de plongement d'arbres binaires pour lequel la matrice de transistion correspondante est la suivante, pour $N = 4$:

$$A_4 = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \end{pmatrix}$$

La forme de la matrice indique que le graphe sous-jacent de la matrice peut être partionné en deux composantes indépendantes. En effet, on ne jamis atteindre les processeurs 3 et 4 s'il on se trouvait dans les processeurs 1 ou 2. Pour $N = 16$, et à l'aide du logiciel numérique Matlab, on vérifie que 1 est une valeur propre double. La matrice A_4 n'est pas réguliere ($A_4^\ell(1,3)$ restera toujours égal à 0), on ne peut donc appliquer le résultat du théorème 10.

cas 2 :

très rapide.

	λ	
N	arité=2	arité=4
8	1.37×10^{-6}	3.09×10^{-9}
16	4.74×10^{-5}	2.63×10^{-9}
32	3.99×10^{-4}	3.47×10^{-6}
64	1.95×10^{-3}	3.18×10^{-6}
128	4.64×10^{-3}	4.53×10^{-5}

En reprenant ce qui a été présenté dans le chapitre précédent, on sait que si le voisinage logique ci-dessus est utilisé quand $d = 2$ alors on peut utiliser les mouvements de données d'une manière optimisée. En effet, nous avons déjà montré que les contraintes relatives à l'utilisation optimisée sont toujours vérifiées. Notons que la même preuve reste valable pour $d = 4$.

6.11.1 Expérimentations sur la MasPar

Afin de valider l'approche de plongement, nous avons implémenté un algorithme de plongement qui utilise le voisinage logique décrit ci-dessus pour $d = 2$, sur la grille bidimensionnelle $N = 127 \times 127$ de la machine MasPar. Nous avons plongé un arbre binaire quelconque qui croît d'une manière quelconque au cours de l'algorithme (comme cela est posé dans le problème enoncé par Bhatt et Cai dans [17] et par Leighton dans [81]). La table ci-dessous montre les résultats en termes de charge obtenus dans le cas où l'aspect aléatoire est introduit pour équilibrer la charge, et dans le cas où cet aspect n'est pas utilisé. Plus précisément, l'algorithme de plongement est aléatoire quand il utilisé comme cela a été présenté ci-dessus. Quand l'aspect aléatoire n'est pas utilisé, le premier fils d'un sommet de l'abre placé dans un processeur j est affecté au processeur $n(j, 1)$, le deuxième fils s'il existe est toujours affecté au processeur $n(j, 2)$. La table 6.1 ci-dessous montre bien que l'introduction de l'aspect aléatoire dans l'algorithme de plongement permet d'éviter des conséquences désastreuses au niveau de la charge.

6.12 Expérimentations en utilisant PVM et PM²

Comme cela a été dejà mentionné, la méthode de plongement aléatoire présentée dans ce chapitre peut être utilisé quelle que soit la topologie du réseau de communication. En effet, l'analyse du comportement d'un algorithme de plongement régulier ne dépend que de la fonction de projection associée, sachant qu'elle implique un processus de Markov.

Afin de valider notre approche de plongement, nous avons implémenté la méthode de plongement sur un réseau de stations de travail. Nous avons utilisé tout d'abord l'environnement de programmation parallèle pour architecture distribuée PVM [114]. Cet environnement est basé sur le concept de processus lourd. Nous avons donc été vite limité quant au nombre de processus crées par machine ; le nombre de processus PVM toléré sur une même station de travail est limité à 64.

TAB. 6.1 - *Cette table liste la charge obtenue en distribuant M sommets d'un arbre quelconque, avec et sans utiliser l'algorithme alátoire, sur la grille $N = 127 \times 127$ de la MasPar.*

Nombre de sommets M	Charge sans l'algorithme aléatoire	Charge avec l'algorithme aléatoire	Charge idéale $\frac{M}{N}$
3123	93.00	1.17	1
55791	219.00	4.59	3.41
65135	4.00	4.00	3.98
99325	16.00	6.82	6.06
204383	64.00	13.43	12.47
731923	64.00	45.86	44.67
1640123	256.00	102.59	100.11

Nous avons donc utilisé l'environnement de programmation parallèle "multithreadé" PM^2 [116]. Cet environnement est basé sur le concept de processus légers (threads) et permet l'exécution de plusieurs centaines de thread à l'intérieur d'un processus lourd (jusqu'à épuisement des ressources de la machine). Comme cela a été fait sur la MasPar, on a plongé des arbres binaires quelconques qui croîent d'une manière quelconque au cours de l'algorithme (ce qui correspond à la primitive *fork* d'Unix). Les tables 6.2 et 6.3 montre les résultats en termes de charge obtenus dans le cas où l'algorithme aléatoire est utilisé pour équilibrer la charge, et dans le cas où cet algorithme n'est pas utilisé, pour $N = 8$ et $N = 16$ respectivement.

TAB. 6.2 - *Cette table liste, pour $N = 8$ stations, la charge effective de chaque station obtenue avec et sans utiliser l'algorithme alátoire. La valeur ToTal représente le nombre total de sommets de l'arbre quelconque plongé.*

N=8			
Numéro Station	Charge sans l'algorithme aléatoire	Charge avec l'algorithme aléatoire	Charge idéale $\frac{ToTal}{N}$
0	4171	16106	17271.50
1	4012	16164	17271.50
2	7302	17050	17271.50
3	7289	16012	17271.50
4	3605	16627	17271.50
5	3412	16597	17271.50
6	51722	20394	17271.50
7	56659	19222	17271.50
ToTal	138172.00		

TAB. 6.3 - *Cette table liste, pour $N = 16$ stations, la charge effective de chaque station obtenue avec et sans utiliser l'algorithme alátoire.*

N=16			
Numéro Station	Charge sans l'algorithme aléatoire	Charge avec l'algorithme aléatoire	Charge idéale $\frac{ToTal}{N}$
0	290	1299	1258.50
1	293	1366	1258.50
2	487	1137	1258.50
3	515	1306	1258.50
4	286	1213	1258.50
5	218	1082	1258.50
6	902	1329	1258.50
7	843	1269	1258.50
8	228	1124	1258.50
9	230	1157	1258.50
10	462	1278	1258.50
11	354	1098	1258.50
12	199	1240	1258.50
13	144	1256	1258.50
14	7372	1459	1258.50
15	7313	1523	1258.50
ToTal	20136.00		

6.13 Conclusion et perspectives

Dans ce chapitre, nous avons présenté une approche de plongement aléatoire d'arbres quelconques de M sommets dans un réseau quelconque de N sommets. A l'aide de processus de Markov, nous avons analysé le comportement alátoire et nous avons montré que la charge du plongement est en $O(M/N+\varepsilon)$. La valeur ε dépend de la fonction de projection choisie.

En effet, le choix d'une fonction de projection bien définie implique la donnée d'une matrice de Markov A. Si cette matrice vérifie des conditions d'ergodicité et elle est doublement stochastique alors au bout d'un certain nombre d'étapes t, la chaine de Markov converge vers une distribution stationaire unique et uniforme. La vitesse de convergence de la chaine dépend de λ, la valeur propre sous-dominante de A. Nous avons ainsi montré que $\varepsilon = O((k\lambda)^t)$. Rappelons que ε mesure l'écart entre la charge effective du plongement et une charge optimale en $O(M/N)$.

Le problème suivant reste ouvert. Pour construire un plongement, nous utilisons une fonction de projection bien définie. Le choix d'une fonction précise implique, par construction,

le choix de la dilatation et de la congestion du plongement. La question posé est quelle est la relation entre la vitesse de convergence de la chaine de Markov vers la distribution stationnaire, la dilatation et la congestion. La réponse à cette question pourrait donner la borne minimale sur la dilatation et la congestion pouvant être obtenues dans un plongement possédant une charge en $O(\frac{M}{N})$.

Chapitre 7

Perspectives

L'approche de plongement aléatoire et dynamique que venons de décrire dans le chapitre 6 a concernée le plongement d'arbres quelconques de degré borné dans une topologie arbitraire (i.e, un réseau de processeurs ou un réseau de stations de travail).

Dans ce qui suit, nous résumons brièvement le travail effectué afin de présenter les améliorations que l'on peut apporter a cette approche et d'énoncer quelques questions laissées comme problèmes ouverts. Les deux premiers points d'améliorations nous ont été proposés par F.Chung [27].

Comme cela a été vu dans le chapitre précédent, nous avons montré comment plonger un arbre quelconque dans un réseau donné. Comme c'est généralement le cas, le réseau est modélisé par un graphe appelé graphe de voisinage physique. Nous avons construit au dessus de ce graphe, un graphe de voisinage logique en attribuant à chaque sommet un voisinage logique qui peut ne pas correspondre à ces voisins physiques dans le réseau. La démarche a consité ensuite à effectuer le plongement d'un arbre quelconque sur ce nouveau graphe.

L'analyse de Markov que nous avons utilisée pour étudier le comportement aléatoire de l'algorithme de plongement, a revélé que l'efficacité de ce dernier, en terme de charge, est liée à la manière selon laquelle le graphe logique est construit (i.e., le choix des voisins logiques de chacun des sommets). En effet, considérons la matrice d'adjacence M relative au graphe de voisinage logique. Nous avions imposé que ce graphe soit régulier. Soit k le degré de chaque sommet du graphe. Nous obtenons ainsi la matrice de Markov, doublement stochastique, de la manière suivante:

$$A = \frac{1}{k} M$$

Nous avons vu dans le chapitre précédent que si A est une matrice régulière alors la charge du plongement est étroitement liée à la valeur propre sous dominante de cette matrice.

Notons par ailleurs, que la chaîne de Markov que nous avons utilisée est en fait une marche alátoire. En effet, à partir d'un sommet du graphe de vosinage logique, on peut aller dans un de ses voisins avec une probabilité uniforme.

- Il nous semble possible que le résultat décrit dans le chapitre 6 pour les arbres de

degré borné reste valide et peut être généralisé pour les graphes de degré borné.

- Les graphes de voisinage logique que nous avons utilisé dans l'analyse étaient des graphes réguliers. Cette régularité n'est pas nécessaire et on peut étendre les résultats à des graphes plus généraux, sachant que ces derniers sont connectés et non bipartis. Ces deux conditions assurent que la chaîne de Markov associée à ces graphes converge vers une distribution unique et stationnaire, indépendemment de la distribution initiale. les entrées de cette distribution sont:

$$\tilde{p}(x) = \frac{d_x}{\sum_y d_y}$$

A titre d'application, si tous les sommet ont le même degré (i.e., le graphe est régulier), les entrées de $\tilde{p}$ sont les valeurs $\frac{1}{N}$, N étant le nombre de sommets du graphe.

- L'approche du plongement présentée peut être interprétée comme étant le composé de deux plongements. Le premier plomgement est celui du graphe de voisinage logique dans le graphe de voisinage physique. Ce plongment est déterministe et statique. La dilatation et la congestion de ce plongement déterministe varient donc selon le graphe de voisinage logique choisi. Le deuxième plongement, celui que nous avons étudié, est le plongement aléatoire et dynamique d'un arbre quelconque sur le graphe de voisinage logique. La question se posait alors : pourrions-nous comparer les valeurs propres sous-dominates respectives des deux graphes de voisinage physique et logique?. Il semble que la réponse est positive et consiste à utiliser le *théorème de comparaison* de F.Chung, décrit dans son livre récent (1997). F.Chung montre en effet le théorème suivant:

Theorème 11 *[27] Soient G et G' deux graphes réguliers ayant les valeurs propres respectives λ et λ' et les degrés respectifs k et k'. On suppose que chaque arête de $\{x, y\}$ de G est étirée sur un chemin $P(x, y)$ dans G' avec une distante au plus ℓ. De plus, on suppose que chaque arête de G est contenue dans au plus m chemins $P(x, y)$. On a alors:*

$$\lambda' \geq \frac{k\lambda}{k'\ell m}$$

Il est important de préciser cependant que la valeurs propre λ (resp. λ') est déterminée à partir d'une matrice appelée matrice *Laplacian* de G (resp. G'). Cette matrice est définie de la manière suivante:

$$\mathcal{L}(u, v) = \begin{cases} 1 & si\ u = v\ et\ d_v \neq 0 \\ \frac{-1}{\sqrt{d_u d_v}} & si\ u\ et\ v\ sont\ adjacents \\ 0 & sinon \end{cases}$$

Si G est un graphe régulier de degré k, alors $\mathcal{L} = I - \frac{1}{k}M$, soit $\mathcal{L} = I - A$, M étant la matrice d'adjacence de G et A la matrice stochastique de Markov avec laquelle on a travaillé. Les valeurs propres de $\mathcal{L}$ sont non-négatives et elles sont représentées ainsi $0 = \lambda_0 \leq \lambda_1 \leq \ldots \lambda_{n-1}$ dans [27]. En conséquence, la valeur propre sous-dominante

λ que nous avons utilisée dans le chapitre précédent correpond à $1 - \lambda_1$, λ_1 étant la deuxième valeur propre la plus petite de $\mathcal{L}$.

F. Chung assure dans son livre [27] qu'étudier les valeurs propres (le spectre de $\mathcal{L}$, d'où le titre du livre) en utilisant la notion de "Laplacien" permet de traiter des graphes généraux et non seulement les graphes réguliers et symétriques comme c'est le cas la pluspart du temps dans la littérature.

- Le problème suivant reste ouvert. Pour construire un plongement, nous définissons un graphe de voisinage logique. Ceci revient à dire que nous utilisons une fonction de projection bien définie. Le choix d'une fonction précise (i.e., un graphe de voisinage logique précis) implique , par construction, le choix de la dilatation et de la congestion du plongement. En effet, le plongement de l'arbre dans le graphe de voisinage logique possède une dilatation 1. La question posée est quelle est la relation entre la vitesse de convergence de la chaine de Markov vers la distribution stationnaire, la dilatation et la congestion. La réponse à cette question pourrait donner la borne minimale sur la dilatation et la congestion pouvant être obtenues dans un plongement possédant une charge optimale en $O(\frac{M}{N})$.

Table des figures

1.1 modèle d'exécution MIMD à mémoire partagée 18
1.2 modèle d'exécution MIMD à mémoire distribuée 19
1.3 Le modèle d'exécution SIMD 23
1.4 hypercube de dimension 4 30
1.5 Les réseaux (a) grille 2D 4x4 et (b) tore 2D 4x4 31
1.6 un arbre binaire complet de hauteur 4 32
1.7 Une implémentation directe d'un arbre ayant 8 feuilles dans une grille 2D. Le lien connectant la racine au niveau 4 à un de ses deux fils au niveau 3 a une longueur proportionnelle à 2^3. Ce lien possède la longueur maximale. . 32
1.8 Un arbre en H de 6 niveaux dans une grille 2D 33
1.9 Un X-arbre de 8 feuilles (a), une pyramide 4×4 (b) et une multigrille 4×4 (c) 34
1.10 L'arbre élargi (fat-tree) implanté sur la CM5 34
1.11 La grille d'arbres 4×4 35

2.1 Plongement d'une grille 4×4 dans un hypercube de 16 sommets. La dilation du plongement est 1 (les liens en pointillés sont les liens de l'hypercube non utilisés par le plongement de la grille. 40
2.2 Plongement de dilataion 2 d'une grille 3×5 dans un hypercube de 16 sommets. Par exemple, l'arête connectant les deux sommets de la grille $(2,2)$ et $(3,2)$ est projetée sur deux arêtes de l'hypercube [81]. 41
2.3 Exemple d'un plongement d'un arbre binaire complet de 15 sommets dans une grille de 4×15 sommets. Les arêtes de la grille indiquées en traits pleins représentent les chemins associés aux arêtes de l'arbre. Les arêtes de la grille indiqués en traits pointillés sont inutilisées. Les sommets de la grille images des sommets de l'arbre sont représentés par des cercles pleins. 42
2.4 Plongement en H d'un arbre binaire complet de 15 sommets dans une grille de 3×7 sommets. 42
2.5 Numérotation infixée d'un arbre binaire complet. 43

2.6 Plongement de dilataion 2 d'un arbre binaire complet sur un hypercube. . . 44

3.1 Une grille étendue 4×4 50

3.2 Plongement en H d'un arbre T_7 sur une grille bidimensionnelle.La partie encadrée désigne l'unité de base qui représente un arbre de hauteur 3. 51

3.3 Contraction de degré 4 d'un arbre binaire. 52

3.4 (a) Le plongement d'un arbre contracté $T_c(\lceil \frac{h}{2} \rceil)$ (i.e., $T(h)$) est construit d'une manière recursive à partir des 4 exemplaires de plongement de ses 4 sous-arbres $T_c(\lceil \frac{h}{2} \rceil - 1)$ (i.e., $T(h-2)$). La racine est plongée dans un sommet situé au centre de la grille. Les étiquettes au dessus des sommets indiquent les niveaux dans l'arbre, avant contraction, des racines des sous arbres contenus dans R et r_i. (b) Ce schéma illustre le plongement d'un arbre $T_c(3)$. 53

3.5 (a).Le plongement d'un arbre contracté est construit d'une manière recursive à partir des 4 exemplaires du plongement de ses sous arbres jusqu'à atteindre l'unité de base à la hauteur $\lceil \frac{h_0}{2} \rceil$. (b).Ce schéma illustre le plongement d'un arbre $T_c(3)$. . 55

3.6 La dilatation du plongement est la distance séparant R_h et R_{h-2}. Le plongement donné ici est le plongement en X(4, 3, 3) de la figure 3.5. 57

3.7 Comportement du X-Net suivant la distance et la direction de la communication. Les communications selon une direction en diagonale (NE, NW, SE, SW) sont plus lentes que les communications selon une direction orthogonale (N, E, S, W). Les quatre droites de chaque cas se trouvent confondues dans la figure. 64

3.8 Temps d'exécution en secondes de l'algorithme implanté avec les deux plongements en H et en X(3, 4, 4) en fonction de la hauteur de l'arbre. 64

3.9 les temps d'exécution de l'algorithme tri par fusion (merge-sort) obtenus en implantant ce dernier avec chacun des trois plongements. 66

3.10 Tracé du facteur d'accélération obtenu en utilisant le plongement en $X_4(3, 5, 5)$ par rapport à un plongement en H. 66

3.11 Tracés du temps d'exécution de l'algorithme tri par fusion en séquentiel et en parallèle en utilisant le plongement en $X_4(3, 5, 5)$ sur la MaPar. 67

4.1 Exemple d'une opération RAW. Le problème ici est celui de router les enregistrements de $1 \rightarrow 2$, $4 \rightarrow 5$, $5 \rightarrow 6$. Dans un premier temps, on calcule le rang de ces enregistrements en utilisant Rank. On les concentre avec Concentrate. On les distribue ensuite vers leurs destinations exactes avec Distribute. Si les destinations n'etaient pas monotones, on aurait d'abord triés les enregistrements par destination, puis il faut les router vers leurs destinations finales en utilisant Distribute. . . 74

4.2 Exemple d'une opération RAW. Dans cette opération plusieurs enregistrements ont la même destination. Sort trie les enregistrements en fonction de leurs destinations. Rank calcule les rangs. Les enregistrements de même destination se font attribuer le même rang. Concentrate concentre et fusionne les enregistrements de même rang. Distribute route dans une dernière étape les données vers leurs destinations finales. 76

4.3 Exemple d'une opération RAR. Dans cette opération, plusieurs processeurs veulent lire des données détenues par un seul processeur. On commence par trier les enregistrements par destination. Les indices des enregistrements avant le tri sont rangés dans une variable T. Un enregistrement parmi ceux qui ont la même destination est séléctionné (les enregistrements séléctionnés sont désignés par des astérisques). Une suite convenable d'appels des procédures Rank, Concentrate et Distribute ramène les données supposées contenues dans des registres $df(i)$. Ces données destinées à être copiées par la suite sont concentrées par Concentrate puis copiées en utilisant Generalize. Les copies creées ainsi sont envoyées ensuite vers leurs destinations finales en utilisant Sort suivant la clé donnée par T. 77

4.4 (a) numérotation par mélange de ligne d'abord. (b) numérotation par ligne d'abord 79

4.5 Exemple d'emploi de la procédure Rank 80

4.6 Exemple d'emploi de la procédure Concentrate 81

4.7 Exemple d'emploi de la procédure Distribute 82

4.8 Exemple d'emploi de la procédure Generalize 83

5.1 Deux situations de collisions possibles durant une étape donnée. Dans la situation (1), l'enregistrement a reste sur place et l'enregistrement b va se dépalcer vers le processeur P_i. Dans la situation (2), a et b se déplacent tous les deux vers un processeur P_k. 97

5.2 Un exemple de routage R^+. Chaque enregistrement dans i emprunte un chemin pour rejoindre sa destination $d(i)$ de la manière suivante. Durant l'itération b, le b-ième bit de l'indice de son processeur initial i est remplacé par le b-ième bit de $d(i)$. Si les deux bits sont identiques, l'enregistrement reste sur place durant cette itération. Par contre, si les deux bits sont différents, l'enregistrement traverse l'arête vers le processeur dont le b-ième est complémentaire de celui de i. Notons qu'il n'y a jamais de collision entre deux enregistrements dans un même sommet. . 99

5.3 Un exemple de Distribution. Chaque enregistrement dans i emprunte un chemin pour rejoindre sa destination $d(i)$ de la manière suivante. Durant l'itération b, le b-ième bit de l'indice de son processeur initial i est remplacé par le b-ième bit de $d(i)$. Si les deux bits sont identiques, l'enregistrement reste sur place durant cette itération. Par contre, si les deux bits sont différents, l'enregistrement traverse l'arête vers le processeur dont le b-ième est complémentaire de celui de i. Notons qu'il n'y a jamais de collision entre deux enregistrements dans un même sommet. . 101

5.4 Plongement d'un arbre binaire complet de $N-1$ sommets dans une grille ou un hypercube de N processeurs. Les étiquettes des sommets de l'arbre désignent les processeurs dans lesquels ils sont projetés. Plus précisément, le i-ième sommet de l'arbre est projeté sur le i-ième processeur, et chaque sommet interne i a ses fils gauche et droit dans $2 \times i$ et $2 \times i + 1$, respectivement. 106

5.5 Plongement d'un arbre binaire quelconque. Les étiquettes des sommets de l'arbre désignent les processeurs dans lesquels ils sont projetés. Notons que les fils gauche et droit s'ils existent d'un sommet interne i ne se reçoivent pas nécessairement Les étiquettes $2i$ et $2i+1$. 108

5.6 L'effet de l'optimisation sur le temps d'exécution dans le cas d'un arbre binaire complet. 113

6.1 Le premier algorithme de plongement décrit par Leighton et al. appliqué à un arbre binaire complet et pour $N = 8$. Les étiquettes sur les sommets de l'arbre désignent les processeurs de l'hypercube sur lesquels ils sont projetés. La i-ème feuille est projetée sur le i-ème sommet de l'hypercube, et chaque sommet interne est projeté sur le même sommet que la feuille la plus à gauche de son arbre, $0 \leq i \leq N-1$. Les valeurs des bits d'oscillation ne font aucune différence dans ce cas d'un arbre binaire complet. Mais pour un arbre binaire quelconque, Leighton montre que le comportement aléatoire permet de plonger l'arbre avec une charge en $O(M/N + logN)$ avec une forte probabilité. 120

6.2 Un hypercube de cliques P_r de dimension $r = 4$ et possédant 2^4 sommets. Les arêtes ajoutés à l'hypercube de 16 sommets pour former l'hypercube de cliques de 16 sommets sont représentés en gras. Deux sommets sont voisins s'ils diffèrent d'un seul bit, ou s'ils diffèrent sur leur $\lfloor logr \rfloor$ derniers bits. La taille des cliques ici est 4. 121

6.3 Le graphe H_L correspondant au plongement du cas 2. 139

Liste des tableaux

1.1 Diamètres et bissections de quelques réseaux 35

5.1 optimisation des appels aux procédures de mouvement de données dans le cas des opérations fils→père et père→fils. 109

5.2 optimisation des appels aux procédures de mouvement de données dans le cas des opérations enfants→père et père→enfants. 110

6.1 Cette table liste la charge obtenue en distribuant M sommets d'un arbre quelconque, avec et sans utiliser l'algorithme alátoire, sur la grille $N = 127 \times 127$ de la MasPar. 142

6.2 Cette table liste, pour $N = 8$ stations, la charge effective de chaque station obtenue avec et sans utiliser l'algorithme alátoire. La valeur ToTal représente le nombre total de sommets de l'arbre quelconque plongé. 142

6.3 Cette table liste, pour $N = 16$ stations, la charge effective de chaque station obtenue avec et sans utiliser l'algorithme alátoire. 143

Bibliographie

[1] A.Dingle and I.H.Sudborough. Simulation of binary trees and x-trees on pyramid networks. *Journal of Parallel and Distributed Computing*, 19(2), October 1993.

[2] A.Rau-Chaplin A.Ferreira, F.Dehne. *Algorithmique SIMD: théorie et applications. Algorithmique parallèle, chapitre 4.* Masson, 1992.

[3] A.K.Gupta and S.E.Hambrusch. Multiple network embeddigs into hypercubes. *Journal of Parallel and Distributed Computing*, 19(2), October 1993.

[4] Selim G. AKL. *The design and analysis of parallel algorithms.* Prentice-Hall, 1989.

[5] H. M. Alnuweiri and V. K. Prasanna. Efficient parallel computations on the reduced mesh of trees organization. *Journal of Parallel and Distributed Computing*, 20(2):121–135, February 1994.

[6] A.M.Gibons and M.S.Paterson. Dense edge-disjoint embedding of complete binary trees in interconnection networks. *CS-RR-208*, February 1992.

[7] A.M.Gibons and R.Ziani. The balanced binary tree technique on mesh connected computer. *Information Processing Letters*, 37(2):101–109, January 1991.

[8] S. Andresen. The looping algorithm extended to base 2^t rearrangeable switching networks. *IEEE Transactions on Computers*, vol. C25(no.10):197–203, October 1977.

[9] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, vol 15(no. 1):3–43, March 1983.

[10] A. ARNOLD and I. DE GUESSARIAN. *Mathématiques pour l'informatique.* Logique Mathématiques Informatique, Masson, 1992.

[11] A.S.Wagner. Embeddigs all binary trees in the hypercube. *Journal of Parallel and Distributed Computing*, 18(1), May 1993.

[12] A.S.Wagner. Embeddig the complete tree in the hypercube. *Journal of Parallel and Distributed Computing*, 20(2):241–247, Feb 1994.

[13] A. BEKLEY. Contribution à l'étude des réseaux d'inteconnexion des machines parallèles - utilisation des hyperfréquences -. *Thèse LIFL*, 1994.

[14] A. BEKLEY. Multiplication de matrices par blocs sur hypercom. *Rapport interne LIFL, à paraître*, 1994.

[15] J.L. BENTLY and H.T. KUNG. A tree machine for searching problems. *in Proc. IEEE Int. Conf. Parallel Processing, pp.257-266*, 1979.

[16] S. BHATT, F. CHUNG, T. LEIGHTON, and A. ROSENBERG. Optimal simulations of tree machines. *In 27th Annual Symposuim on Foundations of Computer Science, pp. 274-282, IEEE, October*, 1986.

[17] S.N. BHATT and J.-Y. CAI. talk a walk, grow a tree. *in Proc. 29th Annual IEEE Symposuim on Foundations of Computer, Science, IEEE CS, Washington, DC, pp. 469-478*, 1988.

[18] T. BLANK. The maspar mp-1 architecture. *IEEE computers*, pages 20–24, 1990.

[19] B.Monien and H.Sudborough. Embedding on interconnection network in another. *computing suppl. Spring-Verlag*, 7:257–282, 1990.

[20] B.Monien, R.Feldmann, R.Klasing, and R.Luling. Parallel architecture: Design and efficient use. *TR-93-124, Department of Computer Science, University of Paderborn, Germany*, 1993.

[21] H. BOURZOUFI. un modèle de programmation logique parallèle. *Thèse LIFL*, 1992.

[22] R. BRANSON. *Calcul matriciel.* Série Schaum, McGraw-Hill, 1994.

[23] R. BRIKI. *Thèse, Université des Sciences et Technologies de Lille, à paraître*, 1998.

[24] Ecole CAPA. *Chapitre5: UNIX: communications et parallélisme.* Conception et analyse des algorithmes parallèles, 1995.

[25] David A. Carlson. Modified mesh-connected computers. *IEEE Computer*, 37(10):1315–1321, October 1988.

[26] P. CHAVEL and N. DE BEAUCOUDREY. *Technologies Matérielles Futures de l'Ordinateur.* Editions Frontières, 1992.

[27] Fan.R.K. CHUNG. *Spectral Graph Theory.* AMS., 1997.

[28] P. Clermont. *Algorithmique parallèle, chapitre 8.* Masson, 1992.

[29] A. COMBROUZE. *Probabilités et statistiques.* Collect. Major, Press Universitaire de France, 1993.

[30] M. COSNARD, M. NIVAT, and Y. ROBERT. *Algorithmique parallèle.* Masson, 1992.

[31] A. Despain and D. Patterson. X-tree: A structured multiprocessor computer architecture. *Proc. IEEE 5th Annu. Symp. Comput. Arch.*, pages 144–151, April 1978.

[32] G.R. DESROCHERS. *Principles of Parallel and Multiprocessing, Chapter6: Performance Analysis.* Intertext Publications and McGraw-Hill Book Company, 1987.

[33] D.Gordon, I.Coren, and G.M.Silberman. Embedding tree structures in vlsi hexagonal arrays. *IEEE Transactions on Computers*, January 1984.

[34] D.Nassimi and S.Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-28(1):2–7, Jan. 1979.

[35] D.Nassimi and S.Sahni. Data broadcasting in simd computers. *IEEE Transactions on Computers*, C-30(2):101–107, February 1981.

[36] D.Nassimi and S.Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[37] D.S.Scott and J.Brandenburg. Minimal mesh embeddings in binary hypercubes. *IEEE Computer*, October 1988.

[38] E.Ma and L.Tao. Embeddigs among meshes and tori. *Journal of Parallel and Distributed Computing*, 18(1), May 1993.

[39] Digital equip. corp. Decmpp programming language reference manual. *Maynard, Massachusetts, January*, 1992.

[40] D. ETIEMBLE. Impact de la technologie sur les architectures. *Technologies Matérielles futures de l'ordinateur, Mars*, 1992.

[41] D. ETIEMBLE. Evolution des machines parallèles et massivement parallèles. *Ecole d'été des jeunes chercheurs en architecture*, 1994.

[42] F.Baude. Tree contraction on distributed-memory parallel architectures, department of computing and information science, queen's university, kingston, canada. 1992.

[43] F.Dehne, A.G.Ferreira, and A.Rau-Chaplin. Parallel b&b on fine grained hypercube multiprocessor. *Parallel Computing*, 15(1-3):201–209, 1990.

[44] F.Dehne and A.Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8(4):367–375, April 1990.

[45] M.J. FLYNN. Some computer organizations and their effectiveness. *IEEE trans. on Computers, vol. C21, No. 9, September*, 1972.

[46] J. Gaber, B. Toursel, G. Goncalves, P. Lecouffe, and T.HSU. Non-numerical data parallel algorithms. *Proc. of 2nd Euromicro Int. Workshop on Parallel and Distributed Processing PDP'94, University of Malaga, Published by IEEE Computer Society, December*, 1993.

[47] G.Aharoni, Y.Farber, and A.Barak. A strategy for the run-time management of fine-grained parallelism. *CSTR 91-07 Tech. reports series, Dept. of Electronics and Computer Science, University of Southampton, 509 SNH*, June 1991.

[48] G.C.Plaxton. Load balancing, selection and sorting on the hypercube. *In Proceedings of the 1st Annual ACM Symposuim on Parallel Algorithms and Architectures, Santa Fe, New Mexico.*, pages 64–73, June 1989.

[49] G.CYBENKO. Dynamic load balancing for distributed memory architecture. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[50] A. GOTTLIEB. Architectures for parallel supercomputing. -, 1992.

[51] W. GROPP. Early experiences with the IBM SP1 and the high-performance switch. *Technical Report ANL-93/41, Argone National Laboratory, November*, 1993.

[52] J.L. HENNESSY and D.A. PATTERSON. *Architecture des Ordinateurs - Une approche quantitative.* Mc Graw-Hill; édition française par D. Etiemble et M. Israel, 1992.

[53] V. HEUN and E.W. MAYR. Efficient dynamic embedding of arbitrary binary trees into hypercubes. *In Proc. of the 3rd IRREGULAR'96, Lecture Notes in Computer Science 1117, Springer-Verlag*, pages 287–298, 1996.

[54] M.-C. HEYDEMANN, J. OPATRNY, and D. SOTTEAU. Embedding of complete binary trees into extended grids with edge congestion 1. *Research Report no. , LRI, Université de Paris Sud, Centre d'Orsay, France*, 1996.

[55] W.D. HILLIS. *La machine à connexions.* Edtion française par F. Lustman (ed Masson), 1988.

[56] E. HOROWITZ and A. ZORAT. A divide and conquer computer. *Tech. Rep., Dept. Compt. Sci., Univ. of Southern California, July*, 1979.

[57] E. HOROWITZ and A. ZORAT. The binary tree as an interconnection network. *Proc. of the 1980 conf. on networks, April*, 1980.

[58] E. HOROWITZ and A. ZORAT. The binary tree as an interconnection network: Applications to multiprocessor systems and vlsi. *IEEE Transactions on Computers*, C-30(4):247–253, Appril 1981.

[59] E. HOROWIZE and A.ZORAT. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, C-32:582–585, 1983.

[60] H.Samet. Hierarchical representations of collections of small rectangles. *ACM computing surveys*, 20(4):271–309, December 1988.

[61] T. HSU. Proposition d'une architecture de réseau d'interconnexion à reconfiguration dynamique et asynchrone. *Thèse, Université des Sciences et Technologies de Lille, Février*, 1993.

[62] K. HWANG. *Advanced Computer Architecture.* Mc GRAW-HILL, 1993.

[63] I.Koren. A reconfigurable and fault-tolerant vlsi multiprocessor array. *Proc. 8th annu. symp. comput. arch.*, pages 65–112, May 1981.

[64] INTEL. The paragon xp/s system at a glance. *Intel*, 1992.

[65] J. Jàjà and K.W. Ryu. Load balancing and routing on the hypercube and related networks. *Journal of Parallel and Distributed Computing*, 14:431–435, 1992.

[66] J.Gaber and B.Toursel. An optimization of data movement operations: Application to the tree-embedding problem. *5nd Euromicro Int. Workshop on Parallel and Distributed Processing PDP'97, Published by IEEE Computer Society, London, UK*, January 1997.

[67] J.Gaber and B.Toursel. Randomized load distribution of arbitrary trees on a distributed network. *accepté et à paraître dans 1998 ACM Symposium on Applied Computing SAC'98, Atlanta, USA*, February 27 - March 1 1998.

[68] J.Gaber, B.Toursel, and G.Goncalves. Embedding arbitrary trees in the hypercube and the q-dimensional mesh. *1996 International Conference on High Performance Computing HPC'96, IEEE, ACM-SIGARCH, India*, Dec 19 - Dec 22 1996.

[69] J.Gaber, B.Toursel, and G.Goncalves. Plongements d'arbres quelconques dans une grille. *Poster, Renpar8, Bordeaux, France*, 20-24 mai 1996.

[70] J.Gaber, B.Toursel, G.Goncalves, and T.Hsu. Embedding tree structures in mpcs: Application to the maspar mp-2. *Conference on Massively Parallel Computing Systems MPCS'94: the challenges of General-Purpose and Special-Purpose Computing, Euromicro-IEEE computer Society, Ischia, Italy*, May 1994.

[71] J.Gaber, B.Toursel, G.Goncalves, and T.Hsu. Embedding tree structures in massively parallel computers. *1995 ACM Symposium on Applied Computing SAC'95, Nashville, USA*, Feb 26-28 1995.

[72] J.Gaber, B.Toursel, G.Goncalves, and T.Hsu. Embedding trees in massively parallel computers. *Journal of Systems Architecture*, 42(3):165–170, October 1996. Elsevier N.H.

[73] M. Jiber, A.Bellaachia, J.Gaber, and B.Toursel. Embedding algorithms of trees into square x-mesh. *Technical Report AS-97-178, Lifl, Lille University, France*, June 1997.

[74] Mouna Jiber. Embedding quadtrees into x-mesh. *Master Thesis, Alakhawayn University, Ifrane, Morocco*, 1997.

[75] J.JÀJÀ. *Parallel algorithms.* Addison-Wesley, 1992.

[76] S. KARLIN and H-M. TAYLOR. *A first course in stochastic processes.* Academic Press Inc., 1975.

[77] K.Brockmann and R.Wanka. Efficient oblivious parallel sorting on the maspar mp-1. *TR-96-042, International Computer Science Institute, Berkley*, 1996.

[78] K.E.Batcher. Sorting networks and their applications. *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[79] KENDALL SQUARE RESEARCH. Technical summary. -, 1992.

[80] H. KUNG. Why sytolic architectures? *Computers, vol. C15, No. 1, January*, 1982.

[81] F.T. LEIGHTON. *Introduction to parallel algorithms and architectures.* Morgan Kauffmann Publishers., 1992. Traduit en Français par P.FRAIGNAUD et E.FLEURY, International Thomson publishing France, 1995.

[82] F.T. LEIGHTON, M.J. NEWMAN, A.G. RANADE, and E.J. SCHWABE. Dynamic tree embeddings in butterflies and hypercubes. *Siam Journal on computing*, 21(4):639–654, August 1992.

[83] T. LEIGHTON, B.M.MAGGS, A.G. RANADE, and A.W. RICHA. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, July 1994.

[84] T. LEIGHTON, B.M.MAGGS, and S. RAO. Packet routing and job-shop scheduling in o(congestion + dilation) steps. *Cominatorica*, 14(2):167–180, July 1994.

[85] T. LEIGHTON, B.M.MAGGS, and A.W. RICHA. Fast algorithms for finding o(congestion + dilation) packet routing schedules. *Combinatorica, to appear.*

[86] T. LEIGHTON, B. MAGGS, and S. RAO. Universal packet routing algorithms. *In 29th Annual IEEE Symposuim on the Foundations of Computer Science, pp. 256-269*, 1988.

[87] J. Lenfant. Parallel permutation of data: A Benes networks control algorithm for frequently used permutations. *IEEE Transactions on Computers*, vol.C27(no.7):204–214, July 1978.

[88] D. LENOSKI, J. LAUDON, K. GHARACHORLOO, W. WEBER, A. GUPTA, J. HENNESSY, M. HOROWITZ, and M. S. LAM. The stanford dash multiprocessor. *IEEE Computer, March*, 1992.

[89] K. LI. Analysis of randomized load distribution for reproduction trees in linear arrays and rings. *Proc. 11th Annual International Symposium on High Performance Computers, Winnipeg, Manitoba, Canada (10-12), July*, 1997.

[90] S. LIPSCHUTZ. *Probabilités.* Série Schaum, McGraw-Hill, 1995.

[91] D. LITAIZE. Architectures multiprocesseurs à mémoire commune. *Deuxième symposium sur les Architectures Nouvelles des Machines PRC ANM - CNRS - MRT, September*, 1990.

[92] D. LITAIZE. La liaison à ultra haut débit: une (la?) solution pour les liaisons intermodule en environnement multiprocesseur. *Technologies Matérielles Futures de l'ordinateur, Mars*, 1992.

[93] Neil B. MacDonald. An overview of simd parallel systems: Amt dap, thinking machines cm-2000, & maspar mp-1. *Workshop on parallel Computing, Quaid-i-Azam University, Pakistan*, 26th-30th April 1992.

[94] P. MARQUET. Langages explicites à parallélisme de données. *Thèse LIFL*, 1992.

[95] E.W. Mayr and G.C.Plaxton. Pipelined parallel prefix computations, and sorting on a pipelined hypercube. *Department of Computer Science, Stanford University, Technical Report STAN-CS-89-1261, 16 pages.*, May 1989.

[96] E.W. Mayr and G.C.Plaxton. Pipelined parallel prefix computations, and sorting on a pipelined hypercube. *Journal of Parallel and Distributed Computing*, 17:374–380, 1993.

[97] E.W. Mayr and G.C.Plaxton. Pipelined parallel prefix computations, and sorting on a pipelined hypercube. *Journal of Parallel and Distributed Computing*, 17:374–380, 1993.

[98] C. MEAD and L. CONWAY. *Introduction to VLSI systems.* Addison-wesley publishing company, ISBN 0-201-04358-0, 1980.

[99] R. MELHEM and D. CHIARULLI. Optical computing and interconnection systems. *J.P.D.C. vol.17*, 1993.

[100] M.Jerrum and A.Sinclair. Conductance and the rapid mixing property for markov chains: The approximation of the permanent resolved. *ACM STOC'1988*, pages 235–244, 1988.

[101] M.Jiber and A.Bellaachia. Embedding quadtree structures on x-mesh networks. *Proc. 10th Annual International Symposium on High Performance Computers, Ottawa, Canada (5-7), June*, 1996.

[102] M.Jiber, A.Bellaachia, J.Gaber, and B.Toursel. Camparison of three embedding algorithms of quadtrees into x-nets: Application on maspar machines. *Proc. 11th Annual International Symposium on High Performance Computing systems HPCS'97, Canada*, 18-20 July 1997.

[103] M.Mihail. Conductance and convergence of markov chains - a combinatorial treatment of expanders. *1989 IEEE??*, pages 526–531, 1989.

[104] M.Rottger, U.P.Schroeder, and J.Simon. Virtual topology library for parix. *TR, University of Paderborn, Germany*, 1995.

[105] M.S.Paterson, W.L.Ruzzo, and L.Synder. Bounds on minmax edge length for complete binary trees. *Proc. STOC, Milwakee*, pages 293–299, 1981.

[106] R. MULLER and Q.F.STOUT. Data mouvement techniques for the pyramid computer. *Siam Journal on computing*, 16(1):38–60, February 1987.

[107] T. MUNTEAN and P. WAILLE. L'architecture des machines supernode. *La lettre du transputer, 7:11-40, September*, 1990.

[108] J. R. NICKOLLS. The deseign of the MASPAR MP1: a cost effective massively parallel computer. *IEEE Digest of papers-CompCon*, 1990.

[109] W. OED. The cray research massively parallel processor system CRAY T3D. *Cray Research GmbH, November 15*, 1993.

[110] P.S. PACHECO. *Parallel programming with MPI.* Morgan Kaufmann publishers, 1997.

[111] Ian PARBERRY. *Problems on algorithms.* Prentice Hall, 1995.

[112] R. PICKERING and J. COOK. A first course in programming the decmpp/sx. *Parallab, dept. of informatics, University of Bergen, N-5020 Bergen, Norway,* 1993.

[113] P.Kulasinghe and S.Bettayeb. Embeddings binary trees into crossed cubes. *IEEE Transactions on Computers,* 44(7):923–929, July 1995.

[114] P.Vandekan, F.Moyen, and G.Goncalves. Guide du débutant pvm. *Rapport EUDIL,* 1994.

[115] R.Namyst. Pm^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières. *Thèse LIFL,* 1996.

[116] R.Namyst and J.F.Méhaut. Pm^2 : Parallel multithreaded machine, guide d'utilisation, version 1. *Rapport LIFL,* 1996.

[117] A.L. ROSENBERG. Data encodings and their costs. *Acta information, pp.273-292,* 1978.

[118] A. RUEGG. *Processus stochastiques.* Méthodes Mathématiques pour l'Ingenieur, Press Polytechniques Romandes, 1989.

[119] Ecole RUMEUR. *Communications dans les réseaux de processeurs.* Institut d'études scientifiques de Cargesse, Corse, 1992.

[120] P. SAINRAT. Réseau d'interconnexion du multiprocesseur m3s - etude et mise en oeuvre. *Thèse, Université Paul Sabatier, Janvier,* 1991.

[121] J-C.Wang S.B.Choi and N.Yeh. Embedding meshes on the star graph. *Journal of Parallel and Distributed Computing,* 19(2), October 1993.

[122] S.Bhatt and J-Y.Cai. Taking random walks to grow trees in hypercubes. *Journal of the ACM,* 40(3):741–764, July 1993.

[123] C.L. SEITZ. The cosmic cube. *Communications of the ACM, vol 28, Number 1, January,* 1985.

[124] A. SEZNEC, A.M. KERMARREC, and T. VAULERON. Etude comparée des architectures des microprocesseurs MIPS R4000, DEC 21064 ET T.I. SUPERSPARC. *IRISA,* 1992.

[125] A. SEZNEC and Y. MEVEL. Etudes des architectures des microprocesseurs dec 21164, ibm power2 et mips r8000. *Rapport de rechecrhe 2553, INRIA, Juin,* 1995.

[126] A. SEZNEC and T. VAULERON. Etude comparative des architectures des microprocesseurs INTEL PENTIUM ET POWERPC 601. *IRISA,* 1994.

[127] X. Shen, Q. Hu, and W. Liang. Embedding k-ary complete trees into hypercubes. *Journal of Parallel and Distributed Computing*, 24:100–106, 1995.

[128] L. SNYDER. Introduction to the reconfigurable, highly parallel computer. *IEEE Computers, vol. C15, No. 1, January*, 1982.

[129] S.Ravindram, A.M.Gibbons, and M.S.Paterson. Dense edge-disjoint embedding of complete binary trees in interconnection networks. *TR, University of Warwick, England*, 1993.

[130] S.Q. STOUT. Sorting, merging, selecting and filtering on tree and pyramid machines. *in Proc. IEEE Int. Conf. Parallel Processing*, pages 214–221, 1983.

[131] R. SUAYA and G. BIRTWISTLE. *VLSI and Parallel Computation.* Morgan Kauffman Ed., 1990.

[132] S.Ullman and B.Narahari. Mapping binary precedence trees to hypercubes. *Parallel Processing Letters*, 2(1):81–87, March 1992.

[133] A. TANENBAUM. *Structured Computer Organization.* Prentice-Hall, 1990.

[134] P. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. *Technical Report TR-8802, Rice University, Department of Electrical and Computer Engineering.*, February 1988.

[135] A. WAGNER and D. CORNEIL. Embedding trees in a hypercube is np-complete. *Siam Journal on computing*, 19(3):570–590, June 1990.

[136] L. ZHANG. Emulations and embeddings of meshes of trees and hypercubes of cliques. *Thesis, University of Waterloo, Ontario, Canada*, 1995.