USTL
FLANDRES ARTOIS

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

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 430

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

NIAR Smaïl

CONTRIBUTION A L'ETUDE DES ARCHITECTURES D'ORDINATEURS PARALLELES : STRUCTURE DE LA MACHINE N-ARCH ET EMULATION SUR UN RESEAU DE TRANSPUTERS

Thèse soutenue le 24 Octobre 1989 devant la commission d'Examen

Membres du jury

V. CORDONNIER	Président
G. MAZARE	Rapporteur
M. MERIAUX	Rapporteur
B. TOURSEL	Directeur de thèse
G. GONCALVES	Examinateur
M.P.LECOUFFE	Examinateur

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

REMERCIMENTS

Je veux remercier, ici, Monsieur le professeur Vincent CORDONNIER pour l'honneur qu'il me fait de présider ce jury.

Je tiens a remercier, Monsieur Guy MAZARE, Professeur à l'IMAG de GRENOBLE, pour l'intérêt qu'il a porté à ce travail et pour avoir accepté d'en être le rapporteur.

Je tiens aussi à remercier Monsieur Michél MERIAUX, chargé de recherche au CNRS, d'avoir bien voulu examiner ce rapport, pour ses conseils et sa grande gentillesse.

Que Monsieur Bernard TOURSEL, professeur à l'EUDIL, trouve ici ma gratitude pour avoir bien voulu m'accepter au sein de l'équipe N-ARCH , pour les encouragements et les orientations qu'il m'a apporté depuis mon année de DEA.

Je suis extrêmement reconnaissant envers Madame Marie Paul LECOUFFE pour les conseils et les critiques constructives qui ont énormément contribués à la réalisation de ce travail.

Je tiens à remercier, tout particulièrement, Monsieur Gilles GONCALVES, maître de conférences à l'EUDIL, qui m'a suivi tout le long de ce travail , pour son soutien et sa disposition à tout moment. Qu'il soit assuré de ma gratitude et de ma profonde reconnaissance.

Je ne voudrais pas oublier de remercier tout les membres de l'équipe N-ARCH, pour le confort et la parfaite ambiance de travail qu'ils ont su créer.

Je remercie, enfin, toute ma famille pour la patience et les encouragements qu'ils m'ont apporté dans les moments les plus difficiles.

A MON PERE, MA MERE

MA FEMME, MES FRERES

A MA GRAND MERE

I INTRODUCTION 1

II LES ARCHITECTURES MULTIPROCESSEURS ET LES LANGAGES DECLARATIFS 5

II.1 LES NOUVELLES ARCHITECTURES 5

II.1.1 Les architectures controlées par les données 5

II.1.2 Les architectures controlées par les demande 9

II.1.2.1 La réduction de chaines 10

II.1.2.2 La réduction de graphes 11

II.2 LE PARALLELISME DANS LES LANGAGES DE PROGRAMMATION 13

II.2.1 Les langages impératifs 13

II.2.1.1 Les langages impératifs et les architectures parallèles 13

II.2.1.2 Conclusion 15

II.2.2 Les langages fonctionnels 16

II.2.2.1 Introduction 16

II.2.2.2 Evaluation parallèle d'un programme fonctionnel 17

II.2.2.3 Quelques améliorations 18

II.2.2.4 L'évaluation paresseuse 21

II.2.2.5 Les fonctions de haut niveau 22

II.2.3 Les langages de la programmation logique 23

II.2.3.1 Introduction 23

II.2.3.2 Exécution d'un programme PROLOG 23

II.3 INTEGRATION DES LANGAGES FONCTIONNELS ET LOGIQUES 25

II.4 PRESENTATION DE QUELQUES MACHINES PARALLES POUR LA PROGRAMMATION DECLARATIVE 27

II.4.1 La machine à combinateurs MaRS 27

II.4.1.1 Introduction 27

II.4.1.2 Les combinateurs 28

II.4.1.3 Implémentation de la machine MaRS 29

II.4 PRESENTATION DE QUELQUES MACHINES PARALLES POUR LA PROGRAMMATION DECLARATIVE 27
II.4.1 La machine à combinateurs MaRS 27
II.4.1.1 Introduction 27
II.4.1.2 Les combinateurs 28
II.4.1.3 Implémentation de la machine MaRS 29
II.4.1.4 Conclusion 30
II.4.2 La machine de réduction ALICE 31
II.4.2.1 Introduction 31
II.4.2.2 Le modèle d'exécution 31
II.4.2.3 Le prototype de la machine 33
II.4.3 La machine PIM-R 35
II.4.3.1 Introduction 35
II.4.3.2 Le modèle d'exécution 35
II.4.3.3 L'architecture de la machine PIM-R 37
II.4.3.4 Conclusion 39
II.5 CONCLUSION 40
III LA MACHINE N-ARCH 42
III.1 LES OBJECTIFS DU PROJET 42
III.2 PRESENTATION DE LA MACHINE 44
III.2.1 N-ARCH une machine multiprocesseurs 44
III.2.2 La répartition de la charge par fonctions de hachage 50
III.2.3 Les objets manipulés par la machine 52
III.2.4 Les unités fonctionnelles d'un noeud du réseau 53
III.2.4.1 L'organe de communication (Le commutateur) 53
III.2.4.2 L'organe de traitement 55
III.2.4.3 L'organe de mémorisation 57
III.3 EXECUTION DES PROGRAMMES DECLARATIFS 60
III.3.1 Exécution parallèle de programme FP 60
III.3.2 Exécution de programmes PROLOG sur N-ARCH 62
IV EMULATION DE LA MACHINE N-ARCH SUR UN RESEAU DE TRANSPUTERS 66
IV.1 INTRODUCTION

III.2.4.2 L'organe de traitement 55
III.2.4.3 L'organe de mémorisation 57
III.3 EXECUTION DES PROGRAMMES DECLARATIFS 60
III.3.1 Exécution parallèle de programme FP 60
III.3.2 Exécution de programmes PROLOG sur N-ARCH 62
IV EMULATION DE LA MACHINE N-ARCH SUR UN RESEAU DE TRANSPUTERS 66
IV.1 INTRODUCTION
Pourquoi un émulateur ? 66
IV.1.1 Test par simulation 66
IV.1.2 Test par la réalisation physique 67
IV.1.3 Test par émulation 68
IV.2 PRESENTATION DU LANGAGE OCCAM ET DU TRANSPUTER 69
IV.2.1 Le langage OCCAM
Présentation 69
IV.2.2 Les concepts de base dans OCCAM 70
IV.2.3 Implémentation du langage OCCAM sur Transputer 72
IV.2.4 Conclusion 77
IV.3 LES PAQUETS DE L'EMULATEUR 78
IV.3.1 Les paquets initialisation 78
IV.3.2 Les paquets requête 80
IV.3.2 Les paquets réponse 80
IV.4 LES COMPOSANTS DU NOYAU
Les processus OCCAM de N-ARCH 81
IV.4.1 Le processus d'entrée 85
IV.4.2 Le processus buffer d'entrée 85
IV.4.3 Le processus de contrôle 88
IV.4.4 Le processus mémoire associative des programmes 90
IV.4.5 Le processus de calcul 98
IV.4.6 Le processus mémoire associative des paquets 99
IV.4.7 Le processus buffer de sortie 103
IV.4.8 Le processus de sortie 104

IV.4.1 Le processus d'entrée 85
IV.4.2 Le processus buffer d'entrée 85
IV.4.3 Le processus de contrôle 88
IV.4.4 Le processus mémoire associative des programmes 90
IV.4.5 Le processus de calcul 98
IV.4.6 Le processus mémoire associative des paquets 99
IV.4.7 Le processus buffer de sortie 103
IV.4.8 Le processus de sortie 104
IV.4.9 Conclusion 107
IV.5 LES FONCTIONS DE HACHAGE DANS L'EMULATEUR 108
IV.5.1 La fonction de hachage globale 108
IV.5.2 La fonction de hachage locale 114
V TEST ET MESURES DE PERFORMANCES DU NOYAU N-ARCH 121
V.1 LES PROGRAMMES TESTES 122
V.1.1 Le langage de simulation 122
V.1.2 La répartition des programmes 124
V.1.3 L'exécution d'un programme 127
V.2 L'ACCELERATION THEORIQUE 128
V.2.1 Définitions 128
V.2.2 Le calcul de l'accélération théorique 129
V.2.3 L'accélération théorique matérielle et l'accélération théorique logicielle 132
V.3 LES COURBES DES TEMPS D'EXECUTION 135
V.3.1 Les courbes d'exécution sans collisions 135
V.3.2 Les courbes d'exécution avec collisions 141
V.4 CONCLUSIONS 145

VI EXTENSION DU NOYAU A L'EVALUATION DES FONCTIONS.....149
VI.1 INTRODUCTION.....149
VI.2 LE GRAPHE D'EXECUTION DES FONCTIONS.....150
VI.3 LA REPRESENTATION MEMOIRE DU GRAPHE.....152
VI.4 LES TYPES DE PAQUETS UTILISES.....154
VI.4.1 Les paquets d'initialisation.....155
VI.4.2 Les paquets requête et les paquets réponse.....156
VI.4.3 Les paquets requête-fonction.....156
VI.4.4 Les paquets requête-copie.....157
VI.5 LE TRAITEMENT DES PAQUETS PAR L'EMULATEUR.....158
VI.5.1 Sur réception d'un paquet initialisation.....158
VI.5.2 Sur réception d'un paquet requête.....158
VI.5.3 Sur réception d'un paquet requête-fonction.....159
VI.5.4 Sur réception d'un paquête-copie.....160
VI.5.5 Sur réception d'un paquet réponse.....162
VI.6 UN EXEMPLE.....163
VI.7. L'EVALUATION DE L'OPERATEUR IF.....167
VI.8 LA REPARTITION DES COPIES DE FONCTION.....170
VI.9 CONCLUSION.....172
CONCLUSION GENERALE.....173
REFERENCES.....176

CHAPITRE I
INTRODUCTION

De nos jours les domaines d'application des ordinateurs ne cessent de se développer. Cet accroissement s'accompagne malheureusement d'une augmentation importante du degré de complexité des logiciels utilisés et de leurs tailles. De plus certaines de ces applications exigent des temps de réponse très faibles (traitement temps réel ou intéractif).

Ceci a provoqué 2 types de demandes: des demandes concernant le logiciel (software) utilisé pour programmer ces machines et des demandes concernant le matériel (hardware) sur lequel se fait l'exécution des programmes. Pour répondre à ces demandes il faut donc modifier les systèmes de traitement de l'information existants aussi bien sur le niveau logiciel que sur le niveau matériel.

Sur le plan matériel, le progrès technologique rapide enregistré depuis la construction du premier ordinateur a permis la réalisation de circuits intégrés de plus en plus performants. Il est devenu alors possible aujourd'hui de produire à grande échelle des circuits intégrant un calculateur complet et à un prix réduit.

Néanmoins du point de vue vitesse d'exécution, ces circuits ont atteint leur limite technologique (hardware crisis). De ce fait le seul moyen permettant la construction d'ordinateur offrant une grande vitesse de traitement est (théoriquement) l'assemblage d'un grand nombre de calculateurs élémentaires.

La solution proposé ici ne vise donc pas à modifier la manière dont est construit le calculateur ou les éléments utilisés pour cette construction (technologie), mais plutôt à revoir la structure de la machine ou son organisation (on parle aussi d'*architecture* de la machine). En effet, depuis le célèbre ENIAC, construit dans les années 40, l'organisation de base des calculateurs n'a pas changé. Cette organisation est celle proposée par J.Von Neumann. Elle est composée :

- D'une unité de contrôle centralisée unique et séquentielle.
- D'une unité mémoire à accès aléatoire centralisée.
- Et d'un ou plusieurs ports d'entrée/sortie.

Depuis quelques modifications légères ont été réalisées sur le modèle sans pour autant modifier le mode de contrôle séquentiel d'un calculateur. C'est ainsi qu'il a été possible de réaliser des machine SIMD ("Single Instruction Multiple Data") ou MIMD ("Multiple Instruction Multiple Data") comme les machines CRAY ou DAP.

De nos jours les chercheurs travaillant dans le domaine de l'architecture des ordinateurs n'hésitent plus à dire que l'utilisation du modèle de Von Neumann limite les performances de la machine, et qu'il est quasiment inintéressant de retenir ses principes lorsque le nombre d'unités de traitement est important.

Sur le plan logiciel, le haut degré de complexité des nouvelles applications a rendu la tâche du programmeur très difficile (software crisis). Ce dernier doit alors fournir des programmes de tailles plus importantes, et de qualité meilleure (plus clairs, et plus sûrs). L'utilisation des langages classiques ou impératifs n'a pas permis d'atteindre ces objectifs.

Ceci est dû au fait que les langages impératifs imitent le fonctionnement de la machine. Ils ne sont en réalité que des versions de haut niveau de la machine de Von Neumann.

Le projet N-ARCH développé à l'université de LILLE I à pour objectifs de répondre aux 2 demandes citées plus haut. Sur le plan matériel, le projet se propose d'étudier et de développer une architecture de machine parallèle. Le mode de contrôle de cette machine est différent de celui de Von Neumann. Le second objectif est de montrer que l'architecture proposée peut très bien être utilisée pour supporter un nouveau type de langages de programmation. Il s'agit des langages déclaratifs. L'utilisation de ces langages constitue la solution proposée sur le plan logiciel.

Le travail que nous présentons constitue une contribution à ce projet. Celle-ci consiste à définir l'architecture de la machine (constitution d'un noeud) et à étudier les points qui s'y rattachent comme le réseau de communication, le partage des tâches entre les noeuds de la machine, ...etc. Un autre point important qui nous a été confié consiste à réaliser un émulateur de la machine N-ARCH en utilisant un réseau de Transputers.

Cette émulation devra nous permettre la mise au point et la validation de l'architecture proposée.

Ce rapport est constitué de 5 parties:

1er partie (chapitre II): Notions de base

Dans ce chapitre nous rappelons quelques notions de base sur les architectures parallèles et sur les langages déclaratifs. Ces notions devront permettre au lecteur de mieux situer le cadre de travail et de mieux comprendre le reste du rapport.

2eme partie (chapitreIII) : Présentation du projet

Dans cette partie nous discuterons des objectifs du projet N-ARCH. Nous présenterons ensuite l'architecture d'un noeud de la machine, ainsi que le mode de contrôle de celle-ci, et nous discuterons des possibilités d'intégration de certaines parties de cette architecture. Pour terminer, nous donnerons une présentation des autres travaux de recherches qui entrent dans le cadre du projet.

3eme partie (chapitre IV) : L'émulateur de la machine N-ARCH

Cette partie présente l'émulateur de la machine. Elle commence par une introduction aux éléments utilisés pour cette émulation (le Transputer et Occam), puis montre comment à partir de l'architecture de base développée dans le chapitre III, nous avons pu réaliser l'émulateur. Nous présentons à la fin de cette partie les solutions qui ont été adoptées pour résoudre les problèmes relatifs aux architectures parallèles (réseaux de communication, partages des tâches, routage,..etc).

4eme partie (chapitre V) : Résultats d'exécution

Dans ce chapitre nous présentons le matériel utilisé ainsi que le langage de simulation qui a été définie pour tester et mesurer les performances de l'émulateur. Par la suite nous donnons les résultats d'exécution de certains programmes de test. Ces résultats nous ont permis de voir le comportement de l'émulateur ainsi que l'influence du nombre de processeurs dans le réseau et de la taille du programme à exécuter sur les performances de la machine. Nous terminons cette partie par une série de remarques et de conclusions qui devraient permettre l'améliorations des performances de l'émulateur et par voie de conséquence montrer les points critiques sur lesquelles l'architecture d'un noeud N-ARCH devra s'établir.

5eme partie (chapitre 6) : Extensions du noyau

Cette dernière partie présente le travail qui reste à faire pour compléter le noyau. Une première ébauche est proposée pour permettre l'évaluation de fonctions. Cette ébauche peut être complétée et enrichie pour permettre l'exécution d'un langage fonctionnel réel.

Nous terminons ce rapport par une conclusion générale, où nous résumons les résultats auquels nous sommes arrivés à la fin de ce travail. Ces résultats concernent aussi bien l'architecture de la machine que le fonctionnement de l'émulateur.

CHAPITRE II
LES ARCHITECTURES MULTIPROCESSEURS ET LES LANGAGES DECLARATIFS

II.1 LES NOUVELLES ARCHITECTURES

Afin d'apporter des solutions aux problèmes posés par les architectures parallèles de type von Neumann, de nombreux chercheurs ont proposé de nouvelles architectures d'ordinateurs. Le mode de fonctionnement (ou mode de contrôle) de ces machines est très différent de celui d'une machine classique.

Les premiers travaux de recherche dans ce domaine furent commencés dans les années 70. Ils ont été réalisés par J.B.Dennis [Denn74] dans la conception d'une machine contrôlée par les données et par K.J.Berkling [Berk75] afin de concevoir une machine contrôlée par les demandes (ou par nécessité).

De ce fait, ces nouvelles architectures peuvent être classées en 2 catégories : d'une part les architectures contrôlées par les données (appelées aussi architectures contrôlées par disponibilité ou architectures data flow), et d'autres part les architectures contrôlées par les demandes (appelées aussi machines de réduction).

Ces 2 types d'architectures ont pour but commun l'augmentation des performances de la machine par l'exploitation du parallélisme inhérent aux programmes et l'accroissement du nombre de processeurs, mais diffèrent par la manière dont sont réalisés le mode de contrôle, la représentation des programmes et des données, et la gestion des ressources de la machine.

II.1.1 Les architectures contrôlées par les données

Dans une machine cadencée par le flot de données, les programmes sont en général représentés par un graphe orienté. Ce graphe permet de représenter le flot de données entre les instructions. Les noeuds du graphe représentent les opérateurs du programme alors que les arcs reliant les noeuds représentent la circulation des données entre opérateurs (dépendance de données).

Une instruction du programme est exécutée dès que tous les arguments qui lui sont nécessaires deviennent disponibles et si le dernier résultat produit par cette instruction a été consommé par l'instruction destination. Si on se réfère au graphe orienté qui représente le programme, et si nous prenons comme convention de représenter un argument par un jeton sur l'arc correspondant, nous pouvons dire qu'un noeud opérateur du graphe n'est activé que si chaque arc entrant dans le noeud contient un jeton et si l'arc de sortie n'en contient aucun (fig 2.1).

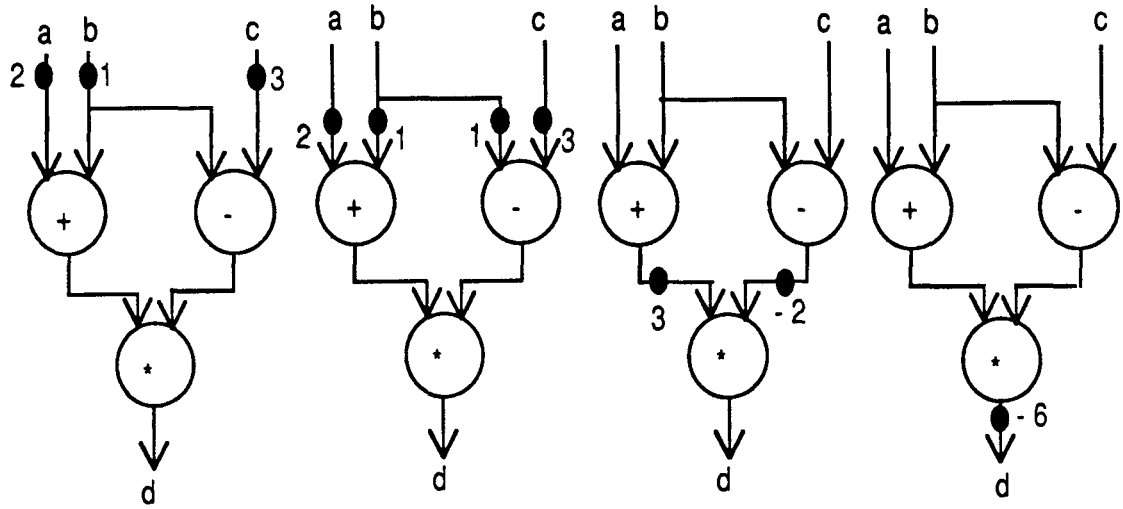


fig 2.1 : La règle de consommation pour évaluer l'expression
d = (a+b) * (b-c)

A ce moment là, les jetons présents en entrée sont enlevés de leurs arcs (consommés) et un jeton résultat est alors déposé sur l'arc de sortie du noeud. Cette règle d'exécution des instructions est appelée règle de consommation (firing rule). Comme on peut le remarquer, les machines contrôlées par les données sont caractérisées par un niveau de parallélisme à grain très fin.

Si un même résultat est utilisé par plusieurs instructions, il est calculé une seule fois, mais à ce moment plusieurs arcs de sortie existent au niveau du noeud correspondant. Par conséquent, lors de l'opération de consommation des jetons d'entrées, il y a production d'autant de jetons en sortie qu'il y a d'arcs de sortie.

Il faut noter à ce niveau que dans un graphe data-flow on peut trouver des jetons de contrôle permettant la gestion de la circulation des jetons de données sur le graphe.

Le principe de l'assignation unique est une caractéristique de base du mode de contrôle data flow. Il spécifie qu'une et une seule valeur peut être affectée à un opérande. Afin de pouvoir exécuter des boucles et permettre un partage de code entre plusieurs modules, il est nécessaire de modifier le modèle de base pour autoriser l'existence de plusieurs jetons sur un même arc, et par la même donner la possibilité à un opérande d'avoir plusieurs valeurs. Pour cela, les arcs reliant les noeuds du graphe doivent être considérés comme des files d'attente.

Pour permettre une exécution correcte des opérateurs, les jetons du graphe doivent être marqués. Cette marque spécifie en fait le contexte dans lequel les données correspondant aux jetons ont été créées. Par conséquent, un opérateur n'est exécuté que si, sur tout arc entrant, il existe un jeton qui a la même marque. L'opérateur produit alors un jeton en sortie qui a lui aussi la même marque que les jetons consommés et le dépose dans la file d'attente de l'arc de sortie. Contrairement au modèle de base, ici il n'est plus nécessaire d'avoir des arcs de sorties vides pour réaliser l'opération de consommation.

La figure 2.2 donne l'architecture typique d'une machine contrôlée par les données. Elle est composée de 4 unités de base:

1) L'unité de correspondance (matching unit) a pour fonction de rassembler les jetons qui sont à destination d'une même instruction. Lorsque tous les jetons destinés à une instruction sont présents, ils sont transmis à l'unité de mise à jour et de recherche d'instruction.

2) L'unité de mise à jour et de recherche d'instruction qui a pour fonction de construire une instruction exécutable en utilisant l'ensemble complet d'arguments transmis par l'unité de correspondance.

L'instruction destination est alors copiée et les arguments sont ensuite mis à leurs positions dans l'instruction. Celle-ci devient exécutable et peut être transmise à l'une des unités de traitement (s'il y a une unité libre).

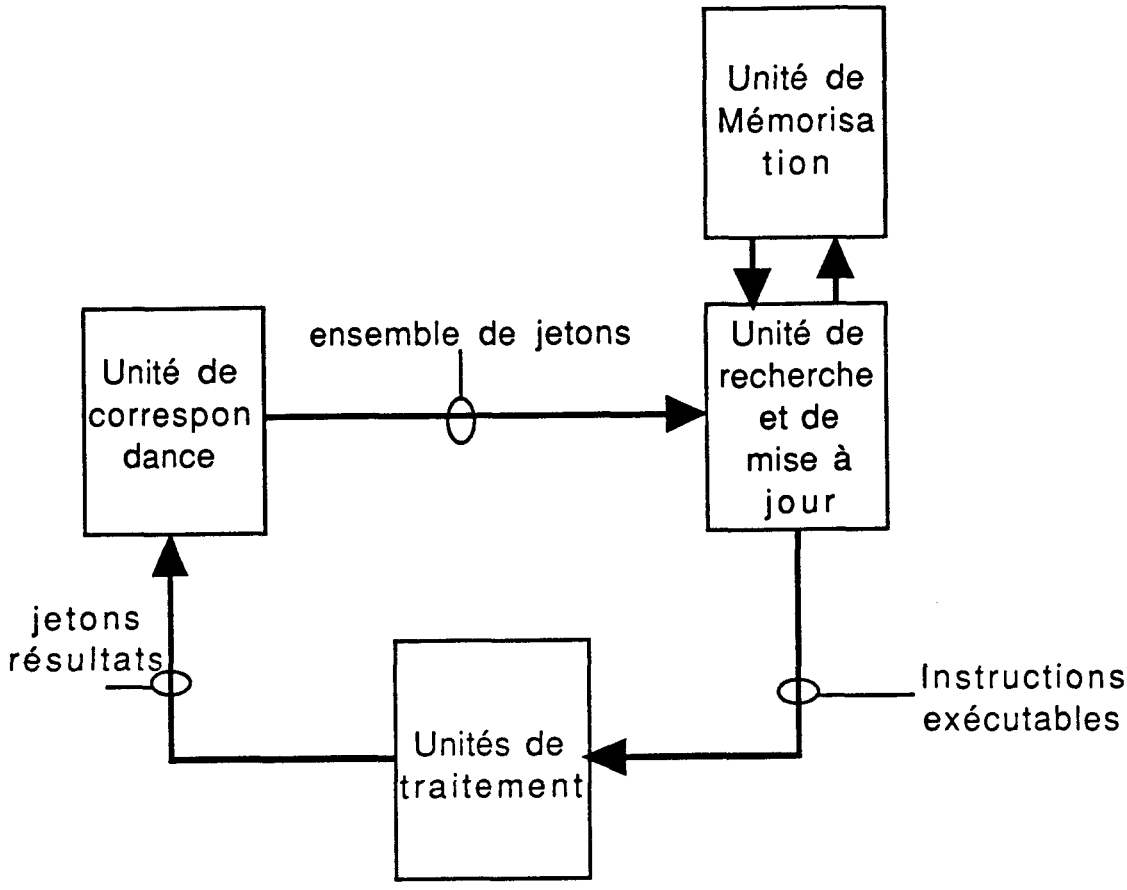


fig 2.2 : l'architecture typique d'une machine Data-Flow

3) Les unités de traitement ont pour fonction d'exécuter une instruction et de générer les jetons résultats à destination des instructions consommatrices.

4) Les unités de mémorisation qui ont pour fonction de stocker les instructions du programme à exécuter.

Dans les systèmes utilisant le mode de contrôle par les données les processeurs sont bien exploités, car ces derniers sont alloués uniquement aux processus représentant les opérateurs qui ont eu tous leurs arguments et dont le résultat peut être consommé par un autre opérateur (firing rule).

En contrepartie les inconvénients dans l'utilisation d'une architecture data flow sont les suivantes [Vegd84] :

1- Une quantité trop importante de parallélisme peut être générée durant l'évaluation du graphe. Il s'en suit que des données non utilisables seront calculées par la machine. Ces données occupent les ressources (mémoire et processeurs) de la machine inutilement. De ce fait la machine peut contenir une quantité importante de résultats

partiels qui sont soit utilisé beaucoup plus tard mais qui peuvent créer à ce moment du traitement une situation d'interblocage, ou bien ce sont des résultats qui ne seront pas utilisés du tout.

2- Dans les architectures data flow exécutant des langages fonctionnels, il n'est pas possible d'évaluer des structures de manière paresseuse (lazy evaluation). Comme on pourra le voir dans le paragraphe suivant, l'évaluation paresseuse des structures (notamment des listes) permet de manipuler des structures infinies en évaluant uniquement les éléments de la structure qui sont nécessaires au calcul. Lorsque le mode de contrôle est du type data flow ceci n'est pas possible car les noeuds du graphe consomment uniquement les données totalement évaluées. Dans le cas des structures, ceci peut conduire à une saturation des ressources de la machine.

Pour résoudre ces problèmes, plusieurs projets tentent d'introduire une évaluation paresseuse dans le modèle data flow [Wei88] [Amam87].

De façon générale, la solution consiste à enrichir le graphe de dépendance de données original par un certain nombre de noeuds et d'arcs de demandes.

II.1.2 Les architectures contrôlées par les demandes

Dans une machine dont le contrôle est du type von Neumann, les données sont considérées comme des éléments passifs sur lesquelles agissent les instructions. Dans un mode de contrôle dirigé par les données, ce sont les instructions qui sont passives. En effet ce sont les données qui déclenchent les opérations pouvant être réalisées.

Dans une machine contrôlée par les demandes, les instructions et les données sont traitées de manière identique, elles sont considérées comme des expressions.

L'application d'un opérateur (instruction) sur des arguments (données) et le résultat de cette application sont considérés alors comme deux formes différentes d'un même objet. Les instructions sont utilisées de ce fait comme un ensemble de règles de réécriture permettant de passer d'une forme à une autre. Cette phase de transformation est réalisée jusqu'à l'obtention d'une forme qui ne peut plus être transformée.

De ce fait, si on demande l'évaluation de l'expression A=B+C avec B égal à 1 et C égal à 2, cette expression est considérée équivalente à

l'expression A = 1 + 2, qui peut être aussi transformée en A = 3. Les deux expressions A = 3 et A = B + C avec B égal à 1 et C égale à 2 sont équivalent.

Dans une machine dirigée par les demandes ou machine de réduction, pour tout objet il existe une et une seule expression qui le définit à un moment donné et toutes les références à cet objet ont la même valeur. Cette caractéristique est appelée *transparence au référent* (referential transparency) [Trel82]. Elle signifie que les résultats sont indépendants de leur historique, mais dépendent uniquement de l'environnement dans lequel ils sont évalués.

Dans ce mode de contrôle, l'évaluation d'une expression n'est réalisée que si le résultat qu'elle produit est nécessaire au calcul du résultat final (i.e contrôle par nécessité).

Il existe 2 types de machines de réduction, suivant que les arguments sont transmis par référence ou par valeur:

II.1.2.1 la réduction de chaînes

Dans le modèle de réduction de chaînes, lorsque la valeur produite par une expression est nécessaire, c'est une copie de l'expression qui est transmise. Dans la figure 2.3 la valeur de l'objet "a" a été demandée. Une copie de la définition de "a" a été réalisée et on remplace le nom de l'objet à évaluer par sa définition, qui est "* i1 i2". Deux demandes sont ensuite générées pour obtenir la définition de i1 et i2. L'expression à évaluer devient alors "(* (+ b 1) (- b c))".

Sous cette forme, l'expression contient 2 noms d'objets qui ne sont pas réduits. De ce fait 2 demandes sont générées (une pour "b" et une pour "c") pour obtenir les valeurs de chacun de ces deux objets. L'objet à réduire devient par conséquent (* (+ 4 1) (- 4 2)), ensuite (*5 2), puis enfin 10, qui est la forme finale (appelée aussi forme normale) de "a".

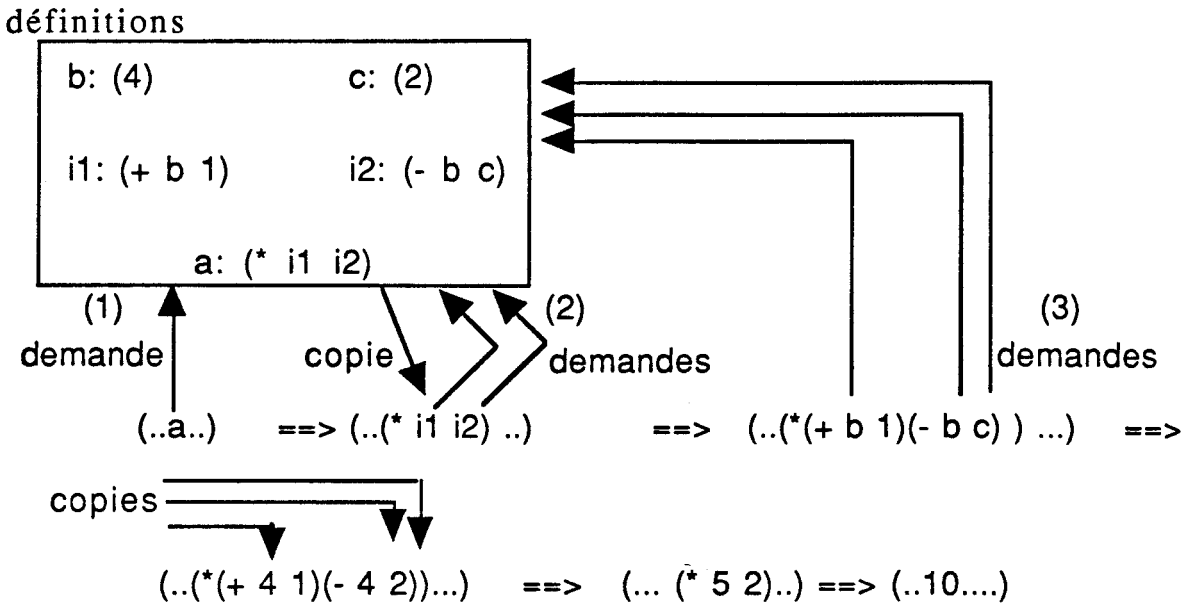


fig 2.3 : la réduction de chaînes pour l'évaluation d'une expression

Comme on peut le voir dans l'exemple précédent, si un même objet est référencé plusieurs fois, son évaluation est à chaque fois réalisée. De plus la taille de l'expression à réduire croit jusqu'à ce qu'elle devienne réductible, c'est à dire jusqu'à ce que tous les arguments soient évalués; dans l'exemple précédent ceci correspond à la forme (* (+ 4 1)(- 4 2)). Par la suite la taille décroît jusqu'à la forme normale. Ceci rend la gestion de la mémoire très difficile. La machine de MAGO [Mago80] fût l'une des premières machines proposées qui a utilisé ce mode de contrôle.

II.1.2.2 La réduction de graphes

Dans une machine dont le mode de contrôle est du type réduction de graphes, lorsque l'objet demandé contient un argument qui n'est pas évalué, c'est une référence à cet argument qui est manipulée (appel par référence).

En d'autres termes si un même argument est demandé par plusieurs expressions, il y a calcul de cette expression une et une seule fois (à la première référence). Si une nouvelle demande arrive alors que l'argument a été déjà demandé et calculé, on utilise le résultat de la première demande d'évaluation. Par conséquent, l'utilisation des

références permet le partage des arguments entre expressions définitions.

La figure 2.4 illustre l'évaluation d'une expression avec le mode de contrôle par réduction de graphes. Ici au lieu de copier les définitions de i1 et i2 dans "a", ce sont 2 références vers ces 2 définitions qui sont utilisées.

L'évaluation de "a" est suspendue pour laisser place à celles de i1 et i2. Des demandes sont alors générées par i1 et i2 pour avoir les valeurs de "b" et "c". Afin de mémoriser le fait que la valeur de "a" a été demandée par "j", un champ supplémentaire est utilisé.

Dans ce champ nous avons écrit [j/1] pour indiquer que le résultat de l'évaluation de "a" sera envoyé vers "j" et qu'il remplacera le 1er argument de j. Par la suite les évaluations de i1 et i2 sont suspendues. Comme les expressions "b" et "c" sont évaluées, leurs valeurs sont directement communiquées. Ceci permet de continuer alors l'évaluation de i1 et i2 qui deviennent égales respectivement à 5 et 2. Ces valeurs sont alors communiquées à l'expression "a", qui peut terminer son calcul, et qui se réduit à 10.

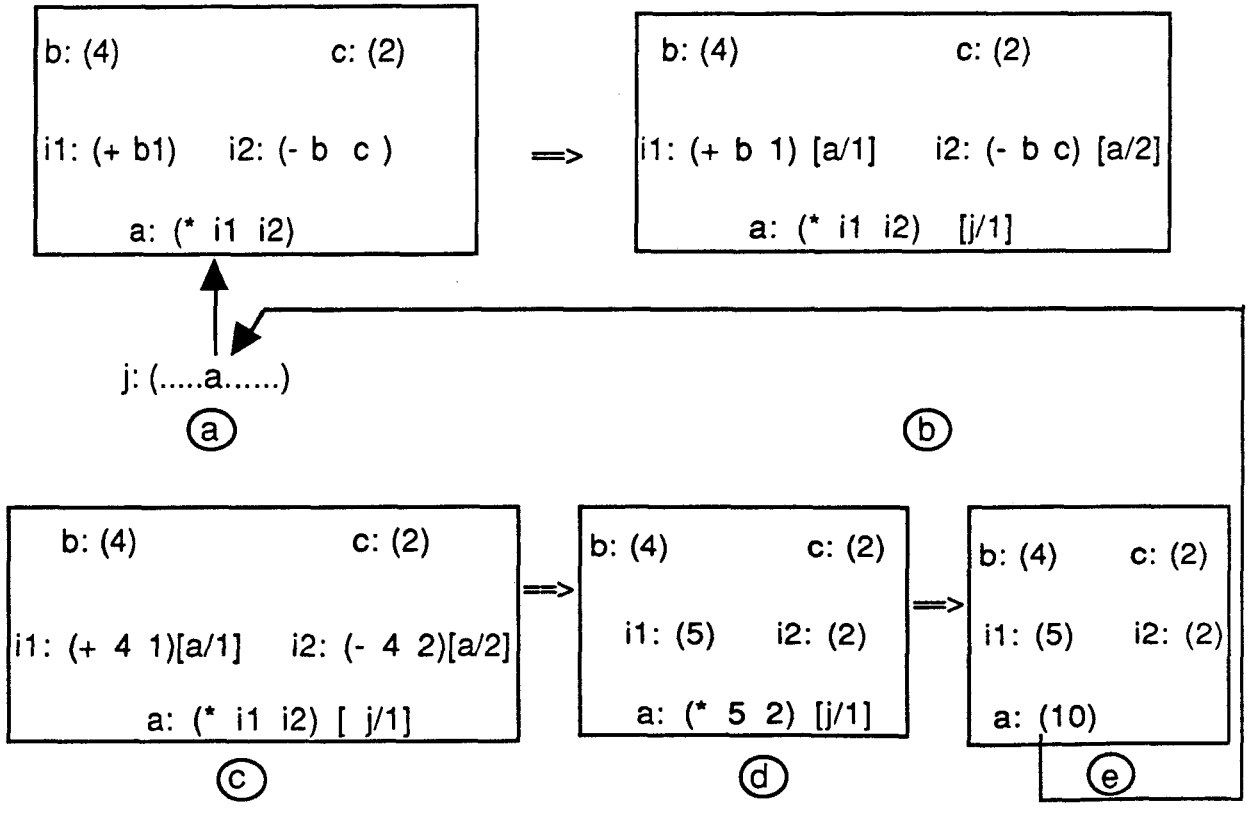


fig 2.4 : La réduction de graphes

Comme on peut le remarquer l'exécution d'un programme dans une machine à réduction de graphes se fait en 2 étapes :

-Une première étape ascendante où les demandes d'expressions sont générées et où les évaluations des expressions sont suspendues.

-Une deuxième étape descendante, où les valeurs sont retournées aux expressions qui ont fait les demandes. Dès qu'une expression a reçu tous ses arguments, son évaluation est reprise.

Par rapport au modèle data-flow, le mécanisme de réduction permet l'évaluation uniquement de ce qui est nécessaire pour le calcul.

Pour les raisons citées dans le paragraphe II.1.2.1, la réduction de graphes est plus utilisée que la réduction de chaînes dans la conception de machines parallèles.

Nous verrons dans le paragraphe II.4 des exemples d'architectures contrôlées par les demandes.

II.2 LE PARALLELISME DANS LES LANGAGES DE PROGRAMMATION

Dans le chapitre I nous avons indiqué quels étaient les 2 types de problèmes (matériel et logiciel) aux quels les futures machines devraient faire face.

Dans le paragraphe précédent nous avons passé en revue les différentes solutions pour résoudre le problème lié à l'architecture de la machine. Dans ce paragraphe, par contre, nous allons voir quelles sont les problèmes liés aux langages utilisés pour programmer ces machines parallèles. Nous proposerons ensuite une solution à ces problèmes, et nous montrerons aussi quels sont les avantages de l'approche proposée.

II.2.1 Les langages impératifs

II.2.1.1 Les langages impératifs et les architectures parallèles

Il a été noté depuis bien longtemps que les langages classiques comme PASCAL ou FORTRAN (appelés langages impératifs) sont mal adaptés à l'écriture de programmes de taille importante [Back78].

En effet, l'effort du programmeur utilisant les langages impératifs est pris par les détails de l'implémentation de sa solution sur la machine. Au lieu que le programmeur spécifie *ce qu'il faut calculer*, la majeure partie

du temps de conception est prise par la description de la manière selon laquelle la machine arrive à la solution (*comment calculer* la solution). Ceci a pour effet de rendre les programmes difficiles à comprendre et par conséquent difficiles à maintenir.

Une caractéristique que l'on regrette souvent dans les langages impératifs est que ces derniers se prètent très mal à une démonstration du programme ou une transformation automatique [Darl87]. Ceci est dû à l'existence des effets de bord dans ces langages. Un nom dans un contexte donné peut, en effet, avoir plusieurs valeurs à différents moments, suivant l'opération d'affectation qui a été dernièrement exécutée.

Lorsque le programme est conçu pour être exécuté sur une machine multiprocesseurs, la tâche du programmeur devient encore plus difficile. En effet ce dernier doit d'abord concevoir un algorithme parallèle qui réalise les spécifications voulues. Cet algorithme doit contenir un nombre élevé mais fixe d'activités (appelées tâches ou processus) afin d'occuper toutes les ressources, sans trop les surcharger.

En général, la division du programme en tâches parallèles est réalisée explicitement par le programmeur au moyen d'opérateurs de contrôle parallèle (Fork et Join, ParBegin et ParEnd, ..). Ces opérateurs permettent d'activer plusieurs tâches de manière concurrente. De ce fait le programmeur doit tenir compte des particularités de la machine sur laquelle va se faire l'exécution du programme. L'un des objectifs primordiaux des nouveaux langages, est justement d'automatiser l'extraction des tâches parallèles en exploitant le parallélisme implicite de ces programmes*.

Le programmeur écrivant un programme parallèle dans un langage impératif doit aussi utiliser un moyen de synchronisation et de

---

* Remarque:

Pour certains langages impératifs on peut trouver une autre méthode permettant l'exécution du programme de manière parallèle. Cette méthode utilise un compilateur spécialisé qui identifie automatiquement les parties du programmes pouvant être exécutées en parallèle. Cette approche est utilisée pour l'exécution par exemple de programmes FORTRAN sur des machines parallèles von Neumann SIMD. Malgré l'avantage offert par cette méthode, les performances ne sont pas à la hauteur des potentialités du parallèlisme existant dans le programme, et dans tout les cas, l'extraction du parallélisme est faite de manière statique.

communication entre les tâches. Ceci doit se faire en évitant de créer des situations d'interblocage (système utilisant le mécanisme d'émission/réception de messages) ou bien de manipuler des données erronées (système utilisant le mécanisme de zones mémoires partagées).

Dans quelques langages impératifs parallèles (comme OCCAM) le programmeur doit en plus attribuer chacune des tâches parallèles de son programme à un processeur spécifique, et il doit s'assurer aussi que deux tâches qui communiquent sont placées sur deux processeurs reliés entre eux.

Pour terminer cette liste d'inconvénients, il faut ajouter que lorsque plusieurs tâches sont exécutées par un même processeur (UC allouée en temps partagé), le programmeur doit s'assurer que le résultat de l'exécution est correct quelque soit l'ordre dans lequel sont exécutées les tâches concurrentes par le processeur (politique de l'allocateur).

II.2.1.2 Conclusion

C'est donc pour résoudre tous ces problèmes que les chercheurs ont commencé à concevoir d'autres types de langages. Parmi ces derniers ceux qui offrent de très bonnes possibilités pour une exécution parallèle sont les langages déclaratifs.

Dans ce type de langages la tâche du programmeur n'est pas de montrer comment implémenter son algorithme sur la machine, mais uniquement d'introduire son algorithme en déclarant ce que l'on désire calculer.

L'ensemble des langages déclaratifs peut être divisé en 2 familles. Nous pouvons distinguer les *langages fonctionnels* (basé sur le lambda calcul), comme les langages FP, SASL, KRC, ML, ou HOPE. De l'autre côté il y a les *langages de la programmation logique* (que nous appelerons pour simplifier les langages logiques), comme PROLOG, PARLOG...etc. Les langages logiques sont basés sur le calcul des prédicats du premier ordre.

Ces deux familles ont pour caractéristiques communes, l'absence de la notion de séquencement (du compteur ordinal), de la notion de cellule mémoire, et de directive de saut (comme la commande GOTO).

Dans la suite de ce paragraphe nous allons introduire chacune de ces 2 familles. On donnera aussi quelques exemples de programmes lorsque ceci est nécessaire.

II.2.2 Les langages fonctionnels [Peyt86][Hend80]

II.2.2.1 introduction

Un programme fonctionnel est considéré comme une fonction mathématique, qui appliquée à une donnée (entrée) fourni un résultat (sortie).

On considère ainsi que le résultat de l'exécution d'un programme fonctionnel et le couple programme-arguments d'entrée ne sont que 2 formes différentes d'un même objet.

Afin de détailler la manière selon laquelle un langage fonctionnel peut être exécuté, nous allons utiliser le mode de contrôle par réduction de graphes. Ce mode de commande est largement utilisé pour concevoir des machines parallèles exécutant un langage fonctionnel. Ce choix permet, en effet d'exploiter la machine de manière efficace [Cous88].

Considérons la fonction qui à x associe (x+1)*(x-1), soit f cette fonction. Considérons aussi que la valeur entière 4 est l'argument d'entrée de cette fonction.

L'expression initiale est donc "f 4" qui peut s'écrire sous les formes suivantes :

```
f 4  -> (4 + 1) * (4-1)
     -> 5 * 2
     -> 10
```

Comme on peut le voir l'exécution d'un programme fonctionnel consiste à transformer l'expression initiale en utilisant les définition des fonctions du programme. Ces dernières peuvent être des fonctions définies par l'utilisateur ou bien des fonctions prédéfinies, c'est à dire connues par l'interpréteur qui va exécuter le code, comme c'est le cas pour les fonctions {+, -, *} de l'exemple précédent.

II.2.2.2 Exécution parallèle d'un programme fonctionnel

Comme exemple de programme fonctionnel nous donnons le programme qui réalise le calcul de la factorielle d'un entier n.

```
Par_Fact n = Dec 1 n
```

```
Dec min max = If (min = max) Then min
                  Else (Dec min mid) * (Dec (mid + 1) max)
Where mid = (min + max) IntDiv 2
```

Dans cet exemple nous utilisons le mot clé Where qui permet de définir des variables locales. L'opérateur IntDiv réalise la division entière de 2 opérandes. Comme on peut le voir le calcul de la factorielle se fait en utilisant la fonction Dec (Diviser et conquérir), qui divise l'intervalle [1, n] en 2 parties, puis calcule le produit de tous les entiers dans chacun de ces intervalles, toujours en divisant les intervalles en intervalles plus petits.

Regardons de plus près comment se fait le calcul de la factorielle de 4.

```
1)  Par_Fact  4 = Dec 1 4
2)  = (Dec 1 2) * (Dec 3 4))
3)  = ( (Dec 1 1 ) * (Dec 2 2) ) * ( (Dec 3 3) * (Dec 4 4 ) )
4)  = ( (1 * 2) * (3 * 4) )
5)  = 2 * 12
6)  = 24
```

Le passage de la forme 2 à la forme 3 peut se faire en calculant Dec 1 2 puis Dec 3 4 ou l'inverse. Sur une machine multiprocesseurs on peut tenter de réaliser les 2 transformations simultanément. Ainsi l'existence de plusieurs expressions à réduire peut donner lieu à un traitement parallèle d'un programme fonctionnel

Dans une machine à réduction de graphes le programme fonctionnel à exécuter (ou l'expression à évaluer) est représenté par un graphe orienté. Ce graphe subit des changements au fur et à mesure que l'on avance dans le traitement.

La figure 2.5 montre les 6 graphes de l'exécution du programme factorielle sur l'entier 4. Ces 6 graphes correspondent aux 6 étapes dans l'évaluation de Par_Fact 4. Nous pouvons donc dire que l'exécution d'un programme fonctionnel consiste à parcourir le graphe en cherchant un sous graphe pouvant être réduit, c'est à dire un sous graphe dont la racine correspond soit à une fonction définie dans le programme ou bien un opérateur primitif dont les arguments sont évalués (voir plus loin l'évaluation paresseuse). Ce sous graphe est appelé *redex* (pour "reducible expression").

Une fois ce redex trouvé, il suffit de le remplacer par le résultat de l'application de la fonction. Cette étape est réalisée jusqu'à ce que le graphe initial soit transformé en une valeur atomique, c'est à dire un graphe où aucune transformation ne peut être appliquée.

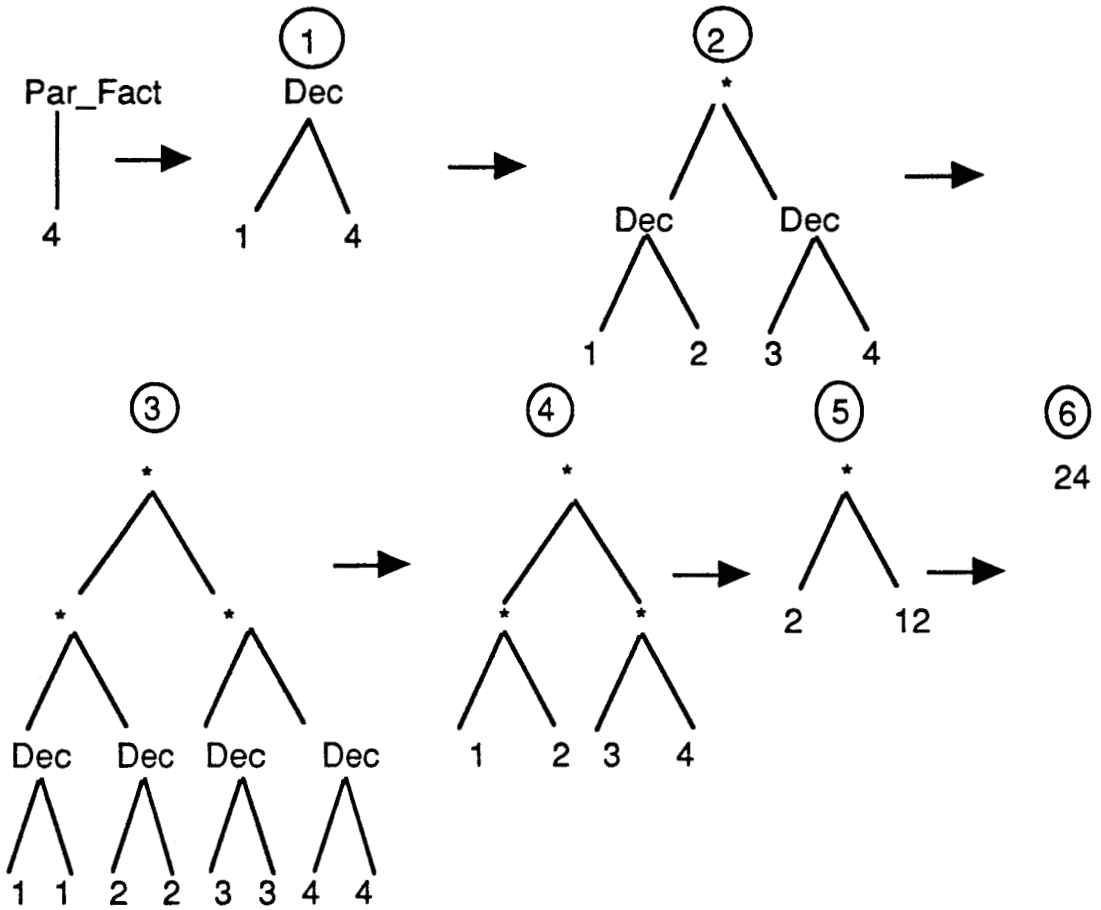


fig 2.5 : Les 6 graphes correspondants aux 6 étapes de réduction de Par_Fact 4

II.2.2.3 Quelques améliorations

Comme on peut le remarquer, rien n'interdit de choisir plusieurs redex en vue d'une réduction parallèle. Ceci est d'autant plus intéressant que les sous graphes candidats sont indépendants et n'exigent aucune synchronisation. De plus l'ordre dans lequel se font ces réductions n'a pas d'effet sur le résultat, et les communications entre les redex sont réalisées à travers les arcs du graphe.

Malheureusement cette technique, qui consiste à créer une tâche parallèle pour chaque redex peut produire un très grand nombre de tâches parallèles, ce qui détériore les performances du système [Peyt89].

En effet le nombre de redex à réduire (qui est le nombre de tâches parallèles) peut être tellement grand que l'on risque de saturer le système. Parfois même de manière inutile, comme c'est le cas de la fonction suivante:

```
f x = If x > 0 Then x + 1
              Else f (x - 1)
```

Ainsi si on tente l'évaluation de "f 0", en autorisant l'évaluation des 3 parties du "If" simultanément nous avons un traitement infini (fig 2.6).

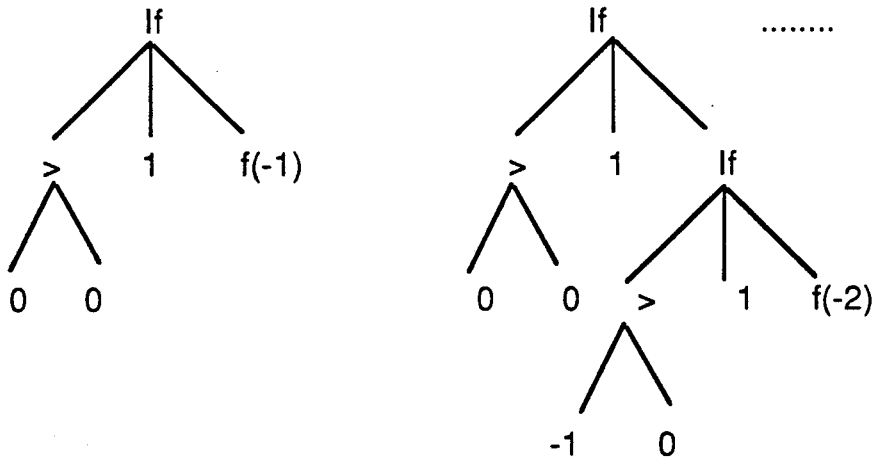


fig 2.6 : un traitement infini pour l'évaluation d'une alternative

Ce problème est généralement résolu en partie en ne permettant l'évaluation parallèle que des redex qui correspondent à des arguments d'opérateurs dit *stricts*. Un opérateur est strict si l'évaluation de tous ses arguments est nécessaire avant celle de l'opérateur même. Les opérateurs arithmétiques {+, -, *, ..etc} sont stricts, et par conséquent leurs évaluation va engendrer des tâches parallèles.

Un opérateur qui n'est pas strict est dit paresseux. Un opérateur paresseux peut être strict pour un (ou une partie) de ses arguments uniquement. C'est ainsi que l'opérateur If est strict uniquement pour son premier argument.

Pour les programmes qui utilisent peu d'opérateurs arithmétiques (comme les programmes de tri) , le nombre de tâches parallèles est réduit.

De façon générale si on considère l'application d'une fonction F à un argument E, il est intéressant de permettre la création d'une tâche parallèle pour évaluer E si l'application de F sur E exige la valeur de E. Plusieurs chercheurs proposent pour cela de réaliser *une analyse de striticité* du programme au moment de la compilation (strictness analysis). Cette analyse consiste à extraire pour chaque fonction du programme quels sont le ou les arguments pouvant être évalués de manière parallèle [Peyt86].

Même lorsque l'opérateur est strict, il n'est intéressant de créer une nouvelle tâche parallèle pour l'évaluation de l'argument, que si le temps nécessaire à cette évaluation est important par rapport au temps de communication et d'initialisation de la tâche. Dans le cas contraire, l'exécution parallèle risque d'augmenter le temps d'exécution.

La solution la plus souvent proposée pour résoudre ce problème est de permettre à l'utilisateur d'ajouter des indications (ou annotations) pour déterminer les tâches qui méritent d'être évaluées en parallèle.

A titre d'exemple, reprenons le programme fonctionnel calculant la factorielle, dans lequel nous ajoutons une annotation :

```
Par_Fact x  =  Dec 1 x
Dec min max  =   If (min = max) Then  min
          Else  (Dec  min  mid) *  {Tâche_par}  (Dec (mid + 1) max))
Where  mid = (min  +  max)  IntDiv 2
And      Tâche_par = (max - min) > 5
```

L'indicateur {Tâche_par} est utilisé pour indiquer qu'il est intéressant de créer une nouvelle tâche pour le second argument de l'opérateur * si la valeur de Tâche_par est vraie, c'est à dire lorsque intervalle [min, max] est suffisamment grand. On trouvera dans [Burt87] une étude détaillée de cette approche.

II.2.2.4 L'évaluation paresseuse

Nous avons précédemment indiqué que l'un des avantages offerts par le mode de contrôle par réduction de graphes était la possibilité de manipuler des structures infinies dans les langages fonctionnels.

L'exemple suivant illustre cette possibilité:

Double_Int x = [2 * First (Int x)] : [Double_Int (Rest (Int x))]

Int x = x : Int (x + 1)

La fonction Int x permet de construire la liste infinie d'entiers [x, x+1, x+2, ...]. Le symbole ":" réalise la construction d'une liste (équivalent au cons du Lisp).

La fonction Double_Int consomme cette liste en multipliant par 2 les éléments de celle-ci. L'appel de Double_Int 1 permet ainsi de construire la liste [2, 4, 6, 8,....]. De ce fait si on demande le troisième élément de cette liste, c'est à dire First(Rest(Rest(Double_Int 1))) et si on considère que l'opérateur ":" est strict , on aura un calcul infini pour l'évaluation de Int 1 (afin d'avoir la liste infinie [1, 2, 3, ...].

L'utilisation de l'évaluation paresseuse permet à la fonction Int de retourner des pointeurs, au lieu de retourner une liste totalement évaluée. Ceci se fait on considérant que l'opérateur ":" est paresseux, c'est à dire que son évaluation peut être réalisée même si les 2 arguments ne sont pas complètement réduits. Le résultat de son évaluation est tout simplement un pointeur vers la tête de liste (c'est dire First x) et un pointeur vers la queue de la liste (c'est à dire Rest x).

Comme l'opérateur * est strict, son évaluation force le calcul de First x à chaque appel. Par conséquent l'évaluation de First (Rest (Rest (Double_Int 1))) donne :

First (Rest (Rest (Double_Int (Int 1))))

= First (Rest (Rest (2*1 : Double_Int (Rest(Int 1)))))

= First (Rest (Double_Int (Rest (Int 1))))

= First (Rest (2*2 : Double_Int (Rest (Int 2))))

= First (Double_Int (Rest (Int 2)))

= First (Double_Int (Int (3)))

= First (3*2 : Double_Int (Rest (Int 3)))

= 6

Dans une machine multiprocesseurs l'évaluation paresseuse conduit à un traitement séquentiel des éléments de la liste. En effet si nous reprenons l'exemple précédent, nous pouvons noter que les deux fonctions Double_Int et Int fonctionnent de manière séquentielle, car les éléments de la liste sont évalués un par un.

L'une des solutions à ce problème, est de permettre à la fonction Int de poursuivre la construction du reste de la liste infinie après avoir retourné le résultat intermédiaire à la fonction Double_Int (résultat représenté sous forme de 2 pointeurs). De cette manière, la fonction Double_Int accède à la liste élément par élément, alors que la fonction Int achève la construction. Cette approche est appelée *parallélisme spéculatif* [Burt87].

II.2.2.4 Les fonctions de haut niveau

Une autre caractéristique intéressante dans les langages fonctionnels est la possibilité de manipuler des fonctions de la même manière que des valeurs atomiques. Une fonction qui accepte des fonctions comme arguments, les combine pour former de nouvelles fonctions est appelée forme fonctionnelle ou fonction de haut niveau.

A titre d'exemple, nous pouvons définir une fonction "map" qui a comme argument une fonction f et dont le résultat est une fonction qui lorsqu'elle est appliquée à une liste [e1, e2, e3, ...] donne la liste [f(e1), f(e2), f(e3), ...]. On peut définir cette fonction de cette manière :

```
map f  =  λ(x)  If  ( x = [] )    Then []
          Else  f(First x)  :  ( (map f) (Rest x))
```

Dans cette exemple nous avons introduit le symbole λ(x) pour indiquer que le résultat de l'application de la fonction map sur une fonction f donne une fonction utilisant comme paramètre une liste x.

Avant de terminer ce paragraphe, il faudrait noter que le programme calculant la factorielle que nous avons donné dans le paragraphe précédent peut aussi s'écrire sous une autre forme :

```
Seq_Fact x  =  If (x = 0) Then 1
               Else  x * Seq_Fact (x -1)
```

Comme on peut le remarquer contrairement à la version Par_Fact, cette dernière version, à cause de la dépendance de données ne permet pas une évaluation parallèle de plusieurs redex.

La remarque inverse peut être faite pour la fonction map, où au lieu d'appliquer la fonction f séquentiellement sur les éléments de la liste, on peut définir une fonction map qui s'applique de manière concurrente sur tous les éléments de la liste [Deve86].

II.2.3 Les langages de la programmation logique

II.2.3.1 Introduction

Les langages de la programmation logique comme PROLOG sont basés sur le calcul des prédicats du premier ordre. Un programme écrit dans un tel langage est composé d'un ensemble de relations. Ces relations sont appelées clauses, et chaque clause a la forme:

P <- B1, B2, B3, ...Bn n>0

Cette écriture signifie que le prédicat P est prouvé si on prouve B1 et B2....et Bn.

Un prédicat Bi peut avoir la forme bi(x1, x2, .., xn). Les xi sont alors appelés les termes du prédicats.

L'ensemble des clauses comportant le même prédicat de tête de clause constitue la définition de ce prédicat, et chaque clause est appelée alors clause alternative.

De façon générale une question est spécifiée par une suite de buts à résoudre :

<- L1, L2, L3, ...Ln

qui consiste à prouver L1 et L2 ...etc.

II.2.3.2 Exécution d'un programme PROLOG

Dans l'exécution d'un programme logique les 2 opérations de base sont :

1- L'unification : Cette phase consiste à déterminer les instances communes des termes de 2 prédicats ayant le même nom (ou symbole). Elle se fait en remplaçant les variables libres de l'un par des termes de l'autre.

2- La substitution : Ici il s'agit de remplacer le but qui s'est unifié avec une clause par le corps de celle-ci.

La séparation, dans l'exécution d'un programme PROLOG, entre la logique et le contrôle facilite énormément une exécution parallèle. En effet cette séparation permet de changer de mode de contrôle (passer du séquentiel au parallèle) tout en gardant la logique du programme.

D'après Conery [Cone86], une exécution parallèle d'un programme logique est possible de 3 manières:

1- Le parallélisme OU: Il correspond à l'exécution simultanée de toutes les clauses alternatives de la définition pour un but donné. La suite du traitement peut être faite soit en considérant toutes les solutions obtenues, ou bien en choisissant au hasard une solution (cas des langages gardés).

Le principal problème rencontré dans les machines utilisant ce type de parallélisme est l'augmentation rapide du nombre de buts à résoudre (croissance rapide de la charge).

2- Le parallélisme ET : Si le but à prouver est composé de plusieurs prédicats, le parallélisme ET consiste à évaluer simultanément chacun des prédicats dans la suite des buts. Lorsque 2 prédicats possèdent une variable commune, il se pose le problème de cohérence entre les termes affectés à cette variable.

Plusieurs méthodes ont été concues pour éviter ce problème de conflit.

a: La méthode la plus simple mais aussi la moins performante consiste à calculer pour chaque prédicat l'ensemble des solutions possibles, puis d'utiliser des opérations de jointures entre ces ensembles de façon à avoir un ensemble cohérent de solutions.

b: La deuxième méthode utilise une annotation des variables dans la suite des buts à exécuter (comme dans Concurrent Prolog). De cette façon le programmeur désigne un prédicat comme étant le producteur d'une variable donnée. Seul ce producteur peut réaliser une instanciation sur cette variable. Tous les autres prédicats utilisant la variable doivent être mis en attente (processus consommateurs), jusqu'à ce que le prédicat producteur ait été évalué et la variable instanciée.

c: Dans la troisième méthode, un algorithme d'ordonnancement des buts est employé pour sélectionner un prédicat comme étant le producteur, et cela pour chaque variable apparaissant dans la chaîne des buts. L'algorithme d'ordonnancement calcule un graphe de dépendance

de données entre les prédicats. Ce graphe est calculé à l'exécution, impliquant ainsi une charge supplémentaire pour le système.

3- Le parallélisme à l'unification : ce parallélisme consiste à réaliser les unifications simultanées des différents arguments de la tête de clause avec ceux du but. Ceci doit se faire en prenant garde au problème d'inconsistance des liens comme dans l'unification entre le but P(X, Z, X) et la clause P (a, b, c) <- .... .

Comme on peut le remarquer les possibilités d'exécution parallèle pour la programmation logique ne manquent pas. De plus leur combinaison est possible.

Pour cette raison, nous assistons aujourd'hui à un nombre de plus en plus grand de projets dont le but est de réaliser une machine parallèle pour la programmation logique. On trouvera dans [Niar86] une introduction aux différentes architectures pour exploiter le parallélisme dans les programmes logiques.

II.3 INTEGRATION DES LANGAGES FONCTIONNELS ET LOGIQUES

Récemment un grand nombre de travaux ont été menés afin d'intégrer les deux classes de langages déclaratifs, à savoir les langages fonctionnels et les langages logiques, dans un même modèle de traitement.

Malgré la nature déclarative de chacune de ces 2 familles de langages, il y a des différences fondamentales. Il est, de ce fait, intéressant d'intégrer dans une même machine ces 2 types de langages afin d'offrir au programmeur les avantages de l'un et de l'autre des langages.

On peut résumer les différences dans les points suivants [Bell86]:

1- La notation : Les langages logiques permettent la définition de relations. Néanmoins quelques problèmes peuvent être décrits de manière naturelle sous formes de fonctions. Il serait, donc, bénéfique pour le programmeur d'avoir les 2 possibilités de notations.

2- La plupart des langages fonctionnels permettent l'utilisation de fonctions de haut niveau, alors que les langages logiques actuels utilisent les prédicats de premier ordre. L'utilisation de fonctions de haut niveau augmente la puissance du langage, et permet d'avoir des programmes courts et lisibles. L'évaluation paresseuse et le typage des variables sont

aussi des caractéristiques intéressantes existant dans les langages fonctionnels mais absentes dans les langages logiques.

3- Le sens des variables: Une clause dans un programme logique ne suppose rien sur le sens des variables du prédicat de tête. De ce fait la substitution des variables, durant l'opération d'unification, peut être faite soit en *entrée* (liaison des variables de la tête de la clause à des termes du but) ou en *sortie* (liaison des variables du but à des termes de la tête de clause).

A titre d'exemple nous donnons le programme Prolog qui réalise la concaténation de 2 listes.

Append (nil, L, L) <-

Append (x::L1, L2, x::L3) <- Append (L1, L2, L3)

On peut soit utiliser ce programme de manière fonctionnelle pour concaténer 2 listes en posant le but Append ([1, 2], [3, 4], L) et qui donne comme résultat L égal à [1, 2, 3, 4]. La liaison de L1 et L2 est faite en entrée alors que celle de L est en sortie.

On peut aussi demander la suite des listes L1, L2 dont la concaténation donne la liste [1, 2, 3, 4] en posant le but Append (L1, L2, [1, 2, 3, 4]). Le résultat de l'exécution du programme Prolog donne comme résultat pour L1, L2:

[], [1, 2, 3, 4]

[1], [2, 3, 4]

[1, 2], [3, 4]

[1, 2, 3], [4]

[1, 2, 3, 4], []

Dans ce cas L1 et L2 sont liées en sortie et L est liée en entrée.

A l'inverse dans un programme fonctionnel une fois qu'il y a correspondance entre un redex et le membre gauche de la définition d'une fonction, les liaisons se font toujours en entrée, c'est à dire du redex vers la fonction. En d'autres termes, pour avoir les possibilités d'exécutions du programme précédent, il faut définir deux programmes fonctionnels différents.

De ce fait les langages logiques sont plus expressifs, car un programme logique peut représenter plusieurs programmes fonctionnels.

4- Un autre avantage des langages logiques par rapport aux langages fonctionnels est que le résultat produit par l'exécution d'un programme logique peut contenir des variables non instanciées (Non ground

variables). C'est à dire que le résultat peut ne pas être totalement évalué. Si nous prenons à titre d'exemple le programme qui calcule la longueur d'une liste.

Length (Nil, 0) <-

Length (x::L, Nl) <- Length (L, n), Plus (n, 1, Nl)

On peut très bien poser le but Length (L, 2). Le résultat produit a la forme e1:: (e2:: Nil), qui est la forme générale (ou squelette) de toutes listes de longueur 2.

Les langages fonctionnels n'ont pas cette possibilité. En effet le résultat de l'application d'un programme fonctionnel sur un argument donne toujours un argument qui est soit une constante ou bien un constructeur (liste ou tout autre type de données structurées).

Afin d'offrir à l'utilisateur l'ensemble de ces caractéristiques, plusieurs approches sont étudiées:

1- Soit en définissant un langage de nature fonctionnelle augmenté de possibilités pour réaliser l'unification.

2- Soit de définir un langage logique permettant de définir des fonctions et la manipulation des fonctions de haut niveau.

3- Ou bien alors intégrer les 2 langages dans un seul environnement de programmation.

II.4 PRESENTATION DE QUELQUES MACHINES PARALLELES POUR LA PROGRAMMATION DECLARATIVE

Nous allons dans ce paragraphe présenter de manière succincte quelques machines dédiées à l'exécution de programmes déclaratifs. Nous allons nous borner dans cette étude uniquement aux machines parallèles utilisant un mode de contrôle non-von Neumann.

II.4.1 La machine à combinateurs MaRS [Cous88]

II.4.1.1 Introduction

La machine MaRS (Machine à Réduction Symbolique) est une machine multiprocesseur à contrôle distribué et dédiée pour l'exécution de programmes fonctionnels (initialement Lisp). Cette machine est développée au CERT à TOULOUSE. Le modèle d'exécution retenue pour

cette machine est la réduction de graphes par utilisation des combinateurs.

II.4.1.2 Les combinateurs

L'utilisation des combinateurs permet de transformer un programme écrit en langage fonctionnel (exprimé sous forme de λ-expressions) en un programme ne contenant aucune variable liée.

L'élimination des occurrences de variables liées par l'utilisation des combinateurs permet l'exécution du code sans avoir à gérer des environnements. Les travaux de Turner [Turn79] furent les premiers dans le domaine de l'application des combinateurs pour l'exécution des langages fonctionnels. Turner proposa alors 3 combinateurs de base (S, K et I) définis de la manière suivante :

$\lambda x\ (E1\ E2) \Rightarrow S\ (\lambda x\ E1)\ (\lambda x\ E2)$

$\lambda x\ x \Rightarrow I$

$\lambda x\ y \Rightarrow K\ y$

où E1 et E2 sont des λ-expressions, x et y sont des variables.

Afin de montrer sur un exemple simple comment ces combinateurs peuvent être utilisés, nous donnons ci-après la transformation de la fonction Inc x définie de la manière suivante:

Inc x = 1 + x et qui peut s'écrire aussi sous la forme

Inc = $\lambda x\ (x + 1)$

Inc = S $(\lambda x\ (+\ 1\ ))\ (\lambda x\ x)$

Inc = S (S $((\lambda x\ +)(\lambda x\ 1))\ )\ (\lambda x\ x)$

Inc = S (S (K +)(K 1)) I

Turner propose de compiler le code initial en code contenant uniquement des combinateurs et des constantes. L'exécution de ce code par la machine revient à construire l'arbre de réduction et ensuite à déclencher les opérations de réduction. De cette manière les combinateurs peuvent être considérés comme un moyen permettant de mettre les variables à leur position dans l'expression à réduire.

Bien que Turner ait proposé son jeu de combinateurs pour une machine séquentielle, un très grand nombre de chercheurs ont utilisés cette idée pour implémenter un langage fonctionnel aussi bien sur des machines séquentielles, comme la machine abstraite CAM [Cousi86] , que

sur des machines parallèles comme la machine MaRS, la machine FLAGSHIP [Wats87] ou la machine GRIP [Peyt87].

II.4.1.3 Implémentation de la machine MaRS

Dans la machine Mars, les concepteurs ont défini un jeu de combinateurs plus important que celui de Turner [Cast86]. Ces combinateurs permettent de manipuler de manière simple des fonctions à plusieurs arguments sans que cela influe de manière négative sur le code généré.

Afin de contrôler le nombre de processus parallèles de façon à ne pas saturer la machine, par défaut, les paramètres des fonctions sont transmis par référence. Néanmoins, l'analyse de stricticité réalisée par le compilateur, et les annotations fournies par le programmeur peuvent transformer cet appel en appel par valeur, ce qui permet d'augmenter le taux de parallélisme.

Physiquement la machine MaRS est composée de 4 types d'éléments:

1) Les processeurs de réduction, dont le rôle est d'exécuter le code d'un combinateur (ce code est microprogrammé).

2) Les processeurs mémoire, dont le rôle est de stocker le graphe représentant le code à exécuter, de faire des opérations de ramasse-miettes, utiles lorsque des cellules mémoires se trouve libérées et enfin de générer des paquets de contrôle afin d'harmoniser les charges entre les différents processeurs mémoires.

3) Les processeurs de communication, qui sont des éléments responsables du routage des messages entre les différents processeurs.

4) Les ports d'entrée/sortie. Ces unités réalisent les échanges de données avec l'extérieur.

La figure 2.7 représente un petit module contenant 4 processeurs de réduction (R) , 8 processeurs de communication (C) , 3 processeurs de mémorisation (M) et 1 port d'E/S (E). Sur cette figure nous avons aussi indiqué le sens de transit des paquets contrôle et de données.

Dans le réseau 1, ces paquets sont ceux transmis par les processeurs de réduction vers les processeurs mémoires, afin de connaître les processeurs de réduction libres (donc ceux qui pourraient recevoir de la charge).

Dans le réseau 2, les paquets de contrôle qui y circulent, sont des paquets générés par les processeurs mémoire afin d'informer les

processeurs de réductions de l'espace disponible dans chaque module mémoire. Cette information est ensuite utilisée par les processeurs de réduction pour expédier les nouveaux paquets de données générés.

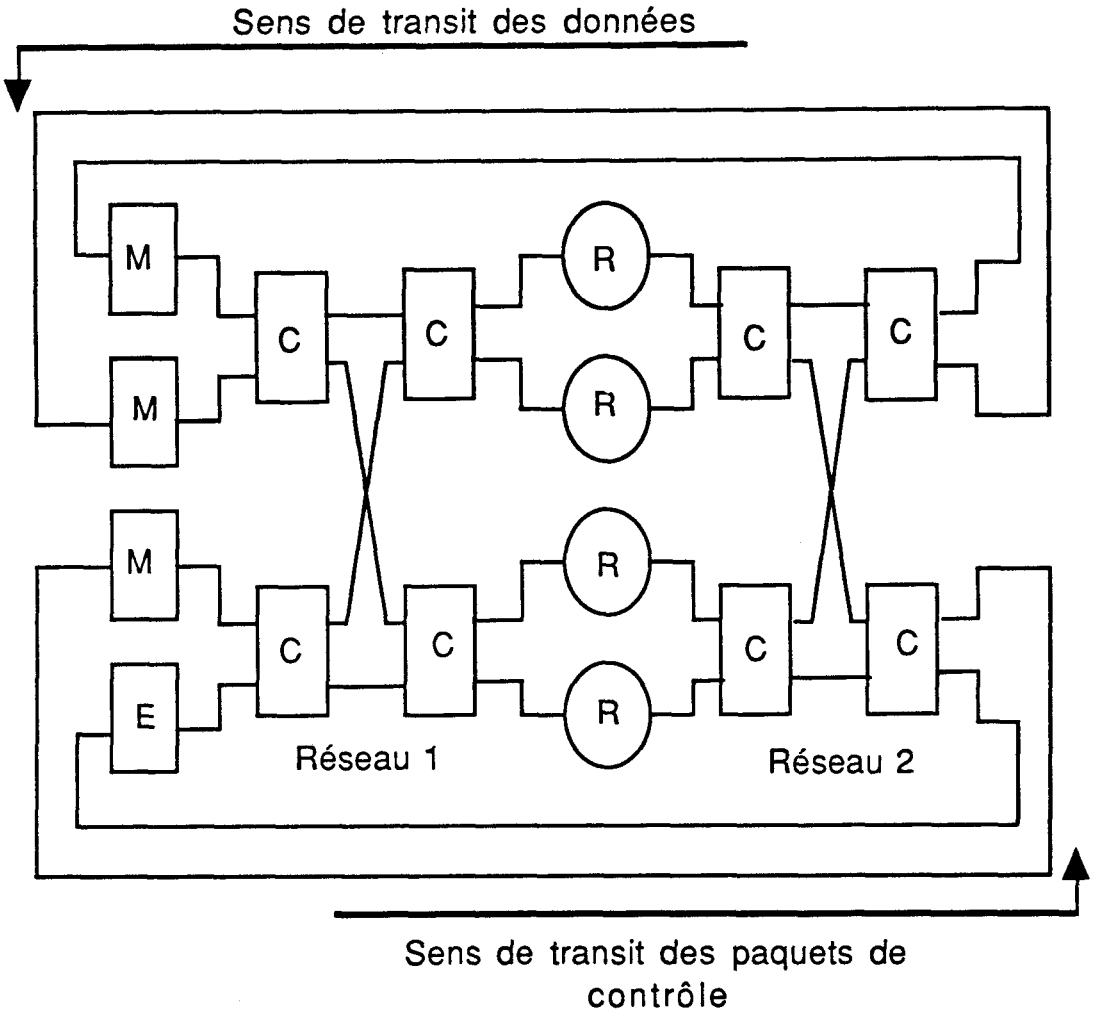


fig 2.7 : Architecture de la machine MaRS

Dans le sens inverse des paquets de contrôle et dans le réseau 1, circulent les paquets données en provenance des mémoires ou des organes d'E/S, alors que dans le réseau 2 nous trouvons les paquets contenant les résultats de calcul.

II.4.1.4 Conclusion

Un prototype de la machine MaRS a été réalisé. Ce prototype contient 32 processeurs de réductions, 24 processeurs de mémorisation, 8

processeurs d'E/S et 5 étages de 16 processeurs de communication chacun. Ces derniers ont été configurés en réseau OMEGA.

D'autres études sont menées afin d'implèmenter chacun des processeurs précédent sur un circuit VLSI.

Actuellement des travaux sont menées afin de permettre la mise au point d'un environnement de programmation et d'un compilateur optimisé (réalisant l'analyse de striticité). Les tests finaux et intégration de l'ensemble du système devront avoir lieu en 1990.

II.4.2 La machine de réduction ALICE [Darl81]

II.4.2.1 Introduction

Le projet ALICE (Applicative Language Idealized Computing Engine) développée à l' "Imperial College" de LONDRES, est l'un des premiers à avoir été lancé pour la conception d'une machine multiprocesseurs dédiée à l'exécution de programmes déclaratifs [Darl81]. Là aussi le modèle d'exécution est la réduction de graphes, mais sans utilisation des combinateurs. Le code exécuté par la machine est sous forme de λ_expressions.

L'une des caractéristiques intéressantes du projet est que l'étude de l'intégration d'un langage logique et d'un langage fonctionnel a été approfondie. Un autre point intéressant aussi, que l'on développera dans les paragraphes suivants est l'utilisation des microprocesseurs Transputers pour la réalisation d'un prototype de la machine.

II.4.2.2 Le modèle d'exécution

Dans la machine ALICE l'expression à réduire est représentée sous forme d'un graphe orienté. Ce graphe contient 2 types de noeuds :

- Les opérateurs prédéfinis et les noms de fonctions définies dans le programme.
- Les constantes et les constructeurs de type données.

Nous allons voir un exemple de programme, écrit en langage fonctionnel Hope, qui réalise le calcul de la factorielle par la méthode diviser et conquérir.

```
Factorial : Integer -> Integer
Factorial (n) <= FactB (0, n)

FactB : Integer ; Integer -> Integer
FactB (i; i) <= i
FactB (i, i+1) <= i+1
FactB (i, j) <= FactB (i, mid) * FactB (mid, j)
Where mid = IntegerDivide (i+j, 2)
```

Dans le langage Hope chaque définition de fonction est précédée du type des arguments d'entrée et de sortie. L'utilisateur peut, par le biais des types et des constructeurs existant, former de nouveaux types de données.

Lorsque que l'on demande l'évaluation de l'expression Factorial (5), il y a une opération de correspondance de modèle (pattern matching) entre Factorial (5) et la définition de la fonction Factorial. L'expression à réduire devient FactB (0, 5). Celle-ci est utilisée pour faire une autre opération de correspondance avec la définition de la fonction FactB. Ce qui donne FactB(0, 2) * FactB(3, 5). Cette opération de correspondance de modèle et de remplacement de l'expression à réduire par le membre droit de la fonction est réalisée jusqu'à l'obtention du résultat final.

La figure 2.8 illustre le mécanisme de réduction dans la machine ALICE.

Pratiquement chaque noeud du graphe de réduction est représenté par un paquet. Ce dernier contient les champs suivants:

Identificateur	Fonction/type	Liste des arguments ou constante	contrôle

- Le champ identificateur contient un nom unique pour chaque noeud. Ce nom permet de référencer le paquet.
- Le champ fonction représente le nom de la fonction à appliquer sur les arguments. Ce champ peut aussi représenter le type de la donnée s'il s'agit d'une constante ou d'un constructeur.
- La liste des arguments: cette liste contient soit les identificateurs des paquets arguments de la fonction, ou bien la valeur d'une constante.
- Les champs de contrôle: ces champs contiennent des informations permettant de mettre en oeuvre le modèle de réduction de graphes. Ils

contiennent par exemple l'état du noeud (repos, réduit, en réduction,..), et le nombre d'arguments restant à évaluer.

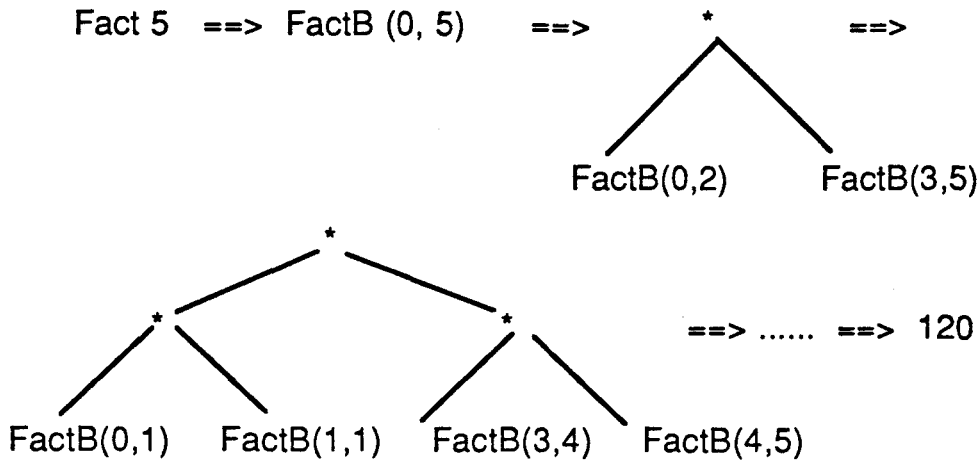


fig 2.8 : La réduction de graphes dans la machine ALICE

II.4.2.3 Le prototype de la machine ALICE [Crip86]

Fonctionnellement la machine ALICE est composé de 3 types d'unités (fig 2.9).

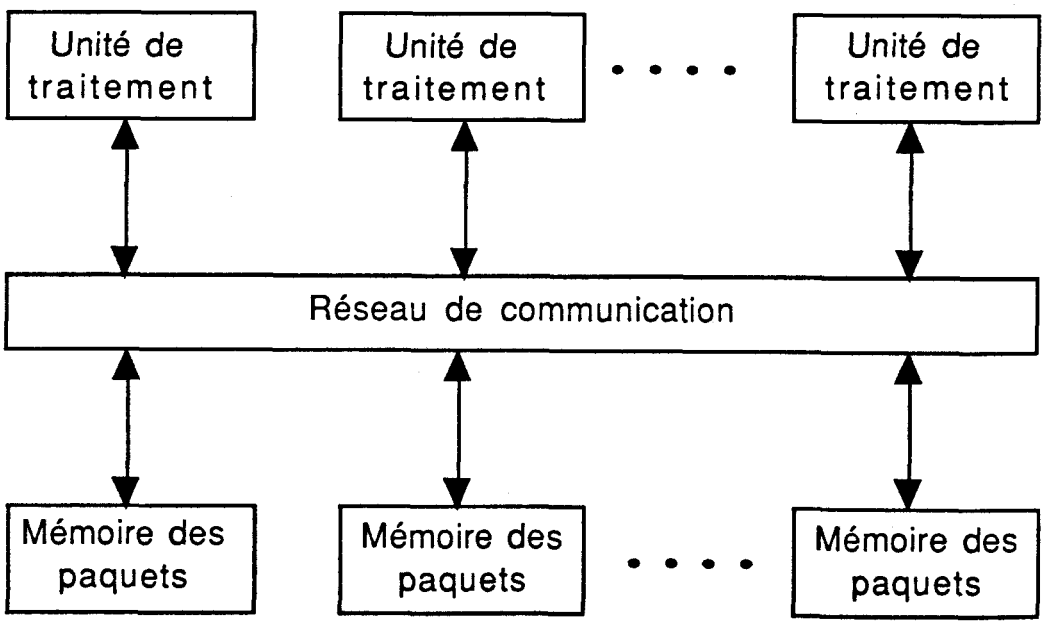


fig 2.9 : Architecture du prototype de la machine ALICE

1) Les unités de traitement réalisent la réduction des noeuds du graphe. Chacune de ces unités a été implèmentée par 5 Transputers. Deux de ces Transputers réalisent l'opération de réécriture des paquets (redex), ils sont nommés PRU (packet rewriting unit). Un Transputer gère la mémoire cache (FDU) qui contient les définitions de fonctions utilisées par les PRU. Les deux autres Transputers réalisent la gestion respectivement des paquets à recevoir (ERU) et des paquets a émettre (ETU).

2) Les mémoires des paquets : Ces mémoires stockent une partie du graphe à réduire. Une unité mémoire de paquets est implèmentée par 2 Transputers. Le premier Transputer (PMU) contrôle les paquets requête en entrée, construit les paquets réponses correspondants et les envoie vers le deuxième Transputer qui les transmet vers le réseau de communication

3) Le réseau de communication : Ce réseau d'interconnection véhicule des paquets de données et de contrôle nécessaires pour équilibrer la charge des différentes unités. Deux types de communication existent dans le prototype:

- Les communications internes à une unité qui sont réalisées au moyen des canaux DMA des Transputers impliqués dans le transfert.
- Les communications externes, c'est à dire entre deux unités différentes, sont réalisées aussi par les canaux DMA des Transputers correspondants mais via un réseau de communication. Ce dernier a été réalisé en utilisant un circuit intégré crossbar à 4 voies. Ce crossbar est ensuite utilisé pour construire un réseau "DELTA" à 3 étages permettant de connecter 64 unités de traitement et 64 unités de mémorisation.

Sur le plan logiciel, un programme Hope, avant d'être exécuté, est compilé dans un langage intermédiaire (CTL: compiler target language). Ce langage représente le code exécutable par la machine. Il permet d'avoir un interface indépendant du langage fonctionnel utilisé, et donne la possibilité de développer la partie logicielle indépendamment de l'implémentation choisie.

II.4.3 La machine PIM-R [Onai85]

II.4.3.1 Introduction

La machine PIM-R (Parallel Inference Machine-Reduction) est une machine multiprocesseur dédiée pour l'exécution de programmes logiques. Cette machine rentre dans le cadre du projet japonais de la cinquième génération mené à l'ICOT TOKYO. Le mode de contrôle choisi pour cette machine est le contrôle par les demandes.

II.4.3.2 Le modèle d'exécution

Les concepteurs de cette machine ont retenu comme langage à exécuter PROLOG et CONCURENT PROLOG (CP) . Nous allons brièvement voir dans ce paragraphe comment se fait l'exécution d'un programme dans les deux cas.

Un programme écrit en Prolog est exécuté en parallélisme OU. Ainsi, si nous prenons le programme suivant à exécuter.

But : p1
p1 :- q1, r1, s1
p1 :- q2, r2, s2
..
p1 :- qm, rm, sm

L'unification du but p1 (processus père) avec les m clauses, permet de générer m résolvants (m processus fils). Chaque processus fils à un pointeur indiquant son processus père, ce dernier contient par contre un compteur indiquant le nombre de processus fils générés (fig 2.10).

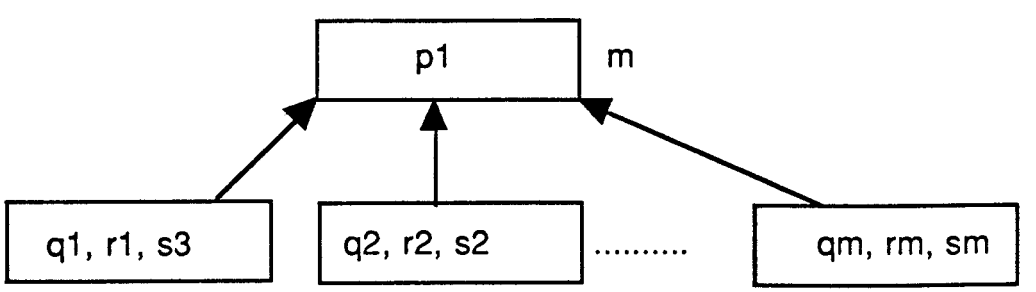


fig 2.10 : Exécution d'un programme en Prolog

La résolution des processus fils est distribuée sur les différents processeurs contenus dans la machine.

L'exécution d'un programme en CP est par contre réalisée en parallélisme ET. Une clause en CP a le format suivant :

t :- g | c

ou t est la tête de clause, g est une suite de buts appelée la garde, et c est aussi une suite de buts appelée corps de la clause. Lorsqu'un but est spécifié, les clauses de la procédure correspondant à ce but sont sélectionnées pour une unification. Les clauses dont l'unification a donnée un succès sont choisies pour évaluer leurs gardes. Seule la première clause qui arrive à évaluer (prouver) sa garde qui est retenue pour la suite de l'exécution. Les autres alternatives sont éliminées ("Don't care non déterminisme). Les buts contenus dans la partie corps de la clause retenue sont exécutés alors en parallélisme ET. Si toute fois il y a des variables partagées entre ces buts, il se crée une relation de producteur-consommateur, et les variables en question jouent le rôle de canaux de communication entre les buts. Lorsqu'un but contient une variable avec la marque "?", il est désigné comme étant consommateur de cette variable. De ce fait, il doit attendre que le but producteur soit exécuté et que cette variable soit instanciée.

Contrairement à un programme PROLOG qui est exécuté en parallélisme OU, ici les opérations d'unification du but avec les différentes clauses et l'évaluation de la garde sont réalisées par le processeur qui contient le but initial. Si nous prenons à titre d'exemple le programme CP suivant:

```
But : p1
p1 :- g1 | c1
p1 :- g2 | c2
..
p1 :- gn | cn
```

L'unification du but et l'évaluation des gardes sont faits par le processeur qui est charge de l'évaluation de p1. Lorsque l'unification et l'évaluation d'une garde sont faites avec succès, il y a test pour savoir s'il s'agit de la première clause qui a réussi à évaluer la garde. Pour cela le processus qui s'occupe de l'évaluation du but p1 a un indicateur, "le c-tag", qui est mis à 1 par le premier processus fils qui évalue avec succès sa garde.

II.4.3.3 L'architecture de la machine PIM-R

La machine PIM-R est composée, comme le montre la figure 2.11, de 2 types de modules: les modules d'inférence (IM) et les modules mémoire des structures (SMM). Ces modules sont reliés par un réseau d'interconnection afin de réaliser des communications soit entre deux IM ou entre un IM et un SMM.

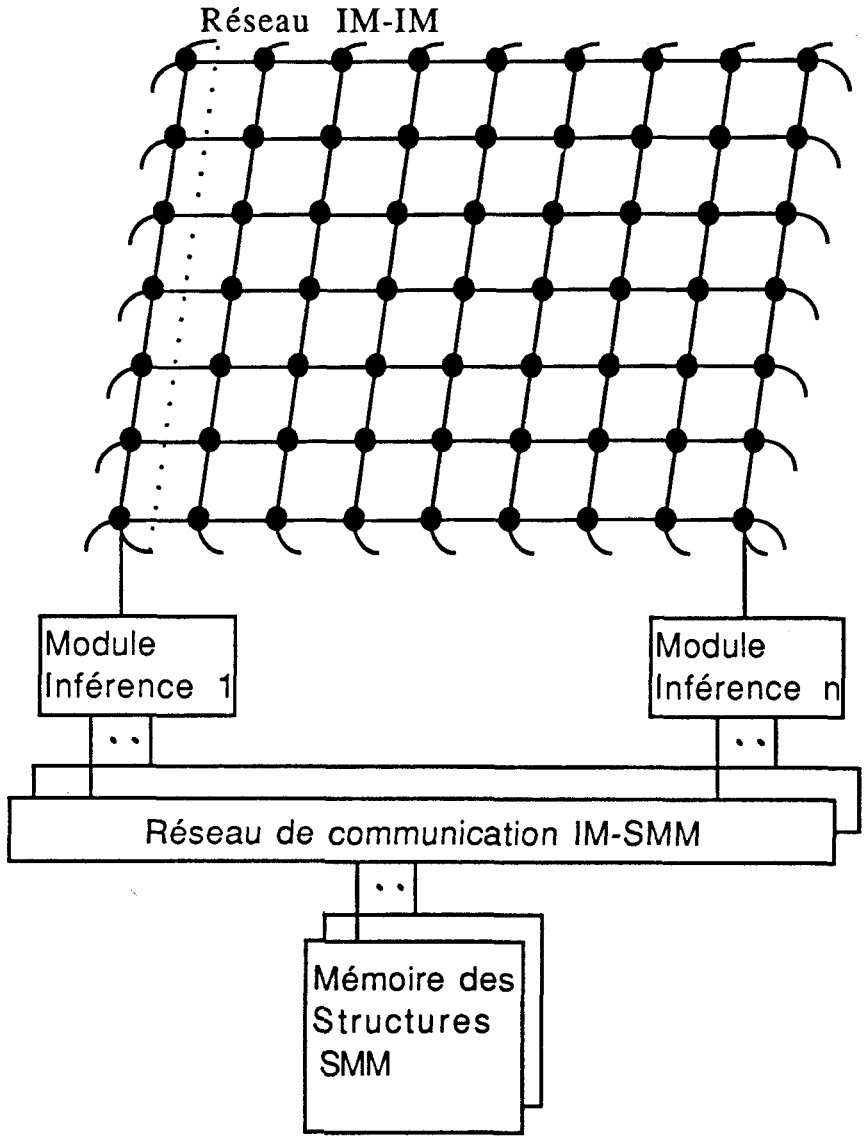


fig 2.11 : Architecture fonctionnelle de la machine PIM-R

a) Les modules d'inférences (IM)

Un IM est composé d'une unité d'unification et d'une unité des processus.

a.1) L'unité d'unification contient:

- Une mémoire des clauses où sont stockées les clauses du programme à exécuter.
- Une unité de correspondance (matcher) qui se charge de sélectionner les clauses dans la mémoire des clauses pouvant s'unifier avec le but reçu de l'unité des processus.
- Un processeur d'unification qui reçoit un but de l'unité de correspondance, une clause de la mémoire des clauses, tente d'unifier le but avec la clause, et envoie le résultat vers le buffer de sortie, qui se charge de l'expédier à l'unité des processus.

a.2) L'unité des processus

L'unité des processus contient 2 type d'unité: une unité de mémorisation et une unité de contrôle (ou contrôleur).

Dans l'unité de mémorisation nous trouvons les buts à traiter (les processus). Afin d'assurer un fonctionnement correcte, l'unité de mémorisation contient aussi des informations de contrôle sur le processus (bloc de contrôle du processus), comme l'état du processus (prêt, en exécution, en attente de réponse d'un processus fils, ...etc), le nombre de processus fils générés, ...etc.

Quant au contrôleur, c'est lui qui se charge de créer, de gérer, ensuite d'éliminer un processus de la mémoire des processus. A titre d'exemple lorsqu'un module d'inférence envoie une suite de buts à l'unité de mémorisation, le contrôleur se charge de créer le (ou les) bloc de contrôle correspondant. De même, lorsqu'un processus est éliminé, le contrôleur réalise une opération de récupération d'espace mémoire libre.

b) Le module mémoire des structures (SMM)

Dans un programme écrit en Prolog ou en CP il y a souvent utilisation de données structurées. La technique qui consiste à copier les données structurées à chaque accès afin de permettre un traitement parallèle, cause une charge système supplémentaire importante.

Pour cette raison, dans la machine PIM-R les données structurées (comme les listes et les vecteurs de taille importante) sont mémorisées dans le module des structures qui permet un partage de la données structurées entre plusieurs IM. En effet un module SMM est connecté à plusieurs IM par le biais du réseau de communication.

II.4.3.4 Conclusion

Un simulateur software de la machine PIM-R a été réalisé et testé. Ainsi un programme Prolog (les 4 reines) et un programme en CP (le quicksort) ont été exécutés afin de collecter des informations sur le fonctionnement de la machine. Ces exécutions ont montrés que le nombre de modules d'inférence influe sur les temps d'exécution. Néanmoins, à partir d'un certain nombre de IM, ce temps d'exécution reste constant (fig 2.12).

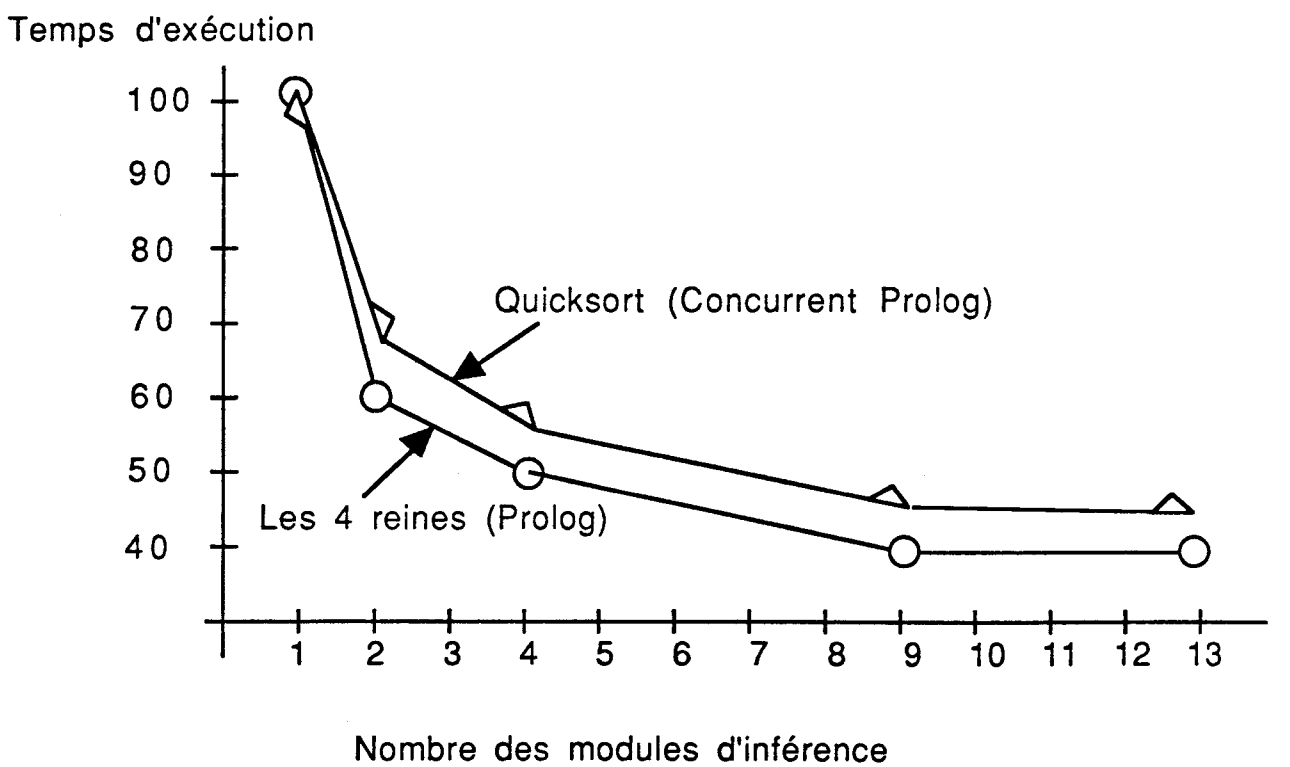


fig 2.12 : Influence du nombre d'IM sur le temps d'exécution

D'autres résultats ont montrés aussi que, de meilleurs performances sont obtenues lorsque la stratégie de distribution des processus fils est faite de manière dynamique, ainsi que lorsqu'un IM réalise un

traitement local en profondeur d'abord. Cette dernière amélioration est réalisée en donnant la priorité haute au but, se trouvant dans la mémoire des processus, qui correspond au noeud le plus profond de l'arbre ET/OU. Lorsqu'une unité d'inférence demande un but à exécuter, c'est celui ayant la plus haute priorité qui est exécuté. Actuellement des simulateurs plus complets sont an phase de développement ainsi qu'un premier prototype en circuit VLSI.

II.5 CONCLUSION

A la fin de ce chapitre nous pouvons dire que, sur le plan de l'architecture de la machine, un mode de contrôle du type non von-Neumann permet de mieux exploiter le parallélisme inhérent à un programme écrit en langage déclaratif. Il faut remarquer néanmoins, que pour les machines controlées par les données des mécanismes supplémentaires sont nécessaires afin de ne pas saturer le système.

Dans les machines controlée par les demandes, il semble qu'il y a des difficultés pour trouver le degré convenable de parallélisme à exploiter, et souvent le programmeur est amené à ajouter des annotations. Ceci est contradictoire avec l'objectif de parallélisation automatique.

Sur le plan logiciel, qu'il s'agissent d'un langage fonctionnel ou d'un langage de la programmation logique, nous pouvons dire que leurs exécutions sur une architecture parallèle posent encore plusieurs problèmes qu'il faut résoudre.

Pour un programme logique par exemple, malgré l'existence de plusieurs types de parallélisme, les chercheurs ne sont pas tous d'accord sur le meilleur choix. Ce dernier devient encore plus difficile à faire lorsqu'on remarque que l'efficacité d'un type de parallélisme donné dépend du type du programme à exécuter.

Comme il a été noté en fin du paragraphe sur les langages fonctionnels, le principal problème est que si le programme a été pensé de manière séquentielle, le parallélisme devient peu accessible, il est par conséquent difficile d'extraire un parallélisme de manière automatique.

Pour éviter ce problème, il est nécessaire que le programmeur change sa manière d'écrire les programmes, afin de prendre en compte le fait que son code est destiné à une exécution sur une machine parallèle. L'une des solutions proposée, c'est d'enrichir le langage par des

opérateurs de traitement parallèles (comme dans FP pour le ApplyToAll ou l'opérateur de construction).

Dans tous les cas on peut dire que la programmation déclarative parallèle n'en est qu'à ses débuts. Afin de prouver sa puissance, il est nécessaire d'avoir des implémentations pour pouvoir exécuter des applications réelles.

Références citées dans le chapitre :

[Amam87], [Back78], [Bell86], [Berk75], [Burt87], [Cone85], [Cous88], [Cousi86], [Darl81], [Darl87], [Deve86], [Denn74], [Hend80], [Magò80], [Niar86], [Onai85], [Peyt86], [Peyt87], [Peyt89], [Trel82], [Turn79], [Vegd84], [Wats87], [Wei88].

CHAPITRE III
LA MACHINE N-ARCH

III.1 LES OBJECTIFS DU PROJET

Le projet N-ARCH, développé au laboratoire d'informatique de l'université des sciences et techniques de LILLE depuis 1986, a pour objectif la conception d'une architecture parallèle adaptée à l'exécution de programmes déclaratifs. Le but initial du projet n'est pas la conception d'une machine langage, mais l'étude des possibilités d'intégration au niveau du matériel de concepts facilitant l'évaluation de programmes de nature déclarative.

La méthode de conception utilisée dans notre projet constitue un compromis entre l'approche descendante (top-down approach) et l'approche ascendante (down-top approach)(fig 3.1). Dans la première approche le but recherché est la conception d'une architecture qui permet l'exécution dans les meilleures conditions (temps, coût, sécurité, ...) d'un langage particulier.

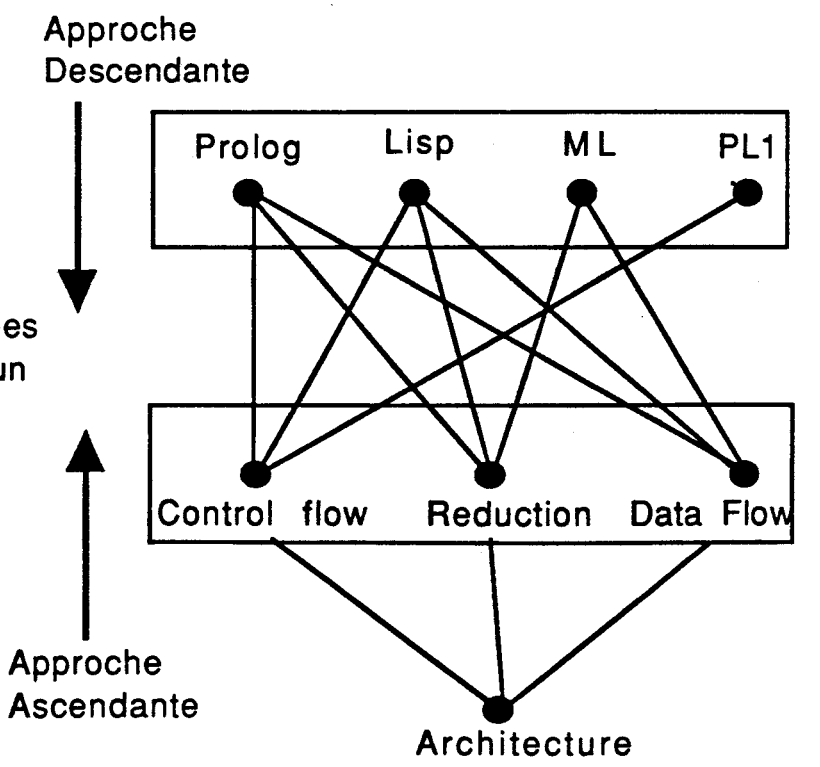


Fig 3.1 : les 2 approches dans la conception d'un ordinateur

Ce type de machines correspond à ce que l'on nomme les machines langages. Un modèle d'évaluation est alors conçu afin de réaliser l'exécution du langage choisi. Ce modèle est détaillé et adapté aux possibilités offertes par la technologie actuelle afin de fournir un modèle physique. Une machine réelle est alors obtenue par la mise en oeuvre de ce modèle physique. L'inconvénient majeur de cette approche est que chaque langage peut avoir sa propre machine [Kenn83].

Le point de départ dans l'approche ascendante est l'architecture de la machine. En effet la tâche essentielle du concepteur dans ce cas, est la mise en oeuvre d'une machine intègrant des capacités matérielles diverses permettant l'exécution de différents langages. L'inconvénient dans ce type d'approche est, bien sûr, l'inefficacité de la machine dans l'exécution de certains types de langages [Hwan87a].

Dans le projet N-ARCH, il s'agit de concevoir une architecture multiprocesseurs permettant l'exécution, de manière efficace, d'un type particulier de langages qui sont les langages déclaratifs. Comme pour l'approche ascendante le point de départ dans le projet N-ARCH est la définition d'une architecture. Néanmoins cette définition est réalisée en prenant en compte les caractéristiques des langages déclaratifs. Comme il a été indiqué dans le chapitre précédent il existe 2 ensembles de langages déclaratifs qui sont: les langages fonctionnels et les langages de la programmation logique.

Nous pouvons dire que, par rapport aux autres machines, la machine N-ARCH se différencie par les points suivants :

- Chaque processeur de la machine dispose d'une mémoire privée (ou mémoire locale). L'utilisation de cette mémoire permet un gain en temps de traitement et reduit la quantité de messages à faire transiter par le réseau de communication.
- L'architecture N-ARCH n'est pas conçue pour exécuter un langage bien déterminé, mais une famille de langages ayant certaines caractéristiques communes, qui sont les langages déclaratifs.
- Le mécanisme de répartition de la charge est réalisé par utilisation de fonctions de hachage.

L'architecture projetée doit répondre à plusieurs critères:

1: Disposer d'un mode d'exécution parallèle, et d'un contrôle totalement distribué.

2: Intégrer des outils associatifs bien adaptés au traitement symbolique et aux machines Non-von Neuman.

3: Supporter de nouveaux modes de commandes (par réduction et contrôle par les données).
4: Exécuter une grande variété de langages déclaratifs.
5: Utiliser des circuits VLSI adaptés.
Dans ce travail nous nous sommes intéressés principalement aux trois premiers critères. Ce sont donc ces 3 objectifs que nous allons détailler par la suite. Les objectifs 4 et 5 font l'objet d'autres travaux. A la fin de ce chapitre, nous donnerons un aperçu sur l'approche et les solutions qui ont été retenues pour atteindre ces deux objectifs.
De ce fait, l'une de nos contributions au projet consiste à définir l'architecture d'un noeud de la machine multiprocesseurs N-ARCH, ainsi que son mode de fonctionnement et de contrôle. L'architecture du noeud devra contenir des outils facilitant l'execution des langages déclaratifs. Un autre aspect du projet étudié dans ce travail est le choix du réseau d'interconnection entre les noeuds du réseau.

III.2 PRÉSENTATION DE L'ARCHITECTURE N-ARCH [Tour86]

III.2.1 : N-ARCH une architecture multiprocesseurs.

Comme indiqué dans le chapitre I, l'augmentation du nombre de processeurs engendre (en général) une réduction du temps d'exécution d'un programme. Ces processeurs sont donc appelés à coopérer pour exécuter les modules (ou tâches) qui composent le programme. Ceci exige, par conséquent, des échanges de données entre les différents processeurs impliqués dans le traitement. Un outil de communication entre les processeurs est de ce fait nécessaire. Cet outil est couramment appelé réseau de communication.

La quantité de données à échanger, la fréquence avec laquelle ces données sont transmises, la vitesse de transmission permise sur le réseau de communication, ainsi que le chemin pris par les données transmises sont des facteurs à prendre en compte dans le choix de la topologie du réseau de communication.

Il existe 2 grandes familles de réseaux de communication, qui ont donné lieu à 2 types de configurations de machines multiprocesseurs. Ce sont: la configuration en mémoire partagée (ou système multiprocesseurs fortement couplé) et la configurations en mémoire privée (ou système faiblement couplé) [Hwan87b].

Dans la première configuration les mémoires et les processeurs sont séparés par un réseau de communication (fig 3.2).

Cette configuration permet d'avoir des temps d'accès uniformes quelque soit le lieu où est stockée l'information accédée. Cependant cette uniformité peut être compromise lorsqu'il y a des conflits d'accès aux commutateurs et de plus ce temps de communication peut augmenter de manière sensible lorsque le nombre de modules mémoires et processeurs devient important.

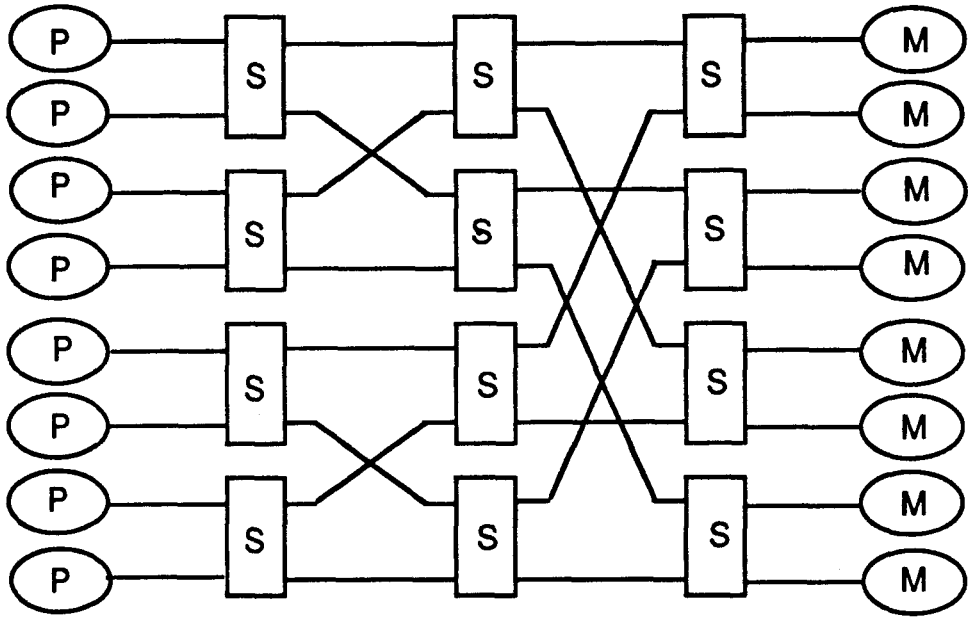


Fig 3.2 : Une configuration multiprocesseurs fortement couplé

Dans la deuxième configuration, chaque processeur est attaché à sa propre mémoire privée (ou locale). Les données se trouvant dans la mémoire locale sont accédées de manière directe, alors que celles qui sont stockées dans la mémoire privée d'un autre processeur sont accédées par émission d'un message de demande d'accès (ou message requête). Ce message est alors acheminé vers le processeur qui gère la memoire locale contenant l'information désirée. Cette configuration permet d'exploiter la localité dans le traitement, car les données interdépendantes peuvent être stockées dans la même mémoire locale. Ceci offre un gain en temps d'accès, du fait que la plus grande partie des données accédées par un processeur se trouve mémorisée localement.

C'est cette configuration qui a été choisie pour la machine N-ARCH. En effet dans la machine N-ARCH chaque noeud peut être représenté par 3

unités fonctionnelles: Un Commutateur (appelé aussi coupleur d'E/S ou switch), un processeur, et une mémoire. (fig3.3).

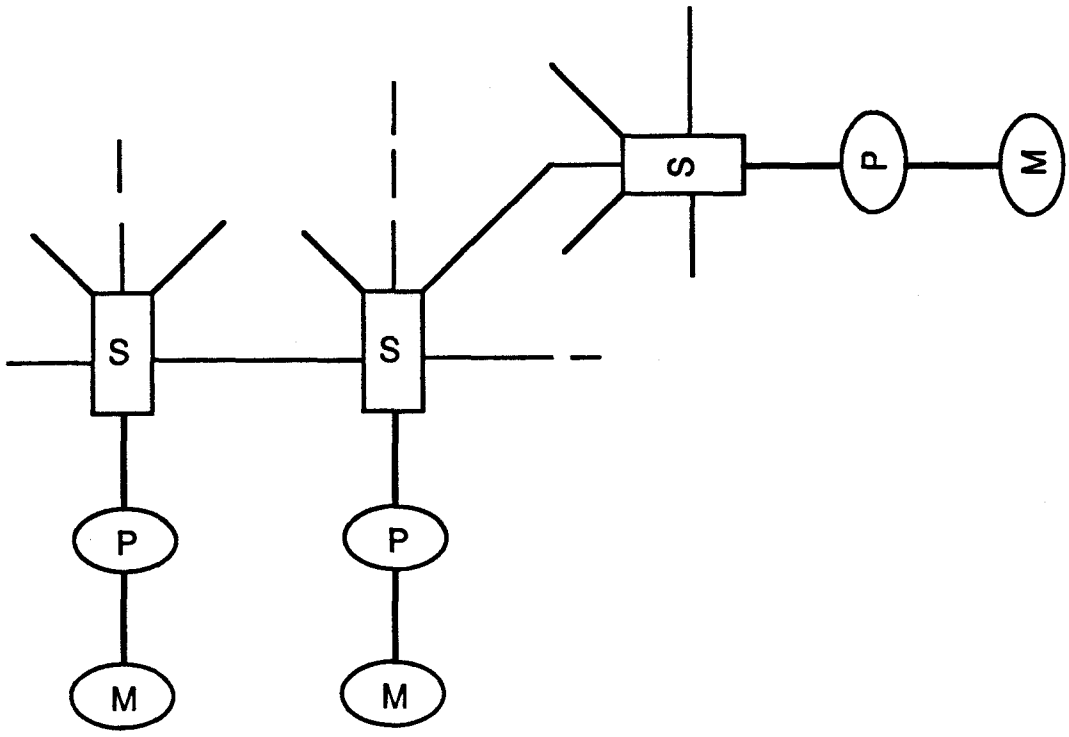


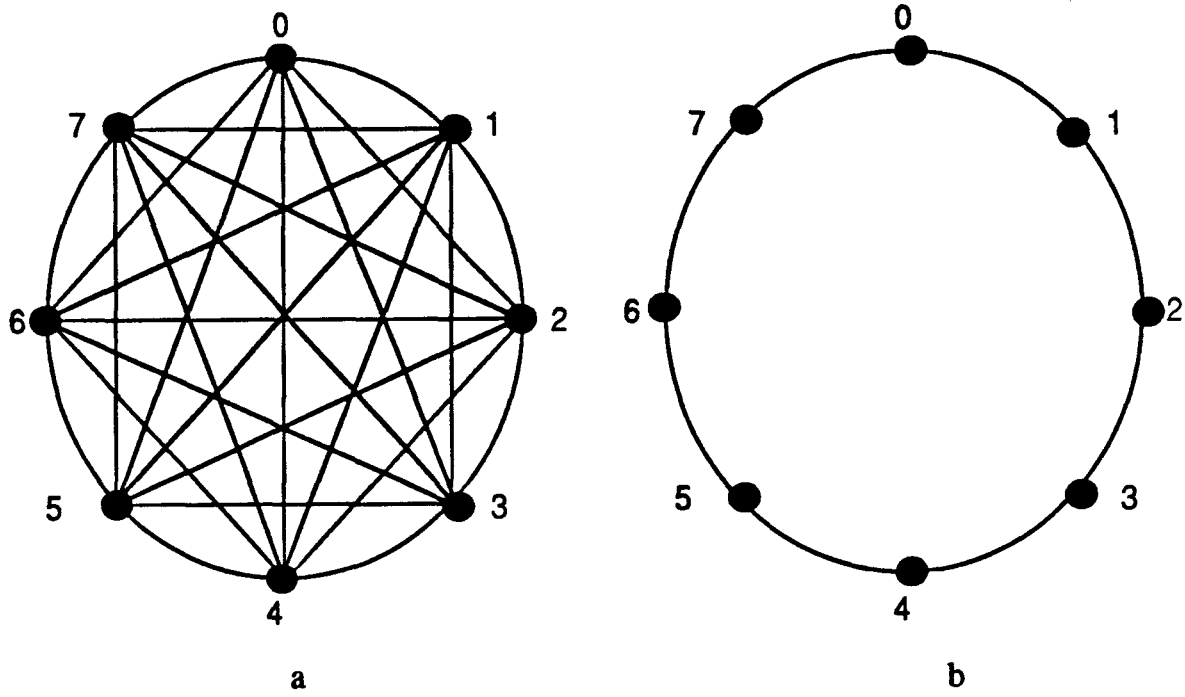
Fig 3.3: Une configuration multiprocesseurs faiblement couplés

Le rôle du commutateur est de recevoir un message et de lire le numéro du noeud destination de ce dernier. Le commutateur se charge aussi d'envoyer ce paquet vers l'unité processeur si le paquet reçu lui est destiné. Si non il est envoyé vers l'un des commutateurs des noeuds voisins. Lorsqu'un noeud dispose de plusieurs voisins, il y a utilisation d'un algorithme de routage pour déterminer lequel des voisins recevra le message.

Les noeuds dans cette configuration peuvent être connectés de plusieurs manières, ce qui donne plusieurs types de réseaux de communication. En général, il existe un compromis entre les performances du réseau de communication et sa complexité.

Les deux situations extrêmes sont la connection complète où tout noeud dispose d'un lien vers chacun des autres noeuds, et la situation où les noeuds sont reliés en anneau (fig 3.4). La topologie en connection complète (fig 3.4a) est celle qui offre le minimum de temps d'accès (donc des performances élevées), un algorithme de routage simple, mais un degré de complexité élevé car elle exige un nombre de liens de l'ordre de

N^2 (où N est le nombre de processeurs). Dans ce paragraphe, nous utiliserons des diagrammes représentant les différentes capacités mémoires qu'un processeur peut accéder en fonction du temps d'accés (fig 3.5). En d'autres termes, pour chaque entier t représentant un certain temps d'accés le diagramme donne la quantité mémoire éxistante dans le réseau qui peut être accédée avec ce temps t. Ce diagramme de répartition du temps d'accés permet de comparer les performances des différents réseaux de communication en rapidité d'accés.



a b

Fig 3.4 : Le réseau en connection complète
et le réseau en anneau

La figure 3.5a donne la répartition des temps d'accès pour un réseau en connection complète, où une petite quantité de mémoire est disponible avec un temps d'accès court et le reste des capacités mémoires est accédé en un temps légèrement plus grand (dû aux temps de communication). Lorsque le nombre de processeurs est de quelque centaines, il est impossible de réaliser (d'un point de vue technologique) des cartes circuit imprimé possédant cent ports d'entrée/sortie.

La topologie en anneau (fig 3.4b), par contre, peut facilement être étendue pour intègrer un nombre plus élevé de processeurs, mais implique un serieux allongement dans les temps d'accès pour les données qui ne sont pas locales à un processeur. En effet pour accéder à une donnée le message demande d'accès risque de traverser un nombre

important de processeurs. Ceci provoque, par conséquent un ralentissement dans les accès et détériore les performances de la machine. La figure 3.5b donne la répartition des temps d'accès dans un réseau de 8 processeurs reliés en anneau.

Plusieurs concepteurs ont proposés des réseaux de communication structurés en arbre pour leurs machines. Lorsque le programme peut être décomposé en sous tâches parallèlles et indépendantes, cette structure d'arbre permet d'avoir une correspondance directe entre le matériel (l'arbre des processeurs) et le programme (l'arbre des tâches ou l'arbre des processus).

Néanmoins, il est en général nécessaire de dupliquer des données stockées dans un noeud du réseau. Ceci provoque alors un goulot d'étranglement au niveau du noeud racine de l'arbre. De plus, comme on le verra dans le paragraphe suivant, la politique de répartition choisie pour notre machine exige que chaque noeud du réseau puisse être racine d'un arbre couvrant tous les noeuds du réseau. De ce fait la topologie en arbre ne convient pas car seul le noeud racine de l'arbre vérifie cette condition.

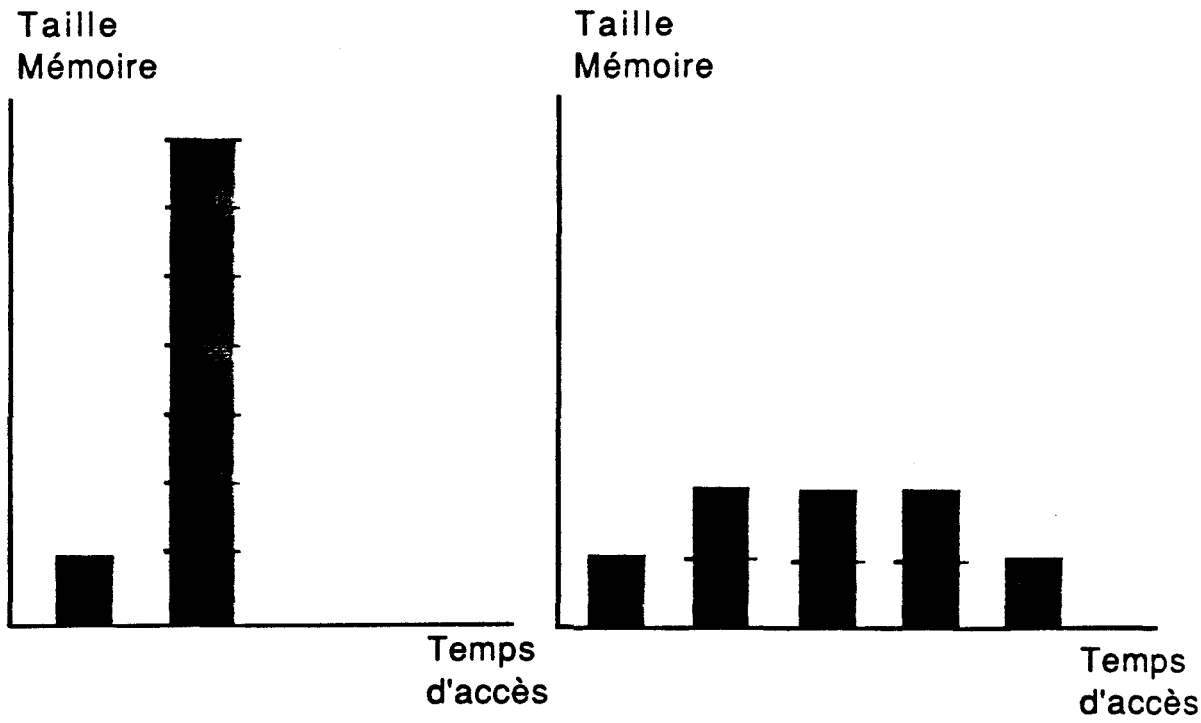
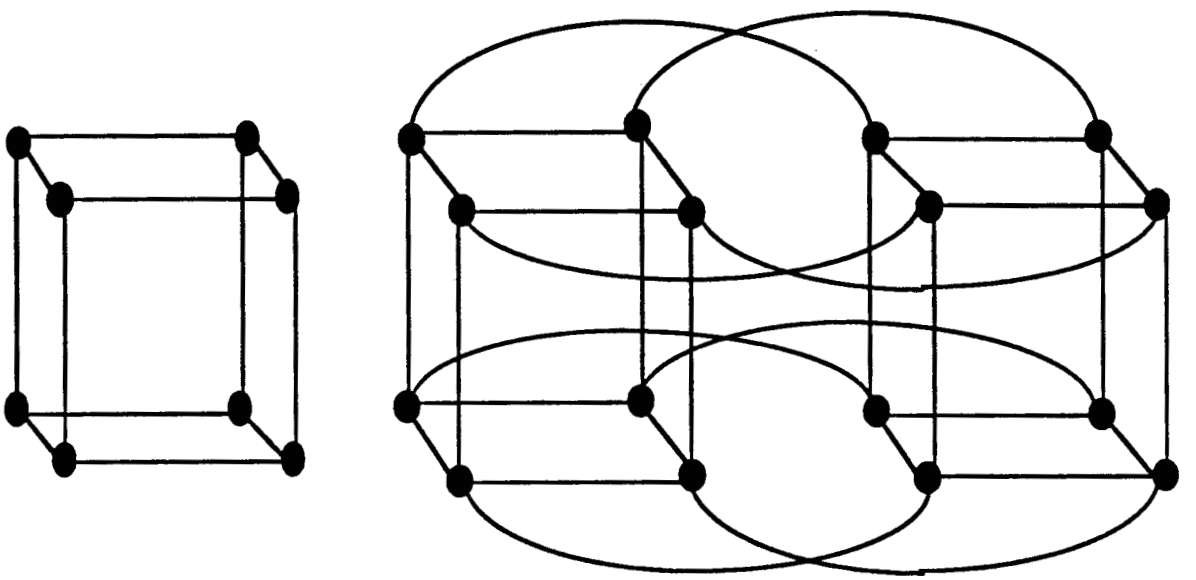
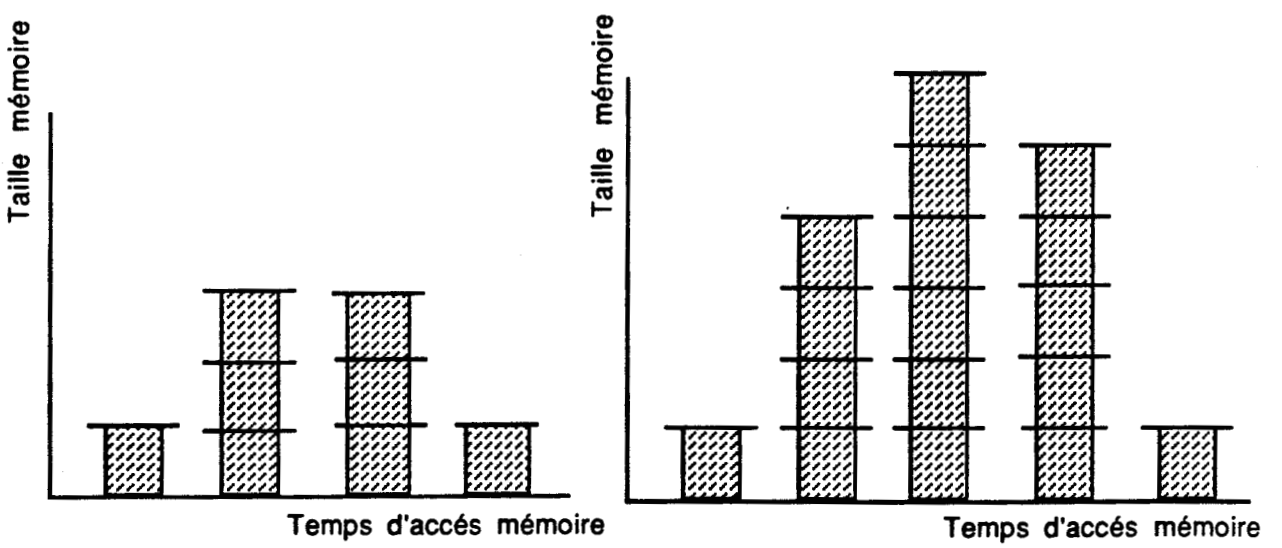


Fig 3.5 : La répartition des temps d'accès
pour la connection complète (a) et le réseau en anneau (b)

Dans le projet N-ARCH plusieurs autres topologies ont été étudiées. Celle qui a été retenue pour la réalisation de l'émulateur, est la structure en hypercube [Bhuy82]. Le nombre de processeurs dans un hypercube s'écrit toujours sous la forme $N = 2^d$, ou d est un entier quelconque appelé degré de l'hypercube. Chaque noeud du réseau représente alors un sommet de l'hypercube de degré d qui est connecté à d voisins. Une connection entre 2 noeuds du réseau hypercube est propre à ces 2 noeuds et elle est réalisée de manière directe (fig 3.6).



a : Le réseau hypercube d'ordre 3 et d'ordre 4



b : La répartition du temps d'accès dans l'hypercube d'ordre 3 et d'ordre 4

Fig 3.6 : La topologie du réseau hypercube

Dans la figure 3.6b, nous avons représenté la répartition du temps d'accès. Cette figure montre que la quantité de mémoire pouvant être accédée croit de manière exponentielle avec le temps d'accès jusqu'à ce que la "ceinture" de l'hypercube à été atteinte. A partir de ce moment elle décroit de manière exponentielle également.

Le réseau en hypercube constitue, comme on peut le voir dans la figure 3.6b un bon compromis entre les 2 situations extremes vues précédemment. En effet cette topologie offre plusieurs points intéressants:

_ La distance moyenne qu'un message traverse pour aller d'un noeud à un autre est relativement réduite par rapport à d'autres types de réseau (logarithmique en nombre de processeurs). Ceci évite d'avoir des temps d'accès importants.

_ Le nombre de liens nécessaires par noeud est égal au degré de l'hypercube. Ceci permet de construire des réseaux contenant un nombre important de processeurs sans pour cela exiger un grand nombre de ports d'entrée/sortie par noeud.

_ L'algorithme de routage dans un réseau en hypercube est trés simple, et comme il existe plusieurs chemins pour aller d'un noeud à un autre, le réseau a une grande tolérance aux pannes.

_ Comme on le verra dans le chapitre IV tout noeud d'un réseau en hypercube peut être racine d'un arbre contenant tous les noeuds du réseau et dont les branches sont des chemins hamiltoniens. Cette caractéristique facilite l'implémention de la méthode de répartition de la charge choisie pour la machine.

Malheureusement, le réseau de communication en hypercube ne peut pas toujours être étendu pour intègrer un plus grand nombre de processeurs, car le nombre de ports d'entrée/sortie disponible au niveau d'un noeud influe sur le degré maximum que peut avoir le réseau hypercube.

III.2.2 : La répartition des programmes par fonctions de hachage

Dans toute architecture multiprocesseurs, l'un des problemes majeurs est souvent le choix de la distribution des objets sur les différentes unités de traitement. Cette distribution doit assurer d'un côté un grand degré de parallélisme en réalisant une distribution uniforme sur tous les noeuds du réseau, ce qui peut provoquer un accroissement des opérations de

communication, et doit exploiter, de l'autre côté, la localité existant dans les traitements des données. Cette localité est difficile à saisir de manière dynamique et peut conduire à des saturations locales des noeuds.

Dans la machine N-ARCH la méthode de répartition choisie utilise une technique de hachage. Cette méthode de hachage projette l'ensemble des noms des objets manipulés sur l'ensemble des noeuds.

L'allocation des objets aux différents noeuds et la procédure de routage des messages sont toutes deux réalisées selon la même méthode, reposant sur l'application de fonctions de hachage aux couples (nom de l'objet, nom de contexte). Ce nom de contexte représente l'environnement dans lequel l'objet a été créé. Ceci permet de déterminer le noeud du réseau où l'objet sera stocké ou aiguiller un message d'un noeud à un autre.

L'utilisation d'une technique de hachage provoque des phénomènes de collisions. Lors d'une collision, un objet qui doit être stocké dans un noeud déterminé par l'évaluation de la fonction de hachage (appelé dans ce cas fonction de hachage globale ou FHG) ne peut y trouver place et se trouve rejeté vers un autre noeud. Afin de minimiser le coût des communications, il est nécessaire de disposer d'une liaison physique entre le noeud où doit être stocké l'objet victime de la collision et le noeud suivant succeptible de l'accueillir. De ce fait les techniques classiques de résolution des collisions ne peuvent être utilisées dans notre architecture.

La solution proposée pour résoudre le probléme des collisions dans la machine N-ARCH est d'associer à un noeud saturé un ensemble de noeuds qui lui sont physiquement connectés, et de sélectionner le noeud candidat à la mémorisation de l'information en appliquant une fonction de hachage locale (ou FHL). Cette fonction de hachage locale a comme image cet ensemble restreint de noeuds. De cette manière les objets en collisions (ou objets synonymes) sont acheminés à travers une suite de noeuds qui forme un chemin. L'ensemble des chemins que les paquets en collision peuvent prendre à partir d'un noeud donné forme une arborescence. Cette arborescence est appelée arbre de collision.

Un objet en collision suivra, par conséquent, un chemin menant de la racine à une feuille de l'arbre de collision jusqu'à la rencontre d'un noeud non saturé. Le chemin suivi est un chemin hamiltonien qui contient tous les noeuds du réseau.

Cette technique permet d'utiliser toutes les capacités mémoires du réseau pour stocker les informations, et évite des passages inutiles par des noeuds précédemment accédés.

Le mécanisme de recherche d'un objet quelconque dans le réseau utilise le même principe. La fonction de hachage globale est alors appliquée sur le couple (nom de l'objet, nom de contexte) pour déterminer le noeud où le message demandeur doit être expédié. Si l'information n'est pas présente (message en collision), la recherche se poursuit à travers un chemin hamiltonien dans l'arbre de collision jusqu'à ce que l'information recherchée soit trouvée. Nous avons ainsi, un mécanisme de répartition des objets basé sur 2 niveaux de hachage. Un premier niveau utilisant la fonction de hachage globale et un deuxieme niveau qui utilise la fonction de hachage locale.

III.2.3: Les objets manipulés dans la machine

Comme nous venons de le voir, les communications dans la machine N-ARCH se font par émission-réception de messages. Dans la suite de ce travail, nous utiliserons aussi le terme *paquet* pour désigner un message.

Nous pouvons en général distinguer trois types de paquets:

1- les paquets d'initialisation (ou de chargement). Un paquet initialisation contient un objet à stocker dans un noeud et n'exige pas de réponse.

2- Les paquets requête qui recherchent un objet donné.

3- Les paquets résultats (ou réponses) qui sont les réponses à des requêtes antérieures.

Afin de faciliter le traitement, il est nécessaire d'associer à chaque objet des informations supplémentaires (les marques) qui précisent d'une part le type de l'objet (marque de type : expression, fonctions, liste, ...), d'autre part l'état de l'objet (marque d'état : objet réduit, partiellement évalué, calculable, ...). Un objet mémorisé dans un noeud du réseau sera donc composé des champs suivants:

_ Marques d'états.

_ Nom de l'objet qui contient une partie statique et une partie dynamique (cette partie dynamique correspond au nom de contexte).

_ Marque de type.

_ Valeur de l'objet (ou sa définition).

III.2.4: Les unités fonctionnelles d'un noeud du réseau

Nous avons précédemment indiqué que la machine N-ARCH est composée d'un certain nombre de noeuds reliés par un réseau hypercube. Nous avons aussi vu que chaque noeud doit être capable de réaliser les 3 fonctions de base qui sont : La communication, le traitement et la mémorisation (fig 3.7).

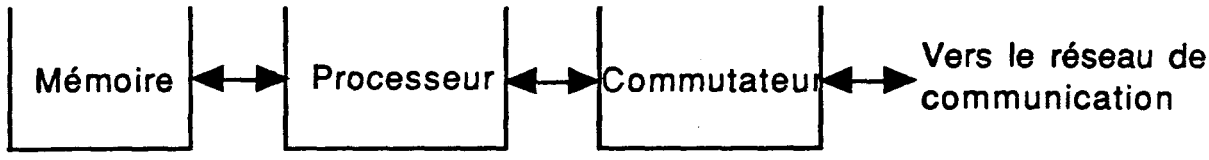


Fig 3.7 : Les 3 unités de base dans un noeud N-ARCH

Nous allons voir, dans ce paragraphe la composition de chacune des unités responsables de ces fonctions.

III.2.4.1 L'organe de communication (ou commutateur)

Les communications entre un noeud et ses voisins sont réalisées de manière autonome vis-à-vis des opérations de traitement réalisées à l'intérieur du noeud. Le rôle de l'organe de communication est donc de décharger l'organe de traitement des opérations liées à l'émission et la réception des messages. Il peut être divisé en deux blocs fonctionnels, qui sont l'unité d'émission et l'unité de réception (fig 3.8).

L'unité de réception a pour fonction la réception des paquets sur chacun des liens d'entrées. Ces paquets sont ensuite examinés par un aiguilleur pour être stockés dans l'une des 2 mémoires. Comme il est montré dans la figure 3.8, nous avons prévu un aiguilleur par récepteur. Ceci permet d'éviter les conflits de demandes de service à un seul aiguilleur. De plus ces aiguilleurs seront très efficaces lorsque le niveau de grain des opérations est fin, impliquant ainsi l'existence d'un grand nombre d'opérations de communications. La fonction d'un aiguilleur est l'envoi du paquet reçu soit vers la RAM d'entrée ou la RAM de sortie, suivant que le paquet est destiné au noeud en question ou non. L'unité d'émission est, elle composée de la RAM de sortie, d'un routeur et de plusieurs émetteurs. La RAM de sortie contient aussi bien les paquets qui

sont générés par l'unité de traitement, que les paquets reçus par l'unité de réception et qui sont à aiguiller vers un des noeuds voisins. Le routeur lit les paquets dans la RAM de sortie un à un, applique l'algorithme de routage et les envoie vers l'un des noeuds voisins si le lien correspondant est libre.

Les récepteurs et les émetteurs de l'organe de communication fonctionnent en paires. Pour cette raison les connections avec les voisins peuvent être représentées par des liaisons bidirectionnelles.

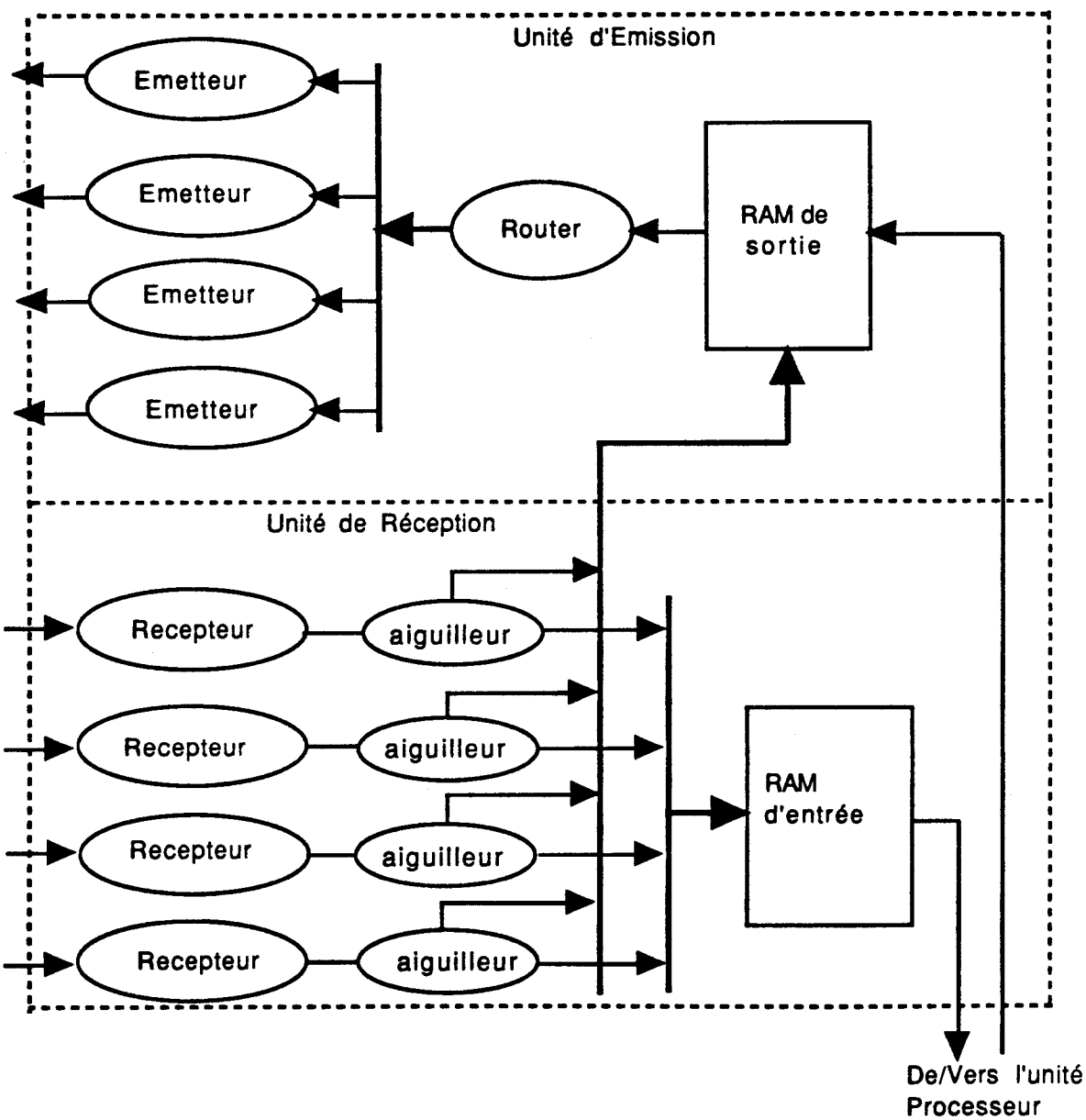


Fig 3.8: Composition de l'unité de communication

III.2.4.2 L'organe de traitement

Cet organe réalise le modèle d'exécution choisi pour la machine. C'est aussi cet organe qui devra réaliser l'ensemble des tâches de contrôle et de gestion des demandes d'accès à l'organe de mémorisation. Il est composé de 2 unités: L'unité de contrôle et l'unité de traitement.

La fonction de l'unité de contrôle (UC) est d'interpréter un paquet et de réaliser ensuite le traitement nécessaire. Comme nous le verrons dans le chapitre IV, ceci peut provoquer la génération d'autres paquets ainsi que l'accès aux différentes mémoires contenues dans le noeud.

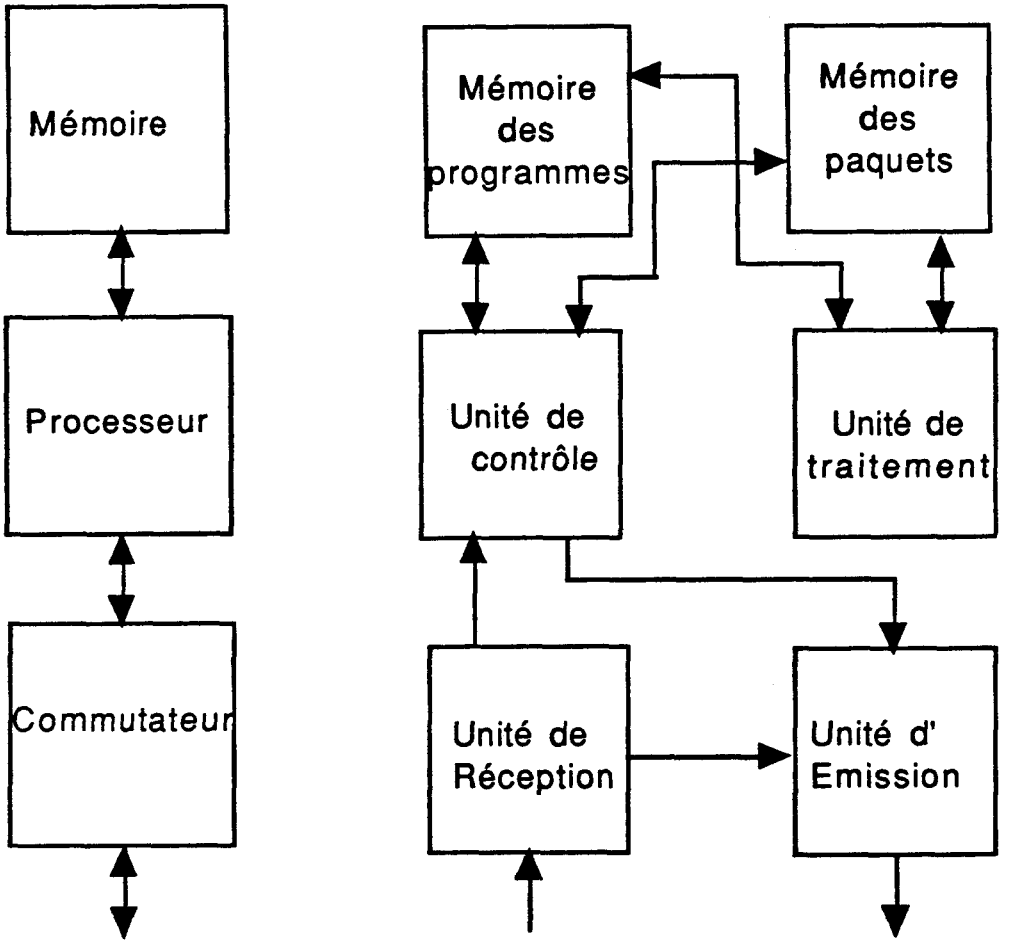


Fig 3.9: Première étape dans la décomposition dans le noeud N-ARCH

Le traitement d'un paquet réponse par exemple nécessite l'accès à la mémoire, afin de mettre à jour l'objet récepteur. Cet objet est alors remplacé soit par sa valeur (réduction de graphes ou contrôlé par les données) ou bien par sa définition (réduction de chaînes). Lorsque l'unité

de contrôle reçoit un paquet requête pour un objet qui existe en mémoire, il y a mémorisation du noeud émetteur de la requête. Cette information nous permet d'éviter d'appliquer les fonctions de hachage (globale et locale) sur les paquets réponse et permet d'expédier ces paquets directement vers leur destinataire. Ceci permet d'avoir un gain de temps et élimine l'utilisation de paquets réponses en collision.

L'unité de traitement a pour fonction de détecter au niveau de la mémoire une opération à effectuer. Ceci est réalisée en testant la marque d'état associée à chaque objet qui indique si ce dernier est réductible (calculable). Un objet calculable peut être une expression qui a reçu tous ses arguments, une fonction qui a reçu toutes ses variables dans un langage fonctionnel, ou un but qui peut s'unifier avec une clause du programme dans un langage logique.

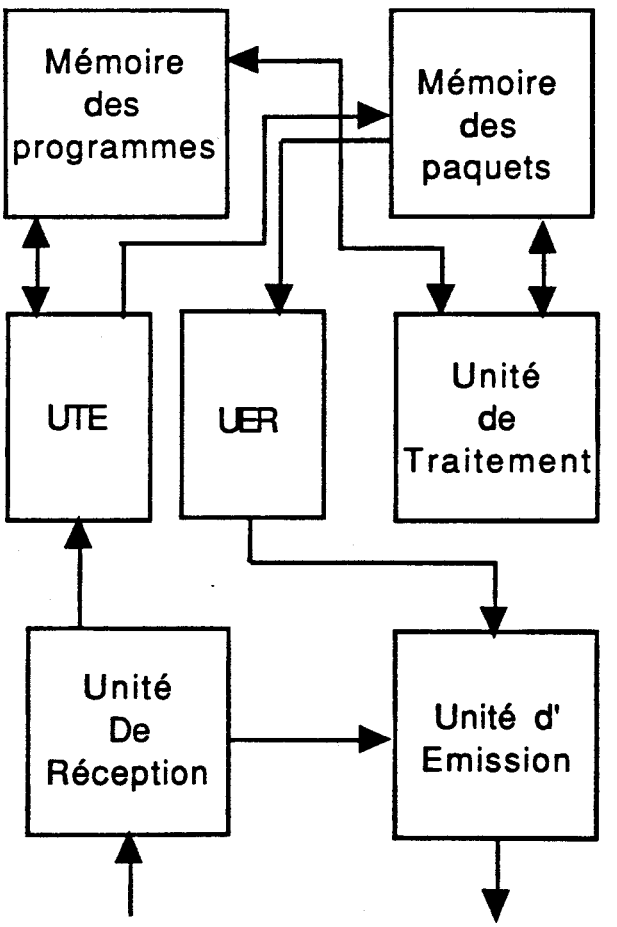


Fig 3.10 : Deuxième étape dans la décomposition du noeud N-ARCH

Une fois l'objet réduit, l'unité de traitement réalise une écriture dans la mémoire afin de remplacer l'objet calculé par son résultat. En mode de

contrôle par réduction de graphe ou par les données, lorsqu'il y a partage d'un objet, cet objet n'est calculé qu'une seule fois. La réduction d'un objet peut par conséquent engendrer l'émission de plusieurs paquets réponses. De ce fait, comme nous allons le voir dans le paragraphe suivant, l'organe de mémorisation est divisé en 2 unités: La mémoire des objets ou des programmes (MAM) et la mémoire des paquets (MAP).

Lorsqu'un résultat a été calculé, il faut donc non seulement mettre à jour l'objet correspondant dans la MAM mais aussi envoyer une copie du résultat vers la MAP afin de permettre l'envoi des paquets réponse aux objets qui partagent ce résultat.

Une fois que l'unité de traitement a mis à jour des paquets réponses, il est nécessaire que l'unité de contrôle puisse y accèder pour les expédier vers l'organe de communication. Afin de permettre un parallélisme dans le traitement des paquets en entrée et l'émission des paquets réponse, nous avons décomposé l'unité de contrôle en 2 sous-unités: Une unité se chargeant du traitement des paquets en entrée (unité de traitement des paquets en entrée ou UTE) et une autre qui prend en charge les paquets réponse mis à jour au niveau de MAP et les dépose dans la RAM de sortie (unité d'émission des paquets réponse ou UER). Ceci permet d'exploiter au mieux les ressources du noeud et donne une description plus fine des organes composant le noeud (fig 3.10).

III.2.4.3 L'organe de mémorisation

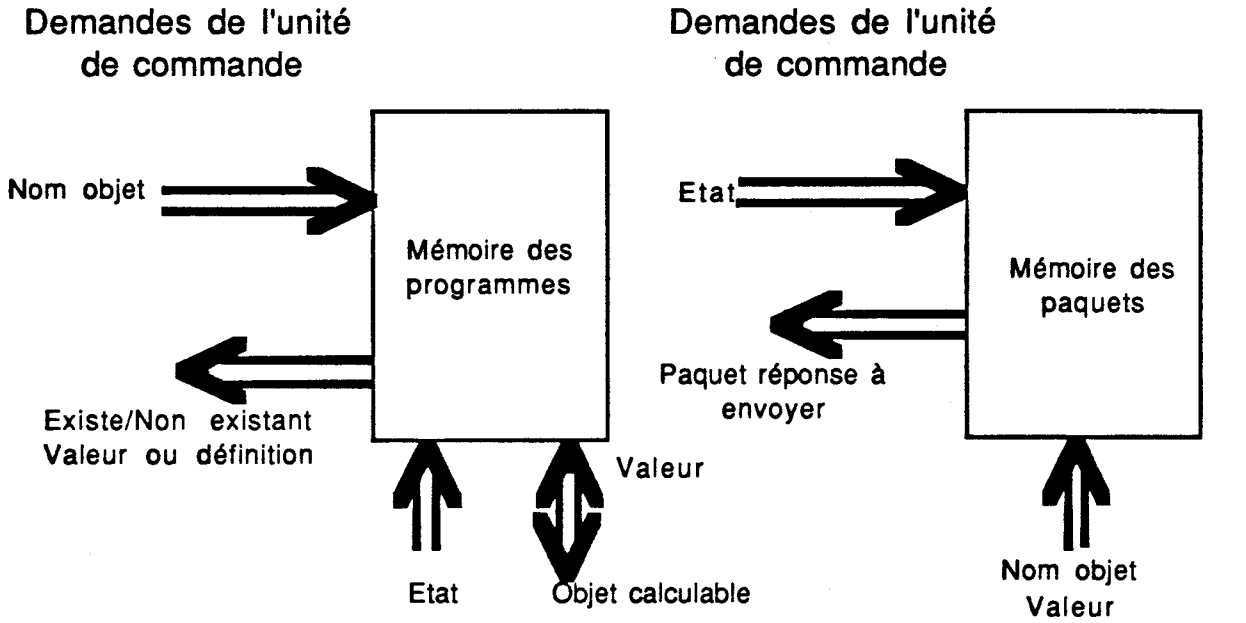
Nous avons précédement indiqué que cet organe a pour fonction le stockage des objets et des paquets réponse en attente de leur résultat. Il est constitué de 2 unités: la mémoire des programmes (MAM) et la mémoire des paquets (MAP).

La mémoire des programmes a pour fonction de stocker le programme (ou partie du programme) qui a été alloué au noeud en question. Ce programme peut être composé d'expressions, de fonctions, de définitions de structures (listes), règles, ...etc. Les accès à la mémoire des programmes peuvent se faire de plusieurs manières:

* S'il s'agit d'une opération de lecture ou de mise à jour par l'unité de commande, l'accès se fait en utilisant le nom de l'objet.

* S'il s'agit par contre d'une opération de lecture ou de mise à jour par l'unité de traitement, l'accès est réalisé en utilisant le champ état de l'objet (calculable à l'étape de lecture ou en phase de calcul à l'étape de

mise à jour) (fig 3.11a). Ces 2 opérations d'accès permettent, en fait, de rechercher un objet parmi un ensemble. L'objet recherché n'est pas connu par son adresse (comme dans les opérations de lecture ou d'écriture dans une RAM) mais par son contenu (ou une partie de son contenu). De ce fait il est intéressant d'utiliser des mémoires adressables par le contenu (CAM: Content Adressable Memory) pour implanter la mémoire des programmes. Ce type de mémoires est aussi appelé "mémoire associative".



a: Lecture/MAJ dans la mémoire des programmes

b: Lecture/MAJ dans la mémoire des paquets

Fig 3.11 : Les accès aux deux mémoires du noeud N-ARCH

La mémoire des paquets, elle, a pour fonction de stocker les paquets réponse en attente d'une mise à jour par l'unité de traitement. Dans cette mémoire 2 types de paquets réponse peuvent exister:

* Les paquets réponse en attente: Ce sont des paquets dont la valeur n'a pas été encore calculée.

* Les paquets réponse complets: Ce sont des paquets dont la valeur a été calculée par l'unité de traitement et qui attendent d'être lus par l'UER pour être expédiés vers leur destinataire.

Les accès à cette mémoire comme le montre la figure 3.11b peuvent se faire:

- Soit en utilisant le champ nom de l'objet s'il s'agit d'une opération de mise à jour par l'unité de traitement.
- Soit alors par le champ état du paquet (paquet réponse complet), s'il s'agit d'opération de lecture par l'UER.

Là aussi, comme pour la MAM, les accès se font en spécifiant une partie du paquet accédé (le nom de l'objet ou l'état du paquet). Il est donc intéressant de réaliser la MAP par une mémoire associative.

Actuellement les mémoires associatives ne sont utilisées que rarement dans la construction des ordinateurs et en petite quantité (généralement pour réaliser le calcul d'adresse effective: table des segments,..etc). Pour cette raison les mémoires associatives existantes n'offrent que de très faibles capacités (quelques kilos octets). Les progrès technologiques dans le domaine des circuits VLSI, ainsi que le nombre d'applications de plus en plus important dans le domaine de l'intelligence artificielle (qui mettent en oeuvre un grand nombre d'opérations de recherche et de sélection d'une information parmi d'autres) nous promettent des mémoires associatives à capacité et prix intéressants dans un avenir proche [Hwan87a].

L'utilisation des mémoires associatives au niveau des noeuds (pour représenter l'organe de mémorisation) et l'application d'une fonction de hachage globale sur le nom de l'objet pour réaliser les opérations de mémorisation et de recherche d'un objet, nous permettent de considérer globalement la machine N-ARCH comme une mémoire associative à deux niveaux. Sa capacité est égale à la somme des capacités de toutes les mémoires des noeuds. Cette grande mémoire est découpée en blocs qui correspondent aux différents noeuds. Le choix d'un bloc pour réaliser une opération de lecture ou d'écriture se fait d'abord, par utilisation de fonctions de hachage (globale et locale), puis le choix d'un emplacement à l'intérieur du bloc se fait par un mécanisme associatif cablé (matériel).

Parallèlement à nos travaux sur la définition du noeud N-ARCH, des chercheurs du groupe se sont intéressés à la conception d'un circuit intégré d'une mémoire associative adaptée aux besoins de la machine N-ARCH. L'un des points originaux de cette mémoire est l'intégration de plusieurs ports de lecture/écriture/sélection ainsi que la possibilité de réaliser plusieures opérations simultanément. La pré-étude a permis de simuler cette mémoire et à montré la possibilité de réalisation d'un circuit

VLSI mémoire associative contenant 128 mots de 80 bits par chip. Ces circuits mémoires peuvent être rassemblées (ou mis en cascades) pour donner une plus grande capacité si nécessaire [Tour87]. Une étude est actuellement en cours afin de permettre une intégration de cette mémoire sous forme d'un circuit VLSI.

III.3 EXÉCUTION DES PROGRAMMES DÉCLARATIFS

Lorsque nous avons défini les unités fonctionnelles ainsi que les caractéristiques de base de la machine N-ARCH, des chercheurs du groupe ont commencé à étudier la mise en oeuvre de modèles d'exécution parallèles de langages fonctionnells et logiques.

III.3.1 Exécution parallèle de programmes FP [Deve88]

Le langage FP est un langage fonctionnel, dans lequel il n' y pas d'assignation ni de variable (langage fonctionnel pur) [Back78]. Un programme FP est construit à l'aide des 3 composantes:

1- Un ensemble d'objets: Un objet peut être un atome (comme 1, AB ou l'élément indéfini noté _|_) ou une liste d'objets comme < 1, 2, <1, AB>>.

2- un ensemble de fonctions qui comprend des fonctions primitives (+, -, car, cons, ..) et des formes fonctionnelles. Une forme fonctionnelle est une fonction qui a comme argument une fonction ou un objet et dont le resultat est une fonction.

3- L'opérateur application qui transforme un objet en un autre objet par utilisation d'une fonction.

Dans le modèle d'exécution défini, le programme est représenté par un graphe appelé graphe fonctionnel. Les noeuds de ce graphe représentent les fonctions à appliquer sur la liste des arguments, ils sont appelés noeuds fonctionnels. La liste des arguments est aussi représentée par un graphe (liste argument). La figure 3.12 donne le graphe fonctionnel et le graphe correspondant à la liste des arguments du programme FP qui réalise le produit interne de 2 listes.

Ou IP = (/ +) o (AA *) o trans. Nous avons donc

IP < < 2, 6, -2>, < 3, -1, 5> > = ((/ +) o (AA *)) < <2, 3>, <6, -1>, <-2, 5> >

= (/ +) <6, -6, -10>

= 6 + ((-6) + (-10)) = -10

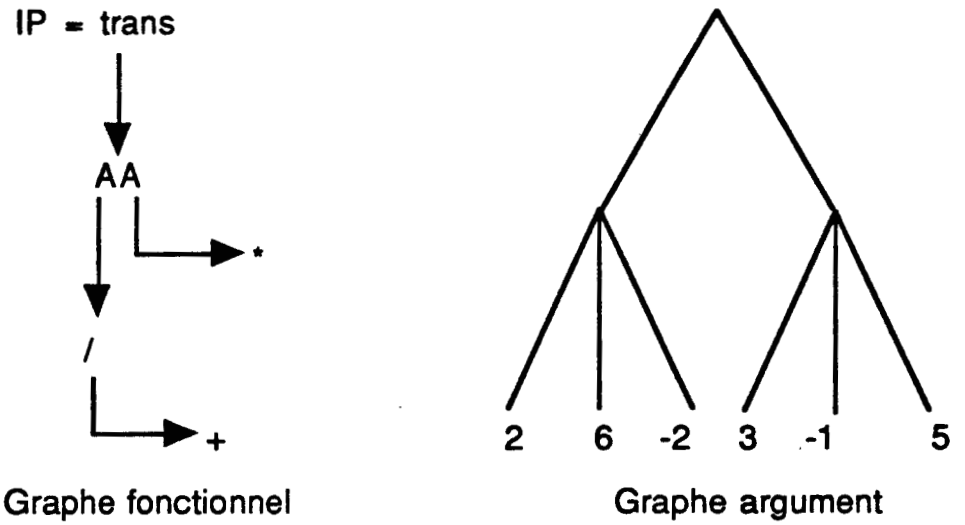


Fig 3.12 : Le graphe fonctionnel et
la liste argument du programme
produit interne

L'exécution du programme FP sur la liste des arguments est réalisée en explorant en parallèle les différentes branches du graphe fonctionnnel de façon à distribuer les fonctions primitives sur les objets de la liste des arguments (fig 3.13). L'exploration des différentes branches du graphe fonctionnel ainsi que le mécanisme de réduction de la liste des arguments sont réalisés en parallèle par émission/réception de messages.

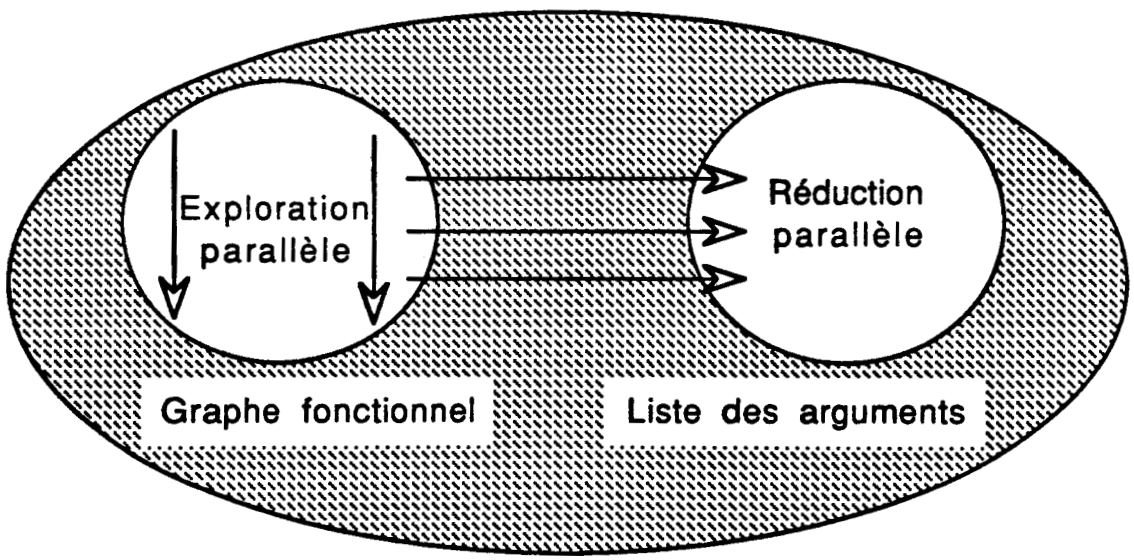


Fig 3.13: Evaluation du graphe fonctionnel

Dans ce modèle le graphe fonctionnel n'est pas modifié par les opérations de réduction. Ceci permet un partage du code de la fonction (ou

de la forme fonctionnelle) s'il y a plusieurs appels à une même fonction. La liste des arguments est, par contre, modifiée à chaque étape de l'opération de réduction.

Dans le langage FP deux formes fonctionnelles peuvent être exploitées pour permettre un parallèlisme dans le traitement. Ces deux formes fonctionnelles sont "Apply to All" notée AA et la contruction notée [ ].

La forme fonctionnelle AA est définie par:

AA f : x =

si x est une liste vide, le résultat est une liste vide,

si x = <x1, x2, ...xn>, le résultat est la liste dont les éléments sont < f:x1, f:x2, ...f:xn>,

autrement le résultat est _|_.

Comme on peut le voir la forme fonctionnnelle AA peut être réalisée en appliquant f simultanément sur tous les éléments de la liste arguments.

La forme fonctionnelle construction est définie par

[f1, f2, ...fn] : x = <f1:x, f2:x, f3:x, ...>

Si on suppose que n copies (c1, c2, c3, cn) de la liste-argument x existent, les fonctions fi peuvent être appliquées simultanément sur les différents ci.

D'autres formes fonctionnelles ont été introduites dans le modèle pour accroitre le degré du parallèlisme exploité du programme FP à exécuter. Une description détaillée du modèle de traitement peut être trouvée dans [Deve88].

III.3.2 Exécution de programme PROLOG sur N-ARCH [Hann88]

Les langages de la programmation logique (comme PROLOG) contiennent de très grande potentialité d'exécution parallèle. Le projet japonais visant la construction de la machine appelée "machine de la cinquième génération" fut sans doute le premier projet dont le but était justement l'exécution d'un langage de la programmation logique sur une machine multiprocesseurs [Mura84].

Le modèle d'exécution parallèle du langage Prolog qui a été défini dans le cadre du projet N-ARCH est un modèle controlé par nécessité (réduction de graphes) et qui exploite le parallélisme OU et le parallélisme ET. Afin d'éviter une charge système importante causée par la gestion de

la grande quantité d'informations lors d'une exploration en largeur d'abord de l'arbre ET/OU, notre modèle ne permet l'activation que d'un nombre restreint de branches au niveau de cet arbre (parallélisme OU restreint). Ainsi, le modèle explore l'arbre ET/OU en profondeur d'abord et de gauche à droite, tout en permettant une exploration parallèle des branches OU les plus profondes. Par conséquent nous avons un parallélisme OU qui croit à partir des feuilles de l'arbre, au lieu d'être activé à partir de la racine.

La figure 3.14 donne un exemple de programme PROLOG. L'arbre ET/OU correspondant est donné dans la figure 3.15.

```
Gpar(X, Y) :- Par(X, Z), Par(Z, Y).
Par(X, Y) :- Père(X, Y).
Par(X, Y) :- Mère(X, Y).
Père(jean, luc).
Père(luc, paul).
Mére(anne, paul).
Mère(marie, luc).
:- Gpar(jean, Y)
```

Fig 3.14 : Un programme PROLOG

Lors de la traversée de l'arbre ET/OU seulement une alternative est activée (la plus à gauche, comme le noeud 3 de la figure 3.15) . Une fois que la première solution a été retournée par un noeud OU (fils gauche) à son noeud père (noeud 3), le deuxième noeud (noeud 6) est alors activé de manière bloquée (positionnement d'un verrou).

Un noeud activé bloqué (ou verrouillé) va évaluer le sous arbre ET/OU dont il est la racine sans retourner de solution au noeud père. Lorsque le fils gauche (noeud 5) a renvoyé toutes ces solutions, le noeud qui était activé bloqué (noeud 6) devient activé débloqué (déverrouillé) et peut alors expédier ses solutions à son noeud père. Cette solution permet d'éviter l'accumulation des données (solutions) au niveau du père.

Afin d'éviter les conflits de liens entre 2 sous-buts qui partagent une même variable, le parallèlisme ET est géré en pipeline.

Comme on peut le voir, les 2 types de parallélisme exploités dans le modèle permettent de controler la quantité de branches OU et ET explorées par un mécanisme de verrouillage/déverrouillage. Ce

mécanisme permet aussi de préserver la sémantique opérationnelle d'un interpréteur PROLOG conventionnel, car les solutions sont retournées en commençant par la plus à gauche dans l'arbre ET/OU.

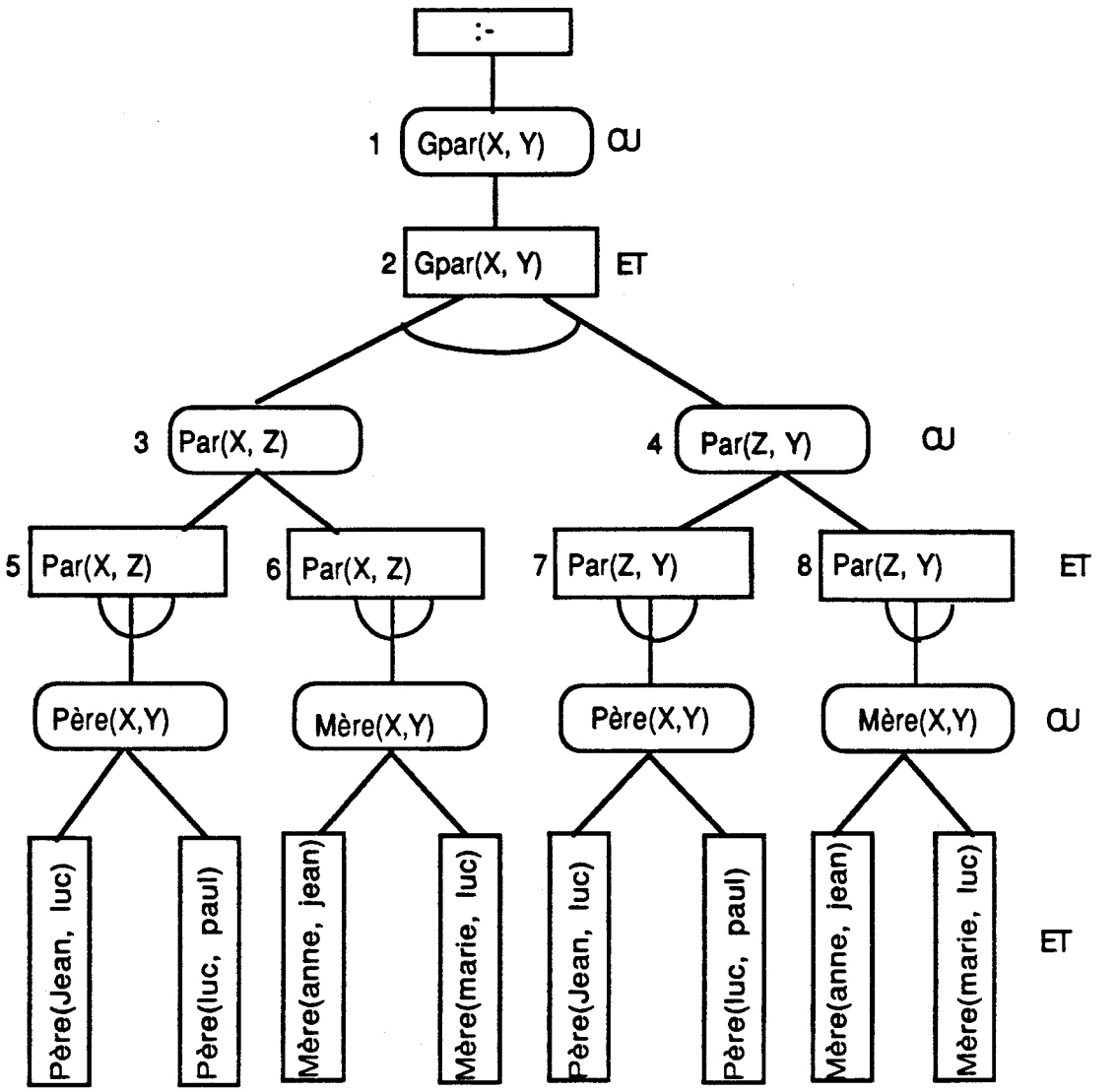


Fig 3.15 : L'arbre ET/OU correspondant au programme de la figure 3.14

Des simulations sont en cours de mise au point afin de valider et étendre le modèle.

Références citées dans le chapitre :

[Back78], [Bhuy82], [Deve88], [Hann89], [Hwan87a], [Hwan87b], [Kenn83], [Mura84], [Tour86], [Tour87].

CHAPITRE IV
EMULATION DE LA MACHINE N-ARCH SUR UN RESEAU DE TRANSPUTERS

IV.1 INTRODUCTION: POURQUOI UN EMULATEUR ?

Dans le chapitre précédent nous avons présenté les caractéristiques générales de la machine N-ARCH. Nous avons vu quelles étaient les unités fonctionnelles d'un noeud ainsi que les interactions entre elles.

Une fois l'architecture de la machine définie, il est alors nécessaire de disposer d'outils afin de réaliser des tests sur la validité de l'architecture, pour mettre au point certains détails de l'implantation, et enfin pour évaluer les performances. L'outil recherché doit nous permettre de produire une représentation de la machine, possédant les caractéristiques suivantes:

1- Facilité de mise en oeuvre.
2- Très proche de la machine réelle.
3- Aisément modifiable afin de tester les différentes implantations possibles.

De façon générale, il existe 3 manières de réaliser la représentation recherchée [Heid87].

IV.1.1: Par simulation

Une machine peut être caractérisée par un ensemble de ressources et un ensemble de tâches (processus). Ainsi une manière de représenter le système serait par des files d'attente de demandes aux ressources. Le rôle de chaque processus serait de gérer les demandes qui arrivent à la ressource correspondante. Ceci revient à écrire un programme qui est composé d'un ensemble de processus concurrents qui représentent les unités fonctionnelles de la machine (qui correspondent aux noeuds de N-ARCH) et un processus de contrôle.

Chaque processus concurrent peut monopoliser l'unité centrale de la machine (UC) pour réaliser un traitement donné. L'UC est alors relâchée par un processus soit lorsque le traitement est terminé ou lorsqu'un

événement se produit. La synchronisation et la communication (s'ils ont lieu) entre les différents processus peuvent se faire de plusieurs manières (Rendez-vous, Sémaphores, Boites aux lettres, émissions réceptions de messages...etc). Le processus de contrôle est le processus permettant de collecter les informations qui intéressent le concepteur. Ainsi, à des intervalles réguliers, le processus de contrôle interroge chacun des processus concurrents pour avoir un certain nombre de renseignements, tel que: l'état du processus (bloqué, en attente de l'UC, a le contrôle de l'UC, ...), le nombre de paquets émis, le nombre de paquets dans les files d'attente ,...etc.

L'avantage d'utiliser le simulateur comme outil de test est la possibilité d'exploiter les moyens de la machine sur laquelle se fait la simulation. Ainsi le concepteur peut concentrer son effort sur les points qu'il juge intéressants, sans se soucier d'un certain nombre de détails qui n'ont pas une grande importance dans l'étape actuelle de la conception (ou de la validation). De plus, le simulateur peut facilement subir des modifications pour représenter de nouveaux modes de fonctionnement et tester ainsi, les différentes solutions pour un même problème. L'écriture du simulateur se fait généralement en utilisant un langage de haut niveau.

L'inconvénient dans l'utilisation d'un simulateur est, bien sûr, le manque de fiabilité. En effet il est impossible de prévoir les performances exactes de la machine par l'utilisation d'un simulateur.

IV.1.2: Par la réalisation physique (Le prototype)

Afin de connaitre les performances réelles d'une machine, la construction d'un prototype est sûrement la solution la plus fiable. Le prototype est composé d'un ou plusieurs circuits imprimés, contenant chacun un certain nombre de composants électroniques (circuits intégrés, résistances, ...) que le concepteur a choisi pour réaliser le prototype et des connections entre les différents composants. Le développement de systèmes et de logiciels pour la conception assistée par ordinateurs (CAO) et les progrès enregistrés dans la réalisation des circuits dédiés (custom and semi-custom circuits) ont permis à cette méthode d'être de plus en plus utilisée pour la conception de machines. Une fois ce prototype réalisé, le concepteur a une idée exacte des performances de la machine construite.

Les tests dans ce cas, peuvent être réalisés soit par des sondes physiques (Hardware probes) ou par des sondes logiques (Software or Micro-code probes). Les sondes physiques sont des lignes électriques de haute impédance, qui sont connectées à la machine. Les états logiques de ces lignes sont traités par un instrument (systèmes de développement ou analyseur logique) qui permet par exemple de montrer le contenu des registres, l'état des bus, de compter le nombre de fois où un événement s'est produit. Les sondes logiques sont des instructions ajoutées au programme que la machine va exécuter, ceci afin de collecter des informations sur son déroulement. Dès qu'un événement déterminé se produit (exemple: exécution d'une instruction, accès à une position mémoire ...etc) une routine d'interruption -la sonde logique- est exécutée afin de récupérer des informations qui intéressent le concepteur. Remarquons enfin que les deux types de sondes peuvent être utilisées afin d'avoir des informations de types différents.

Comme on peut le voir, l'avantage d'utiliser le prototype pour réaliser les opérations de mesures de performances et de tests est le grand degré de fiabilité (100% si uniquement des sondes physiques sont utilisées). Les inconvénients de l'utilisation du prototype comme outil de mise au point sont:

1-La rigidité du système (manque de souplesse dans les modifications).
2-La nécessité de matériels spécialisés pour la réalisation du prototype et des tests.
3- Le temps relativement long pour la mise en oeuvre.

IV.1.3 : Par Emulation

La méthode de test et de mise au point, par émulation représente un compromis entre la méthode de simulation et celle par la réalisation physique. L'émulateur est une machine qui fonctionnellement possède les mêmes caractéristiques que la machine à concevoir. Il est composé d'une partie matérielle possédant des points communs avec la machine à concevoir, et d'une partie logicielle qui complète la partie matérielle, en simulant par des programmes les fonctions qui n'existent pas au niveau du matériel. La partie logicielle est couramment appelée "Noyau de l'émulateur ". Ainsi plus les caractéristiques matérielles de l'émulateur

sont proches de la machine envisagée plus grand est le degré de fiabilité, et moins grand est le degré de souplesse, et vice-versa. L'émulateur d'une machine parallèle, comme la machine N-ARCH, doit contenir plusieurs processeurs. L'existence d'un parallélisme réel permet d'éliminer l'overhead associé à la gestion des processus concurrents dans une machine séquentielle et réduit le temps de simulation. Il est possible d'avoir, par conséquent, une idée plus exacte sur le gain du temps d'exécution obtenu par la multiplication du nombre de processeurs.

Les avantages de l'utilisation de l'émulateur comme outil de test et de mesure de performances sont:

1-Le relatif degré de fiabilité dû à l'existence de la partie matérielle.

2-Une certaine facilité dans les modifications des fonctions de la machine.

La simulation de certaines fonctions de la machine par des programmes permet de tester les différentes possibilités d'implémentations de ces fonctions.

Lorsque nous avons commencé ce travail, le laboratoire d'informatique c'est doté d'un réseau de plusieurs processeurs appelé TRANSPUTERS. Comme on le verra dans les chapitres suivants, l'utilisation d'un réseau de Transputers pour émuler la machine N-ARCH offre, en plus des avantages énumérés précédemment, une grande facilité de mise en oeuvre. Tous ces facteurs, nous ont conduit à choisir cette méthode pour réaliser l'outil de mise au point et de mesures de performances pour la machine N-ARCH.

IV.2 PRESENTATION DU LANGAGE OCCAM ET DU TRANSPUTER

IV.2.1: Le langage OCCAM: Présentation

La construction d'un nombre de plus en plus important de machines multiprocesseurs-rendu possible grâce à de nouvelles technologies VLSI-a provoqué un grand besoin en langages adaptés à ce nouveau type de matériel.

L'une des possibilités envisagées, est la conception de langages intégrant des primitives permettant au programmeur d'exprimer directement le parallélisme dans son programme(parallélisme explicite). Le choix de ces primitives doit respecter au moins 2 critères:

1-Facilité d'utilisation par le programmeur.

2-L'implémentation de ces primitives ne doit pas détériorer les capacités de la machine multi-processeurs, tout en offrant des performances acceptables sur machine mono-processeur.

Le langage OCCAM satisfait à chacune de ces conditions.

IV.2.2: Les concepts de base dans OCCAM [Burn88][Poun87]

IV.2.2.1 Le processus

Le concept le plus important en OCCAM est le processus. En effet un programme OCCAM est tout simplement un ensemble de processus. La construction d'un processus en OCCAM est rendu possible grâce à 3 processus élémentaires (l'affectation, l'émission, et la réception de messages), et à 5 constructeurs de base (SEQ, PAR, WHILE, IF, ALT). Ces constructeurs combinent des processus de base pour en former d'autres de niveau hiérarchique plus élevé.

Les processus de base en OCCAM sont:

-L'affectation:

Le processus d'affectation transfère la valeur d'une expression à une variable. Ceci est noté par

V := E

Ou V est une variable, et E une expression.

-La réception :

Le processus de réception transfère une valeur lue sur un canal à une variable. Une opération de réception sur le canal C vers la variable V est notée:

C ? V

Le processus réception est mis en état d'attente jusqu'à ce qu'une émission utilisant le même canal soit prête à s'exécuter en parallèle avec l'opération de réception. Ainsi le processus réalisant une opération de réception sur un canal doit être contenu dans une construction parallèle qui contient aussi le processus qui réalise l'opération d'émission sur ce canal. De plus un seul processus d'un constructeur parallèle peut réaliser des opérations de réception et un seul autre peut réaliser des opérations d'émission sur un canal déterminé. En d'autres termes, un canal appartient toujours à 2 processus parallèles, l'un pouvant faire

uniquement des opérations de réception et l'autre uniquement des opérations d'émission.

-L'emission:

Le processus d'émission transfère vers un canal la valeur d'une expression. De même que pour le processus de réception, le processus d'émission attend jusqu'à ce qu'un processus de réception soit prêt. L'émission d'une expression évaluable E sur un canal C est notée :

```
C!E
```

Il est intéressant de noter que le mécanisme d'émission et de réception de messages en OCCAM (basé sur le modèle des processus séquentiels communiquants ou "CSP") permet de résoudre les deux problèmes liés à la programmation concurrente, qui sont la communication et la synchronisation.

IV.2.2.2 Les constructeurs

Le rôle d'un constructeur est de permettre une hiérarchisation des processus élémentaires qui peuvent être regroupés en un seul processus.

Il existe en OCCAM 5 constructeurs :

-Le constructeur de processus séquentiel SEQ
-Le constructeur alternatif ALT
-Le constructeur répétitif WHILE
-Le constructeur conditionnel IF
-Le constructeur parallèle PAR

Le dernier constructeur nous permet d'exécuter 2 processus, ou plus, de manière concurrente.

```
PAR
    process1
    process2
    .
    .
    processn
```

Les sous processus d'un processus PAR peuvent être soit complètement indépendant les uns des autres, ou bien contenir des communications par l'utilisation des canaux. Les canaux sont, ainsi, le seul moyen de communication autorisé entre 2 processus concurrents (il n'y a pas de

notion de variable globale). Nous avons déjà indiqué que 2 et 2 processus seulement peuvent utiliser un canal, l'un réalisant des opérations d'émission et l'autre de réception. De cette façon, pour réaliser une communication bidirectionnelle, 2 canaux distincts doivent être utilisés. Le processus PAR est terminé lorsque tous ces sous processus ont été exécutés.

IV.2.3: Implémentation du langage OCCAM sur TRANSPUTER [Inmos87]

IV.2.3.1 Présentation du processeur intégré Transputer

Le développement du langage OCCAM a été énormément facilité par la conception du processeur Transputer chez le constructeur INMOS.

Le Transputer intègre des capacités de traitement ainsi que des fonctions d'entrée/sortie(E/S). La figure 4.1 illustre l'architecture générale du Transputer.

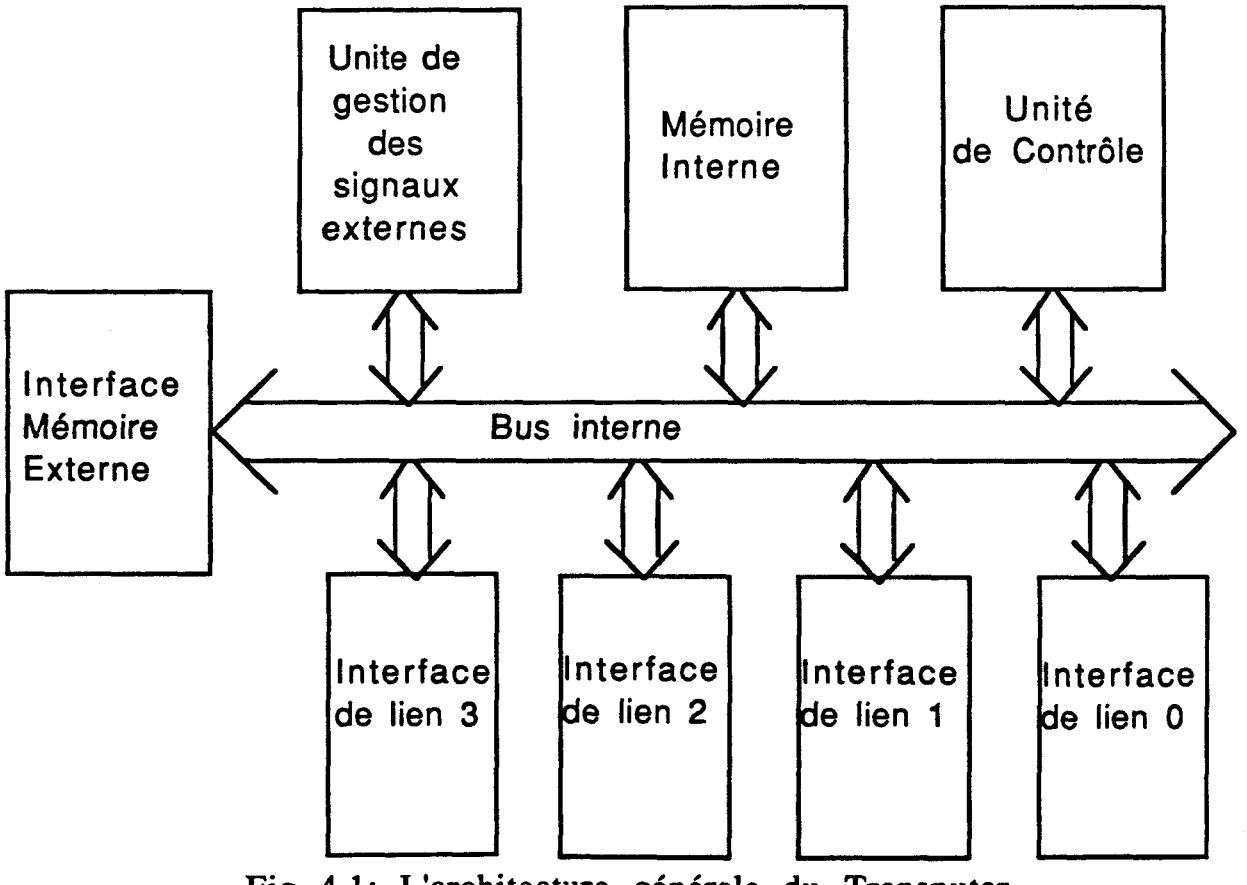


Fig 4.1: L'architecture générale du Transputer

On peut remarquer ainsi l'existence de l'unité de contrôle et de traitement (CPU), de 4 liens d'E/S et d'une mémoire de 2 Kilo-octets. Les liens du Transputer sont gérés par des unités autonomes vis à vis du CPU. Elles sont appelées interfaces d'E/S. Ces interfaces peuvent ainsi réaliser des opérations d'accès direct mémoire et permettent par la même occasion au Transputer de réaliser des opérations d'E/S en même temps qu'un traitement local.

Les opérations de transmission de messages se font en série (émission ou réception bit par bit), en mode asynchrone et une vitesse de 10 Mega-bits par seconde. Les 2 kilos octets de mémoire interne du Transputer peuvent être étendu jusqu'à 4 Giga-octets de mémoire par le moyen d'un bus externe.

D'après les notes techniques fournies par le constructeur, le Transputer peut exécuter 10 méga-instructions (du type RISC) par seconde [Burn88] [Silb88]. Les instructions du Transputer sont d'un codage très simple, en nombre réduit (16 code-opérations), et ont un format unique (8 bits).

Malgré l'existence d'un jeu d'instructions d'assemblage, le niveau le plus bas dans lequel le Transputer peut être programmé est le niveau OCCAM.

IV.2.3.2 Implémentation des processus séquentiels et parallèles sur le Transputer

Le Transputer dispose de 6 registres pour implémenter les processus séquentiels, auquels 2 sont rajoutés pour l'implémentation des processus parallèles. La figure 4.2 donne une idée de l'utilisation des 6 premiers registres pour manipuler des processus séquentiels. Les registres A,B,C sont utilisés comme pile d'évaluation. Le registre opérande a la fonction d'un accumulateur dans une machine classique.

Un processus OCCAM peut être dans l'un des 3 états suivants:

-suspendu: un processus suspendu est en attente d'une émission ou d'une réception sur un canal.

-en cours d'exécution(a le contrôle de la CPU).

-exécutable(en attente de la CPU).

Dans chaque Transputer du réseau, il y a un seul processus en cours d'exécution. Les processus exécutables sont maintenus dans une liste chaînée. Pour cela on utilise 2 registres supplémentaires (tête et queue).

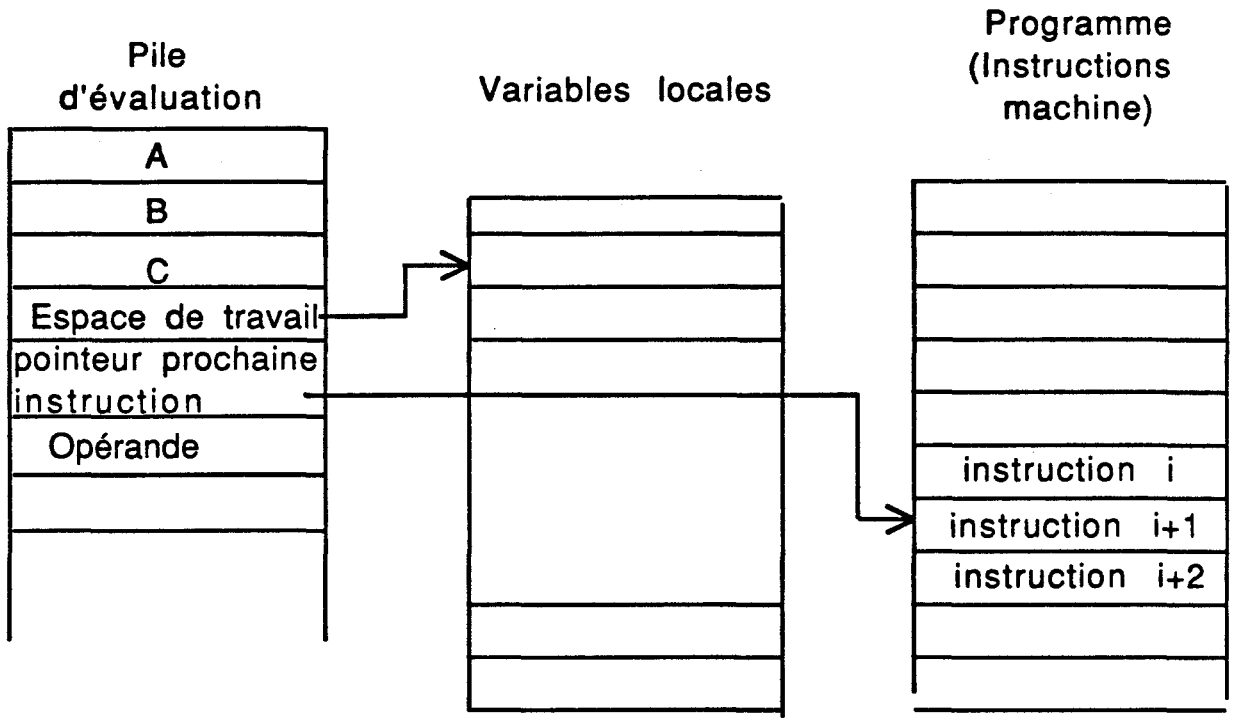


Fig 4.2: Exécution d'un processus séquentiel

La figure 4.3 montre un exemple d'utilisation de ces 2 registres. Dans cette figure les processus P, Q, R, S et T sont exécutés en parallèle. Le processus T est suspendu, P, Q et S sont à l'état exécutable et R est en cours d'exécution.

Lorsqu'un processus suspendu devient exécutable, la liste chaînée des processus exécutables est alors mise à jour afin que le registre queue pointe vers l'adresse début de l'espace de travail du processus en question. Par conséquent, l'exécution des processus parallèles induit un overhead en temps qui est de l'ordre de la micro-seconde. Ce temps est nécessaire pour réaliser les changements de contexte.

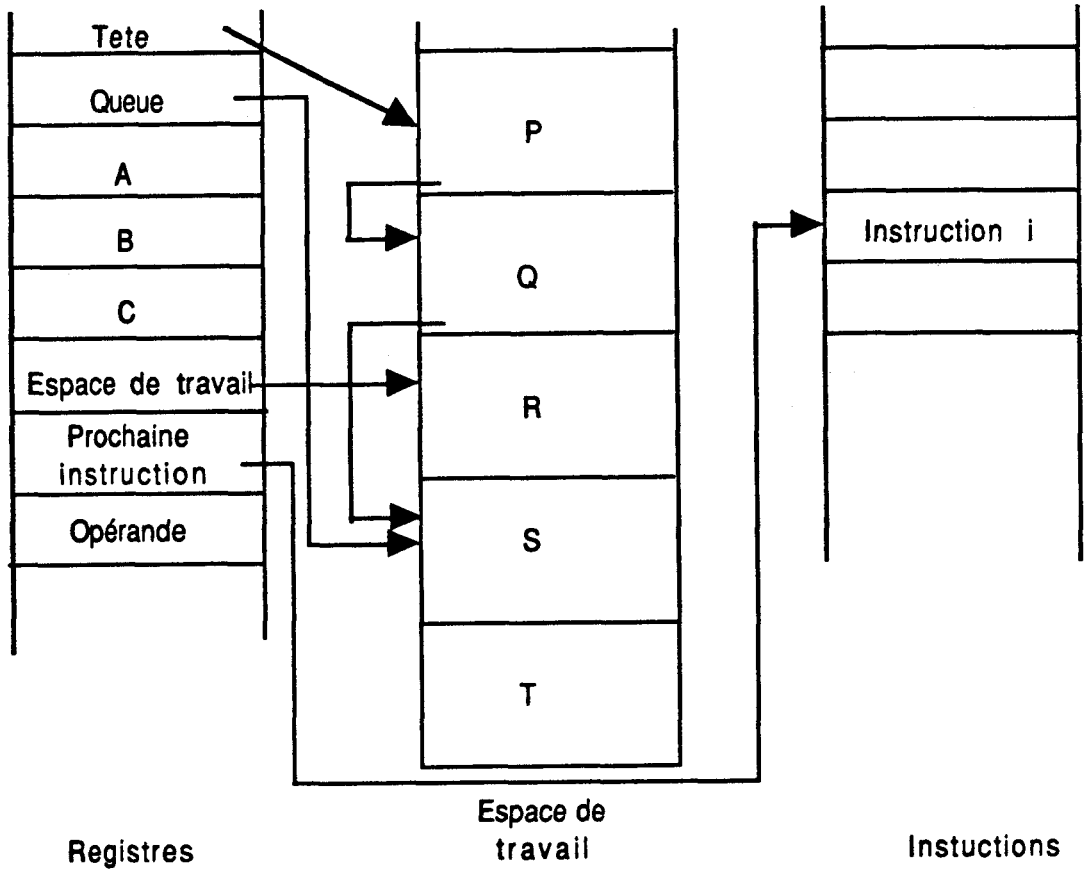


Fig 4.3 : Exécution d'un processus parallèle

IV.2.3.3 La priorité des processus

Dans le Transputer il existe deux niveaux de priorité (haut et bas). Si un processus de niveau haut est exécutable, il est alors plus prioritaire que tous les processus exécutables de niveau bas. Par conséquent son exécution se poursuit jusqu'à ce qu'il soit terminé ou en attente d'une opération d'émission ou de réception. En contre partie les processus de niveau bas sont gérés par un allocataire (scheduler) qui cycliquement donne le contrôle de la CPU à chaque processus de niveau bas (allocation par temps partagé). Cette CPU est relâchée lorsqu'une de ces conditions soit réalisée

-Le processus est terminé ou en attente.

-Un processus de niveau haut est devenu exécutable.

-Il a été exécuté pendant une période supérieure à une tranche de temps (time slice) et il a atteint un point d'arrêt (émission ou réception sur un canal).

Dans le TRANSPUTER, la valeur de la tranche de temps est de l'ordre de 820 micro-secondes.

IV.2.3.4 Implémentation des canaux

-Les canaux internes

Lorsque 2 processus parallèles utilisant un canal sont exécutés sur un même Transputer, le canal est dit canal interne et il est alors représenté par un mot mémoire (le mot canal). La figure 4.4 montre un exemple d'utilisation d'un canal interne. Les processus P et Q sont exécutés en parallèle sur le même processeur.

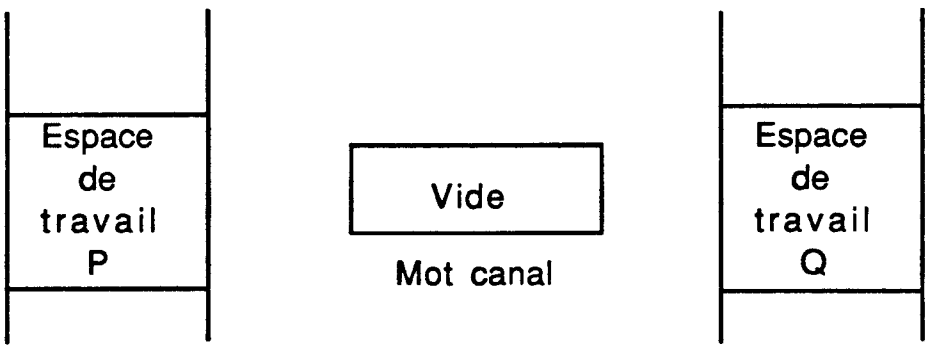


Fig 4.4: Un canal interne vide

-Les canaux externes

Si les deux processus s'exécutant en parallèle se trouve sur 2 Transputers différents, les canaux utilisés correspondent à des canaux externes (canaux physiques). Le contrôle des communications sur un canal externe est réalisé par les 2 interfaces de liens qui se trouvent sur les deux Transputers. Chaque interface dispose de 3 registres: Un pointeur vers l'espace de travail du processus qui utilise le canal, un pointeur vers la zone mémoire qui contient le message à émettre ou à recevoir et un compteur d'octets. Ainsi, dès que l'opération de transfert est terminée, le contrôleur de lien va se charger d'insérer le processus qui vient de réaliser l'opération de transfert - il est devenu exécutable- dans la liste chaînée des processus exécutables.

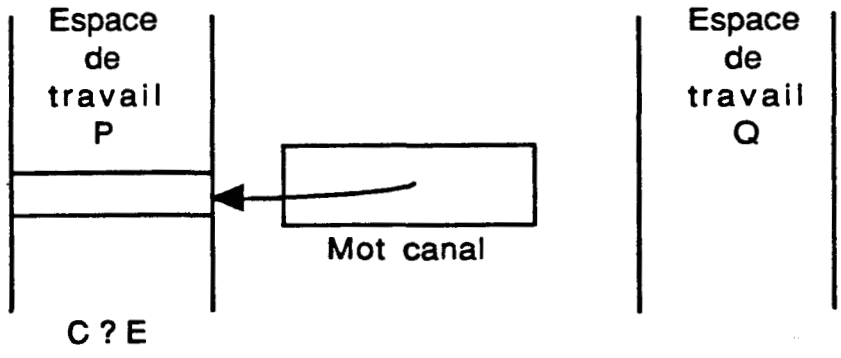


Fig 5.a

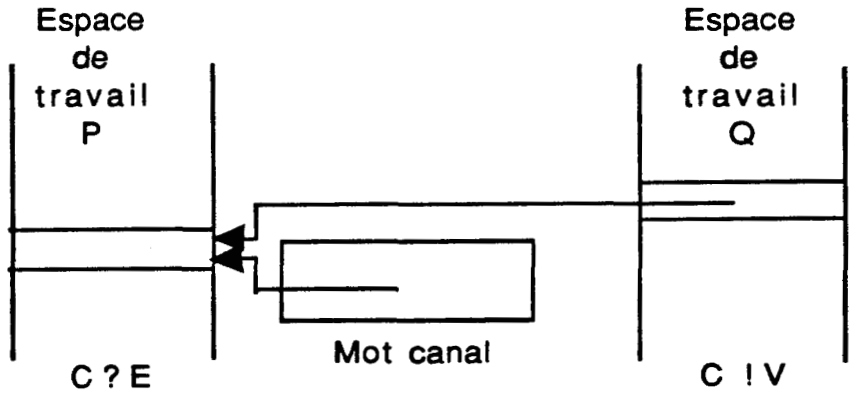


Fig 5.b

Fig 4.5: Un canal interne réalisant une opération de communication

IV.2.4: Conclusion

L'utilisation du langage OCCAM et du Transputer comme outils d'émulation pour la machine N-ARCH est intéressante pour plusieurs raisons:

1- OCCAM est un langage qui offre une grande facilité d'utilisation pour l'exécution de programmes parallèles. Par ailleurs, OCCAM peut être très bien utilisé pour la description de matériels.

2- Les caractéristiques matérielles du Transputer sont très intéressantes (mémoire interne rapide, interfaces d'E/S autonomes, vitesses de transmission de 10 Mega-bits/seconde,...) en vue de la construction d'une machine parallèle [Munt88]. De ce fait, le Transputer est très largement

utilisé pour émuler des machines parallèles comme la machine ZAPP [Burn86] ou la machine ALICE [Crip86].

3-Il existe entre l'architecture du Transputer et celle du noeud N-ARCH un grand nombre de points communs comme:

-L'existence de contrôleurs d'opérations d'E/S autonomes (interfaces de liens).

-Le mode de communication entre processeurs par messages.

Nous pouvons regretter néanmoins l'absence de mémoire associative, ceci nous amènera à simuler cette mémoire par un processus OCCAM.

IV.3 LES PAQUETS DE L'EMULATEUR [Niar87]

L'émulateur de la machine N-ARCH manipule 3 types de paquets :

IV.3.1: Les paquets initialisation:

Ces paquets sont générés et distribués dans le réseau afin de charger le programme en équilibrant la charge entre les différents processeurs. Le programme écrit dans un langage déclaratif est converti en un ensemble de paquets initialisation. Cette phase de transformation, qui correspond à la phase de compilation dans un système classique, est réalisée par la machine hôte en utilisant la fonction de hachage globale pour déterminer pour chaque paquet initialisation le noeud qui le mémorisera.

Comme nous le verrons plus loin, la première version du noyau n'intègre pas l'évaluation des fonctions. Le programme est donc composé uniquement d'expressions et la répartition du programme peut se faire de manière statique (static load balancing).

Un paquet initialisation est composé des champs suivants:

*Son type: qui est soit *initialisation en routage* (IR) si le paquet n'a pas encore atteint son noeud destination, ou *initialisation en collision* (IC) s'il l'a atteint.

*Le nom de l'expression, qui représente la partie gauche au niveau de l'expression

*Le numéro du noeud destination du paquet initialisation qui est obtenu par application de la fonction de hachage globale sur le nom l'expression.

*L'état de l'expression qui peut être: calculée, calculable ou au repos.

*La longueur de l'expression: qui représente le nombre d'éléments de la partie droite de l'expression.

*Une chaîne d'éléments de longueur variable. Chaque élément est composé d'un mot et d'un octet. Ce dernier représente la marque du mot correspondant. En effet, comme N-ARCH contient des mémoires marquées (tagged memories), on associe à chaque mot de l'expression une marque qui est représentée par la plus petite entité que l'on peut accéder en OCCAM c'est à dire l'octet. Trois types de marques existent:

a- Identificateur: Le mot associé est le nom d'une expression.

b- Valeur: Le mot correspondant est une valeur.

c- Opérateur: Le mot associé représente un opérateur(+,-,*,...)ou une fonction prédéfinie du langage(CONS,CAR,...).

La figure 4.6 montre un exemple simple d'un programme réparti sur 4 noeuds. La fonction de hachage globale utilisée est, dans ce cas, le reste de la division par 4 de la valeur ASCII du nom de l'expression.

A = (B*C) - D
B = C + 2
C = 3
D = 5

Nom	Valeur ASCII	Fonction de hachage
A	32	0
B	33	1
C	34	2
D	35	3

IR	A	0	5	-	*	B	C	D
				O	O	N	N	N

IR	B	1	3	+	C	2
				O	N	V

IR	C	2	1	3
				V

IR	D	3	1	5
				V

Fig4.6: Répartition d'un programme sur un réseau de 4 noeuds

IV.3.2: Les paquets requête:

Afin d'évaluer les arguments contenus dans une expression, un noeud devra générer pour chaque argument un paquet requête (évaluation par nécessité). Ce dernier contient les champs suivants.

*Son type: qui est soit *requête en collision* (RC) si le paquet a atteint son noeud destination, ou *requête en routage* (RR) s'il ne l'a pas atteint.

*Le nom de l'expression demandée.

*Le numéro du noeud destination obtenu par application de la fonction de hachage globale sur le nom de l'expression demandée.

*Le nom de l'expression génératrice de la requête.

*Le numéro du noeud qui contient l'expression génératrice de la requête. Cette information est utilisée par le noeud qui va générer le paquet réponse correspondant.

Ceci permet, ainsi, un envoi direct de la réponse vers le noeud destinataire (pas de collision pour les paquets réponse).

Dans l'émulateur, tous ces champs sont représentés par un mot de 32 bits.

A titre d'exemple, le paquet requête pour évaluer l'expression "B" généré lors de l'évaluation de l'expression "A" dans la figure 4.6, a le format suivant :

RR	B	1	A	0

IV.3.3: Les paquets réponse

Pour chaque paquet requête reçu, le noeud devra générer un paquet réponse lorsque l'expression demandée devient évaluée. Un paquet réponse contient les champs suivants:

*Le type: qui est toujours *réponse en routage* (R)

*Le nom de l'expression réceptrice de la réponse. C'est celle qui a généré la requête correspondante.

*Le numéro du noeud destination. C'est le noeud qui contient l'expression réceptrice.

*Le nom de l'expression évaluée (le nom de l'argument qu'il faut mettre à jour).

*Le résultat de l'évaluation: Ce champ peut représenter soit:

- La valeur de l'expression demandée si le mode de contrôle est de la réduction de graphes ou contrôlé par les données (Data Flow).

- La totalité de l'expression demandée s'il s'agit d'une réduction de chaînes.

Pour les raisons déjà citées, nous utiliserons le mode de contrôle par réduction de graphes, et par conséquent ce champ contient la valeur de l'expression demandée.

Le paquet réponse retourné comme réponse au paquet requête donné comme exemple dans le paragraphe précédent est

R	A	0	B	5

IV.4 LES COMPOSANTS DU NOYAU: LES PROCESSUS OCCAM DE N-ARCH [Gonc87][Niar87]

Le noyau de l'émulateur pour la machine N-ARCH est composé de plusieurs processus OCCAM qui sont exécutés en parallèle et qui représentent les unités fonctionnelles de la machine. Cette décomposition en plusieurs processus n'est pas la solution la plus simple pour la conception du noyau, mais offre les avantages suivants:

1- Elle permet d'éviter le temps perdu durant les communications inter-noeud (élimination des attentes actives). Ceci est dû au fait que les communications dans le réseau sont asynchrones.

2- Les possibilités matérielles du Transputer sont exploitées en autorisant le traitement des opérations d'E/S en parallèle avec les opérations de traitement.

3- Cette décomposition peut être exploitée lors de la réalisation VLSI du noeud.

La figure 4.7 montre les 8 processus parallèles qui composent le noyau N-ARCH.

Cette figure peut être traduite en OCCAM par le processus suivant:

```
PAR
    PROCESSUS.ENTREE(.....)
    PROCESSUS.BUFFER.ENTREE(.....)
    PROCESSUS.CONTROLE(.....)
    PROCESSUS.MEMOIRE.ASSOCIATIVE.PROGRAMMES(.....)
    PROCESSUS.MEMOIRE.ASSOCIATIVE.PAQUETS(.....)
    PROCESSUS.EXECUTION(.....)
    PROCESSUS.BUFFER.SORTIE(.....)
    PROCESSUS.SORTIE(.....)
```

Les paramètres de chacun des processus précédents - représentés par (.....) - correspondent en fait aux différents canaux qui permettent la communication entre les processus du noyau. A l'exception des 4 canaux d'entrée et des 4 canaux de sortie utilisés par le processus d'entrée et le processus de sortie, tous les autres canaux sont des canaux internes au Transputer (voir paragraphe IV.2.3.4).

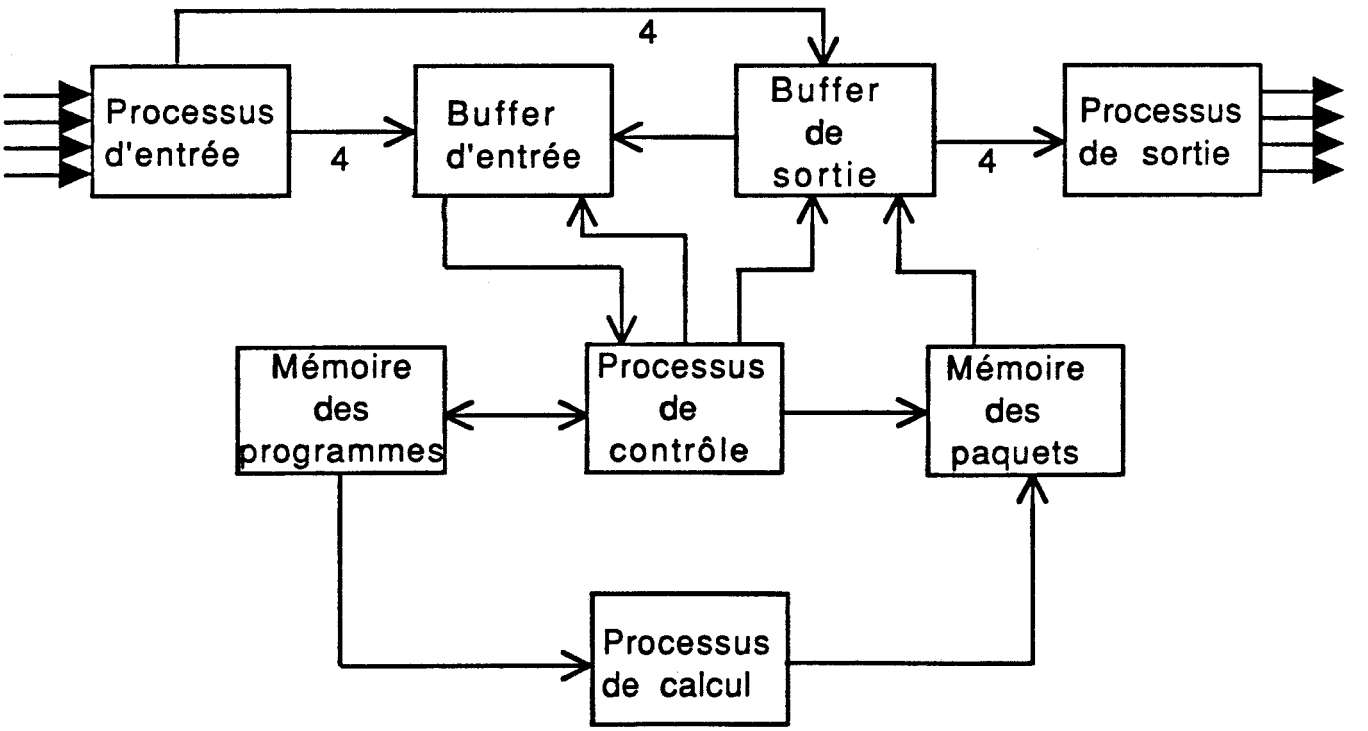


Fig 4.7: Composition du noyau

Remarques:

Pour des raisons de performances au niveau du temps d'exécution, nous avons accordé aux processus d'entrée et de sortie la priorité haute [Atki87]. En d'autres termes le noyau a la forme suivante:

```
PRI PAR
    PAR
        PROCESSUS.ENTREE(.....)
        PROCESSUS.SORTIE(.....)
    PAR
        PROCESSUS.BUFFER.ENTREE(.....)
        PROCESSUS.CONTROLE(.....)
        PROCESSUS.MEMOIRE.ASSOCIATIVE.PROGRAMMES(.....)
        PROCESSUS.MEMOIRE.ASSOCIATIVE.PAQUETS(.....)
        PROCESSUS.EXECUTION(.....)
        PROCESSUS.BUFFER.SORTIE(.....)
```

Les tests réalisé ont montrés que cette configuration permet un gain de temps et une meilleure exploitation des interfaces de liens. En effet, si un Transputer reçoit un paquet qui est attendu par un autre noeud, il est intéressant de transmettre ce dernier dans le minimum de temps.

La figure 4.8 donne une nouvelle fois la décomposition du noyau en unités fonctionnelles.

Dans cette décomposition nous remarquons l'existence de 2 types d'unités:

1- Les unités maîtres : Ce sont les unités qui peuvent émettre des demandes vers d'autres unités (les unités esclaves). Dans la figure 4.8 ces unités sont représentées par des cercles.

2- Les unités esclaves : Ce sont des unités qui exécutent les tâches demandées par les unités maîtres. Dans la figure 4.8 elles sont représentées par des rectangles.

Il existe dans le noyau 4 unités maîtres (les processus d'entrée et de sortie, le processus de contrôle et le processus de calcul) et 4 unités esclaves (les buffers d'entrée et de sortie, et les mémoires associatives des programmes et des paquets).

Par conséquent chaque unité ne peut avoir qu'un seul rôle au niveau du noeud : soit émettre des demandes de services, et dans ce cas elle est maître, soit satisfaire des demandes de service, et dans ce cas elle est esclave. Cette règle de fonctionnement des unités permet d'obtenir un maximum de parallélisme.

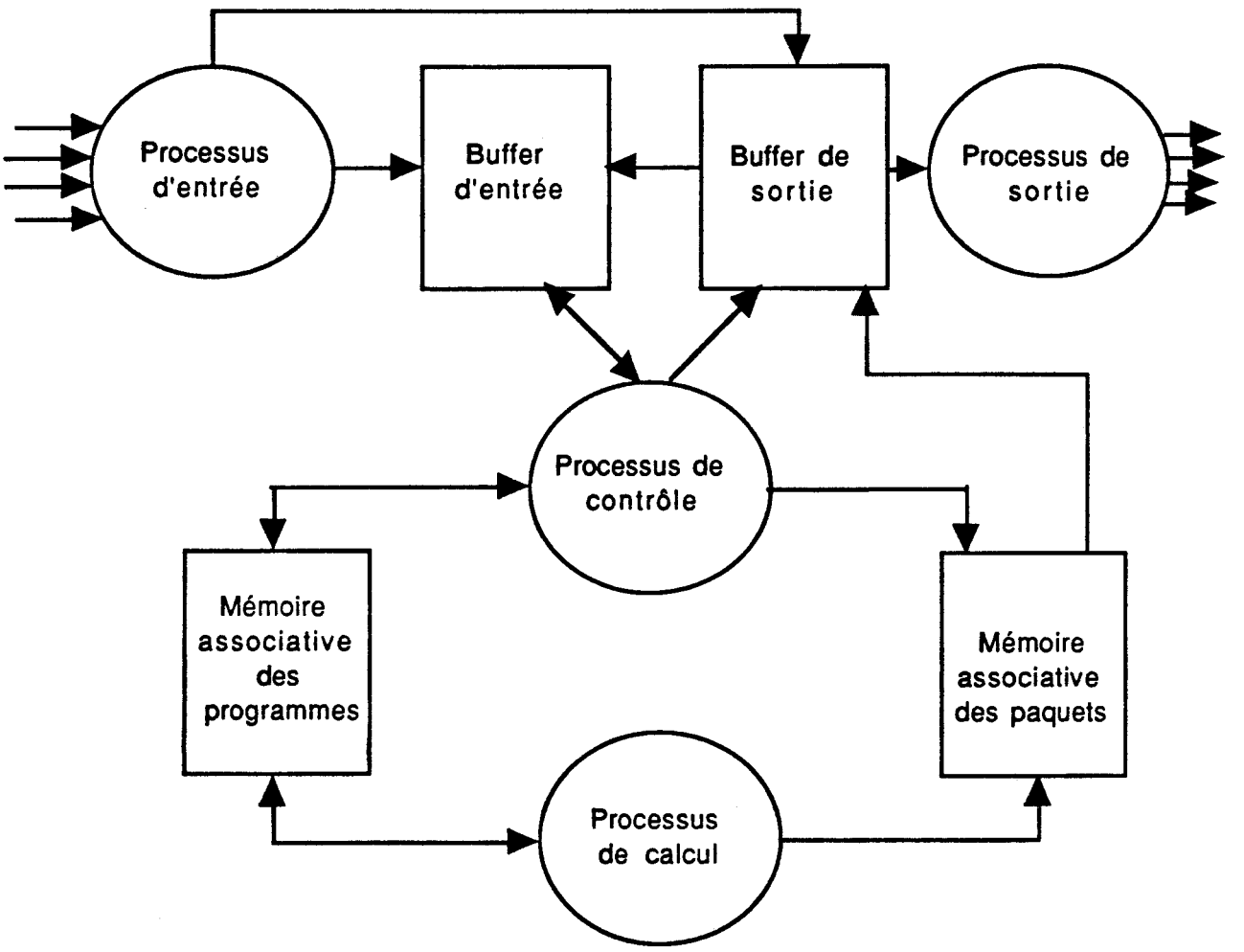


Fig 4.8: Les unités maîtres et les unités escales du noyau

En effet lorsqu'un type de demande ne peut pas être satisfait par un esclave, ceci ne bloque pas l'esclave qui peut prendre en compte d'autres demandes.

Remarque:

Lorsque 2 unités esclaves doivent communiquer, il est nécessaire de prévoir un interface entre les 2 unités. Ceci est le cas pour le couple

mémoire associative des paquets - buffer de sortie et le couple buffer d'entrée - buffer de sortie.

IV.4.1: Le processus d'entrée

La fonction de ce processus est la gestion des 4 canaux d'entrée. Les paquets reçus par ce processus sont transmis au buffer d'entrée si le champ noeud destination est égal au numéro du noeud ou bien si le paquet est en collision (pour les paquets initialisation et requête uniquement). Autrement ils sont transmis vers le buffer de sortie. Afin d'exploiter le parallélisme entre les 4 contrôleurs d'interfaces, nous avons divisé ce processus en 4 processus parallèles (un processus par canal).
En OCCAM nous pouvons écrire:

```
PAR i=0 FOR 4
    WHILE TRUE
        SEQ
            réception.paquet(canal.entrée[i])
            émission.vers.buffer(vers.buffer.entrée[i],
                                 vers.buffer.sortie[i])
```

La procédure réception.paquet mémorise le paquet reçu sur chacun des 4 canaux d'entrée dans un vecteur. Ce vecteur est ensuite lu par la procédure émission.vers.buffer pour qu'il soit expédié, comme expliqué précédemment, soit vers le buffer d'entrée soit vers le buffer de sortie.

IV.4.2: Le processus buffer d'entrée

Ce processus gère les accès à un buffer (mémoire tampon) qui permet aux processus de contrôle et d'entrée de fonctionner à des vitesses différentes. Les paquets à mettre dans ce buffer peuvent provenir soit du processus d'entrée soit du processus de sortie. Dans le premier cas, ce sont les paquets qui ont été reçus à partir des canaux d'entrée, alors que dans le deuxième ce sont des paquet du buffer de sortie dont le champ noeud destination est égal au numéro du noeud. Dans la première version du noyau ce buffer est géré en FIFO -first in first out- c'est à dire que le premier paquet reçu est le premier paquet émis.

Le fragment de programme suivant donne la structure générale de ce processus:

```
WHILE TRUE
  ALT
    ALT i = 0 FOR 4
      (longueur.paquet < espace.libre) & (du.processus.entrée [i]?
                          longueur.paquet)
        SEQ
          mémorise.paquet  (du.processus.entree[i])
          espace.libre := espace.libre - longueur.paquet
          nombre.paquets := nombre.paquets + 1

    (longueur.paquet < espace.libre) & (du.processus.sortie?
                          longueur.paquet
      SEQ
        mémorise.paquet  (du.processus.sortie)
        espace.libre := espace.libre - longueur.paquet
        nombre.paquets := nombre.paquets + 1

    (longueur.paquet < espace.libre) & (du.processus.contrôle.écriture?
                                        longueur.paquet)
      SEQ
        mémorise.paquet  (du.processus.contrôle.ècriture)
        espace.libre := espace.libre - longueur.paquet
        nombre.paquets := nombre.paquets + 1

    (nombre.paquets > 0) & (du.processus.contrôle.lecture ? any)
      SEQ
        émission.paquet  (vers.processus.contrôle,  long)
        espace.libre := espace.libre - long
        nombre.paquets := nombre.paquets - 1
```

Avec la technique de gestion du buffer d'entrée en FIFO, l'évaluation de l'arbre de dépendance, représentant le programme à exécuter, se fait en largeur d'abord. Sur la figure 4.9, nous avons donné un exemple simple de programme à exécuter avec l'arbre de dépendance correspondant. Nous

avons supposé que toutes les expressions sont mémorisées sur le même noeud.

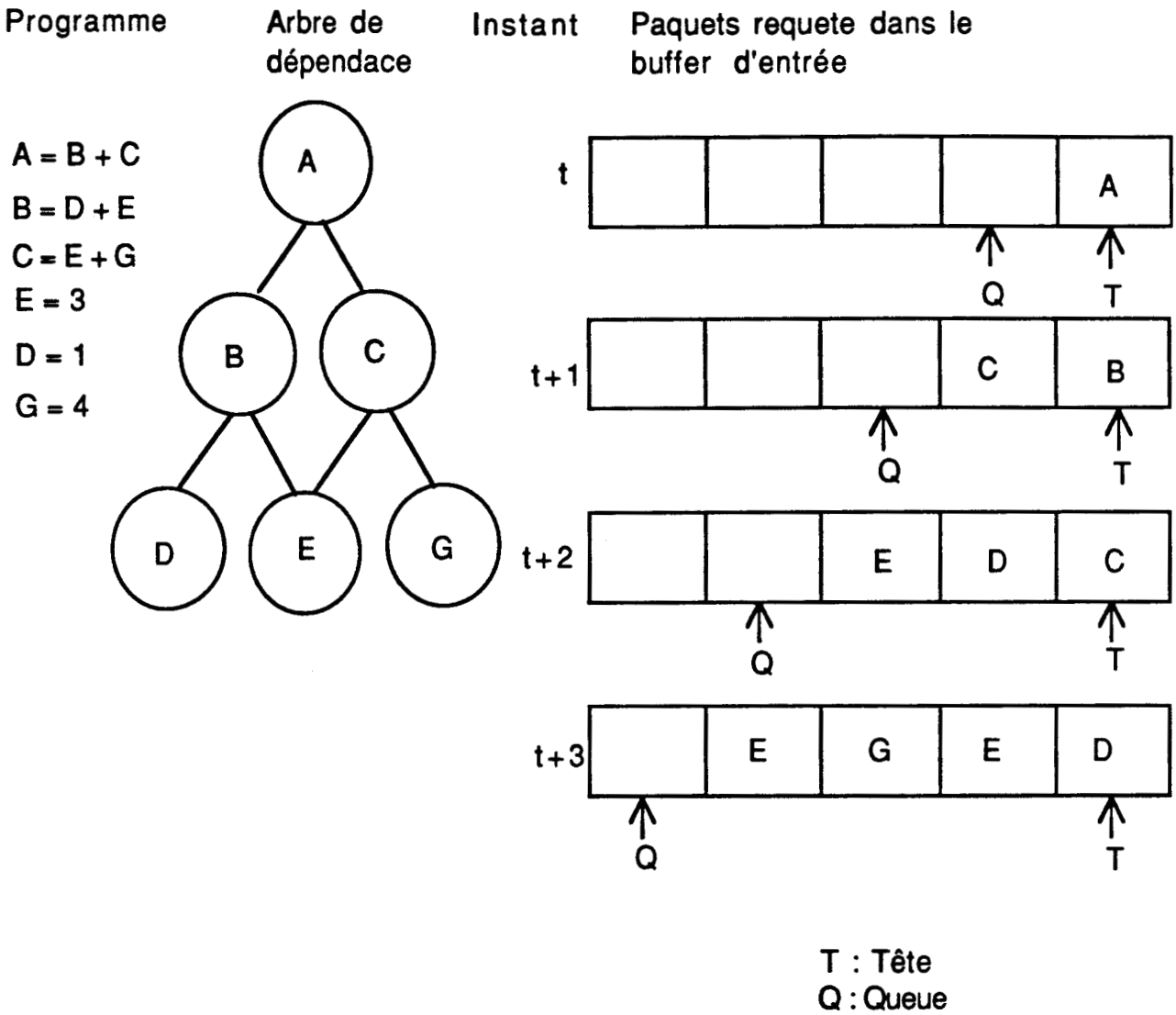


Fig 4.9 : Les paquets requêtes dans le buffer d'entrée

Remarque : Lorsque le processus de contrôle reçoit une requête pour une expression à l'état repos, il doit générer pour chaque argument un paquet requête comme expliqué précédemment. Lorsque le champ destination est égal au numéro du même noeud, ces paquets requêtes sont envoyés directement du processus de contrôle vers le buffer d'entrée au lieu de transiter d'abord par le buffer de sortie.

IV.4.3: Le processus de contrôle

Le processus de contrôle implémente le mode de commande choisi pour la machine. Il est de ce fait le processus le plus important au niveau du noeud. La fonction de ce processus est de lire un paquet et de réaliser le traitement correspondant. 3 cas peuvent se présenter, suivant le type du paquet lu:

1: Le paquet reçu est un paquet initialisation.

La tâche à effectuer consiste à émettre une demande d'écriture d'un paquet initialisation vers la mémoire associative des paquets. Si cette demande est accordée, le processus de contrôle envoie alors le paquet initialisation afin qu'il soit mémorisé. Si la mémoire associative refuse la demande (collision), le paquet est aiguillé vers le buffer de sortie afin que la mémorisation soit tentée de nouveau sur un noeud voisin (hachage local). Ce procédé est répété jusqu'à ce qu'un noeud accepte de mémoriser le paquet. Nous supposons, bien sûr, que la somme totale des capacités mémoires de tous les noeuds est supérieure ou égale à la taille du programme.

```
WHILE TRUE
    SEQ
      vers buffer.entrée ! any
      réception.paquet ( du.buffer.entrée, vecteur)
      IF
         type = initialisation
           réaliser.initialisation (vecteur)
         type = requête
           réaliser.requête (vecteur)
         type = réponse
           réaliser.réponse (vecteur)
```

Comme le programme ne contient que des expressions, les paquets initialisation sont traités par les différents noeuds avant que l'hôte ne génère la première requête. Une fois le programme chargé dans les différents noeuds, seuls les paquets requête ou réponse peuvent transiter d'un noeud à un autre.

2- Le paquet reçu est un paquet requête

Lorsque tout le programme a été chargé dans les différents noeuds, la machine hôte envoie alors un paquet requête vers un noeud du réseau pour déclencher l'exécution du programme.

De façon générale, le traitement de ce paquet va engendrer la génération d'autres paquets requêtes dans le réseau pour évaluer les arguments de l'expression demandée. Lorsqu'un paquet requête atteint son noeud destination il est mémorisé, comme les autres types de paquets, dans le buffer d'entrée, puis lu par le processus de contrôle. Comme indiqué dans le paragraphe IV.2.2, un paquet requête contient toujours un champ "nom de l'expression demandée". Ce mot est alors envoyé par le processus de contrôle au processus mémoire associative des programmes. Deux cas peuvent se présenter:

A- L'expression demandée est stockée dans le noeud.

Dans ce cas la mémoire associative envoie une réponse positive, suivie de l'état dans lequel se trouve l'expression. Si cet état est repos, ceci voudrait dire que l'expression n'a jamais fait l'objet d'une demande auparavant, par conséquent, le processus de contrôle reçoit aussi une copie de l'expression demandée, afin qu'il puisse générer pour chacun des arguments de l'expression un paquet requête. Ceci nécessite l'application de la fonction de hachage globale sur les arguments à évaluer. Si l'expression fait référence plusieurs fois, à un même argument il y a génération d'une requête seulement pour cet argument. Le traitement est terminé par la formation d'un paquet réponse en attente de l'expression demandée qui sera émis vers la mémoire associative des paquets. Ce paquet ne contient pas de champ valeur mais uniquement les champs nom de l'expression demandée et le champ nom de l'expression réceptrice.

Si l'état de l'expression demandée est "calculée", le processus de contrôle reçoit alors la valeur de l'expression. Un paquet réponse est construit puis envoyé vers le buffer de sortie. Ce paquet réponse contient : le nom de l'expression qui a généré la requête (l'expression réceptrice), le nom de l'expression demandée et, enfin, la valeur fournie précédemment par le processus mémoire associative des programmes. Si l'état est en phase d'évaluation, calculable ou en phase de calcul, il y a uniquement envoi d'un paquet réponse en attente vers la mémoire associative des paquets.

B- l'expression demandée n'est pas stockée dans le noeud (collision).

Ceci est indiqué par une réponse négative de la mémoire associative des programmes. Avant de transmettre le paquet requête reçu vers le buffer de sortie, le champ type est mis à "requête en collision". La fonction de hachage locale est alors appliquée au niveau du processus buffer de sortie pour déterminer vers quel voisin il faut transmettre le paquet en collision.

3- Le paquet reçu est un paquet réponse

Dans ce dernier cas, le processus de contrôle envoie directement vers la mémoire associative des programmes: le nom de l'expression destination, le nom de l'expression à mettre à jour, et enfin la valeur calculée pour cette dernière expression.

IV.4.4: La mémoire associative des programmes

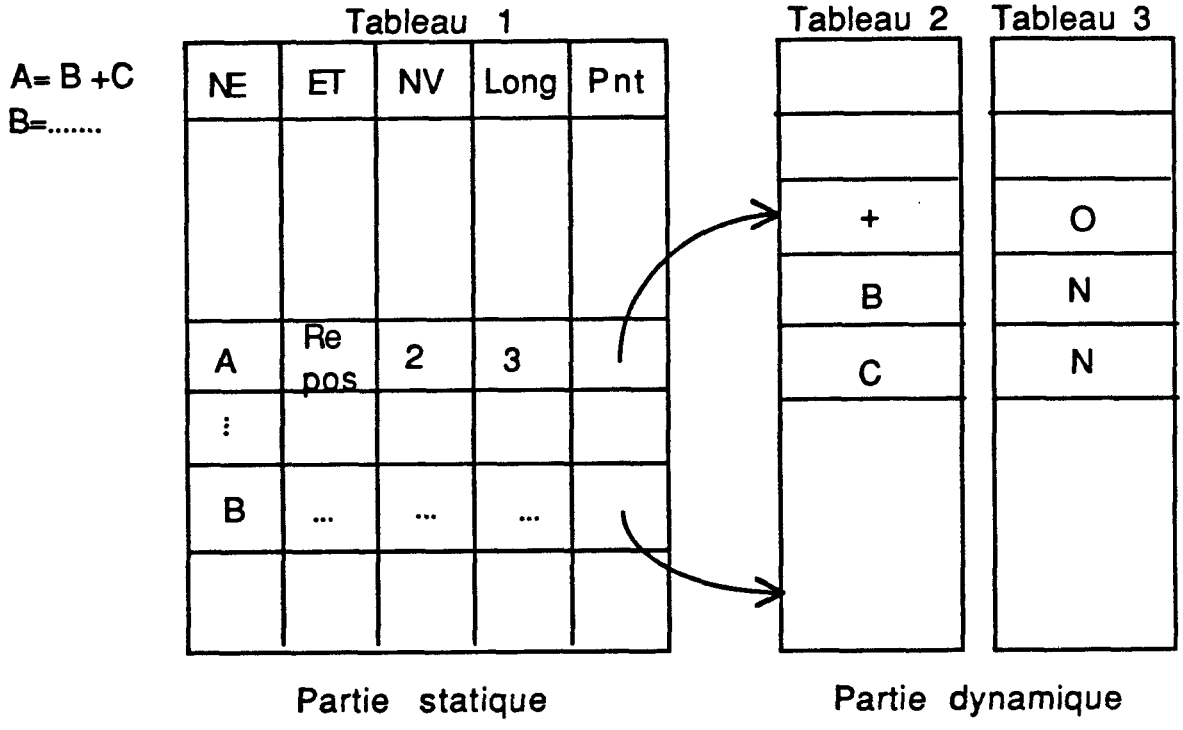
L'un des points originaux du projet N-ARCH, comme il a été indiqué dans le paragraphe III.2.4, est l'utilisation intensive des mémoires associatives [Tour86]. La mémoire associative des programmes a pour fonction de contenir la partie du programme (ensemble d'expressions) qui a été attribuée par le hôte au noeud correspondant. L'utilisation du mode de contrôle "par réduction de graphes", implique que l'image du programme dans cette mémoire est dynamique. Une expression du programme est transformée, en utilisant d'autres expressions comme règles de réécriture, jusqu'à obtention d'une expression non réductible (la forme normale).

L'absence de mémoires associatives dans le Transputer nous a poussé à simuler cette mémoire par une mémoire à accès aléatoire gérée par un processus qui réalise les fonctions d'associativité. Ce processus accède à 3 tableaux :

- Le premier tableau contient la partie statique de l'expression (ou descripteur de l'expression), ce sont les champs sur lesquels un accès associatif peut avoir lieu. Ce tableau contient aussi un pointeur vers le deuxième partie de la mémoire associative des programmes. On peut trouver dans ce tableau le nom de l'expression, son état, le nombre d'arguments non encore évalués, la longueur de la partie dynamique mémorisée dans le deuxième tableau, et un pointeur vers celle-ci.

- Le deuxième tableau contient une chaine de mots qui représentent la partie droite de l'expression. A chaque mot de ce tableau est associé un octet du troisième tableau (la marque).

- le troisième tableau contient les marques de l'expression mémorisée. Les valeurs de ces marques sont les mêmes que celles présentes pour les paquets initialisation (expression, valeur, opérateur). Dans la suite du chapitre nous appelerons ce dernier tableau, le tableau des marques.



NE : Nom de l'expression
ET : Etat
NV : Nombre de variables
Long : longueur
Pnt : pointeur
O : Marque Opérateur
N : Marque Nom d'expression

Fig 4.10 : Constitution du processus mémoire associative des programmes

Cette structure en 3 tableaux offre les avantages suivants:

* Facilite l'écriture du processus mémoire associative, où uniquement les accès sur le premier tableau sont simulés par des accès associatifs.

* Réduit le temps perdu (overhead) par les opérations de récupération d'espace libre. Car ce sont uniquement les zones du deuxième et troisième tableau qui doivent être déplacées une fois qu'une expression a été évaluée (la valeur de l'expression remplace sa définition)

* Peut donner lieu à une réalisation physique directe. Ainsi, en utilisant les circuits mémoire associative disponibles actuellement (de capacité relativement faible), nous pouvons réaliser le premier tableau de la mémoire. Les accès sur le deuxième et troisième tableau se font par adresse -sauf le cas où on réalise une mise à jour d'un argument - et donc elles peuvent être réalisées par des RAM classiques.

```
:
   ALT
     du.processus.contrôle ? opération
       IF
         opération = écriture
           réaliser.écriture(du.processus.contrôle,
                                    vers.processus.contrôle)
         opération = recherche
           réaliser.recherche(du.processus.contrôle,
                                    vers.processus.contrôle)
         opération = mise.à.jour.variable
           réaliser.mise.à.jour(du.processus.contrôle,
                    vers.processus.contrôle, nombre.exp.calculable)
     (nombre.exp.calculable >0) & (du.processus.calcul ? demande)
       IF
         demande = recherche
           réaliser.recherche.exp.calculable (vers.processus.calcul,
                                                index)
         demande = mise.à.jour.exp
           SEQ
             réaliser.mise.à.jour.exp(index, du.processus.calcul)
             nombre.exp.calculable := nombre.exp.calculable - 1
```

Remarque: Le premier tableau de la mémoire associative est pratiquement représenté par un tableau à une dimension (au lieu de deux

comme montré dans la figure 4.10). Ceci a été réalisé afin de réduire le temps d'accès à un élément du tableau en simplifiant le calcul d'index.

L'ensemble des services réalisés par la mémoire associative des programmes peut être divisé en 2 parties suivant l'origine de la demande.

1 Les demandes envoyées par le processus de contrôle

Le processus de contrôle peut envoyer à la mémoire associative des programmes 3 types de demandes:

A) Une demande d'écriture d'un paquet (initialisation):

Une opération de recherche est alors déclenchée pour trouver un emplacement suffisant pour toute l'expression. Deux conditions doivent être vérifiées:

1- Le premier tableau doit contenir assez d'espace pour intégrer une nouvelle entrée.

2- Le deuxième et le troisième tableaux doivent avoir un nombre d'emplacements libres suffisant pour mémoriser l'ensemble des arguments.

Si ces deux conditions sont réalisées, une réponse positive est alors envoyée vers le processus de commande.

B) Une demande de lecture d'une expression

La demande de lecture est obligatoirement accompagnée du nom de l'expression demandée. Une recherche associative est alors déclenchée sur les champs "nom d'expression" du premier tableau. Si le nom de l'expression est trouvé, une réponse positive et l'état sont envoyés vers le processus de commande, comme expliqué dans le paragraphe précédent. Si l'expression contient des arguments non évalués le champ état est mis "en phase d'évaluation". Si non, il est mis à calculable ou calculé.

Si l'expression recherchée n'est pas trouvée suite à une collision, une réponse négative est envoyée vers le processus de commande.

C) Une demande de mise à jour d'une variable

Cette demande correspond au traitement d'un paquet réponse. Elle est accompagnée du nom de l'argument, de la valeur pour cet argument et, du nom de l'expression qui contient cet argument (l'expression réceptrice). Un accès associatif est alors réalisé sur le champ "nom de l'expression" dans le premier tableau par la clé "nom de l'expression réceptrice". Une fois ce nom retrouvé , il y a recherche dans le deuxième tableau de

l'argument à mettre à jour. Ce dernier est remplacé par la valeur reçue dans toute la liste des arguments. Ceci veut dire qu'une mise à jour d'un argument dans une expression agit en fait, sur toutes les occurrences de cet argument dans l'expression. Les marques des mots mis à jour sont alors positionnées à "valeur". Le compteur "nombre d'arguments non calculés" qui se trouve dans le premier tableau est décrémenté du nombre d'occurrences de l'argument dans l'expression. S'il a atteint la valeur nul, le champ "état de l'expression" est mis à calculable.

2 Les demandes provenant du processus d'exécution

L'une des fonctions réalisée par la mémoire associative des programmes est le stockage des expressions calculables. Pour réduire une expression il faut réaliser deux opérations:

A) Envoi de l'expression calculable

Si la mémoire associative des programmes contient des expressions calculables et si le processus de calcul est libre, un accès associatif est réalisé sur le champ "état de l'expression " dans le premier tableau. Cet accès a pour but de déterminer l'adresse (l'indice dans le premier tableau) d'une expression calculable. Cette adresse est par la suite utilisée par le module d'émission de la mémoire associative qui se charge d'envoyer l'expression calculable trouvée vers le processus de calcul.

B) Mise à jour de l'expression calculable.

Lorsque le processus de calcul a reçu une expression calculable et qu'il a terminé son évaluation, une opération de mise à jour d'expression est réalisée. Le processus de calcul envoie un paquet qui contient uniquement la valeur calculée. L'utilisation de l'adresse de l'expression pour réaliser la mise à jour offre les avantages suivants:

1) Gain de temps en réalisant un accès direct pour la mise à jour au lieu d'un accès réalisé après exécution de l'algorithme qui simule l'associativité de la mémoire.

2) Possibilité d'intégrer dans le même noeud plusieurs processus exécution. Afin de réaliser cela la mémoire associative doit traiter les demandes d'envoi d'expressions calculables et les demandes de mise à jour d'expression de manière parallèle.

Dans la version actuelle du noyau il n'y a qu'un seul processus de calcul. Par conséquent, les opérations de recherche et de mise à jour d'expression

calculable sont réalisées de manière séquentielle. L'utilisation de plusieurs unités de calcul n'est intéressante que si le nombre d'expressions calculables est très important durant l'évaluation du programme. Ceci dépend bien sûr du type du programme à exécuter et du niveau de grain des opérations. Lorsque le taux du parallélisme n'est pas très important l'utilisation de plusieurs processus de calcul par noeud n'offre pas beaucoup d'avantages.

Jusqu'à présent, nous avons parlé des accès associatifs sans préciser de quelle manière le gestionnaire de la mémoire réalise ces accès. Dans le noyau N-ARCH 3 solutions ont été réalisées:

A- Par des accès séquentiels.

B- Par méthode de tri des expressions (en utilisant un algorithme dichotomique).

C- Par utilisation d'une fonction de hachage.

Dans la première méthode, les expressions du programme sont stockées dans les trois tableaux les unes à la suite des autres. La zone où est insérée l'expression est la première zone libre dans chacun des 3 tableaux. La recherche d'une expression se fait en accédant séquentiellement au premier tableau, et en comparant chaque fois la clé (qui est le nom de l'expression recherchée) à chacun des champs "nom d'expression" des positions accédées. Si la mémoire associative des programmes contient P expressions, le nombre moyen d'accès et de comparaisons est P/2 pour chaque recherche d'expression.

En utilisant un algorithme de tri, les expressions dans le premier tableau peuvent être accédées plus rapidement. Ces expressions sont alors classées par ordre croissant du champ "nom d'expression". Pour rechercher une expression on compare son champ "nom d'expression" par rapport au nom de l'expression se trouvant au milieu du premier tableau. Un algorithme de recherche dichotomique est ensuite utilisé pour trouver la position où est rangée l'expression demandée si elle existe. Dans cette méthode, si la mémoire contient P expressions, le nombre moyen d'accès pour réaliser une opération de lecture ou d'écriture est au maximum égale à LOG_2P.

Dans ces deux premières méthodes les trois tableaux contiennent 2 zones contiguës : Une zone occupée, et une zone libre. Car ces trois tableaux sont remplis entrée par entrée.

Dans la troisième méthode, la mémoire associative est simulée par une mémoire aléatoire gérée par un processus OCCAM qui utilise une fonction de hachage. Dans le chapitre II nous avons présenté la machine N-ARCH comme une architecture qui contient deux niveaux de mémoires associatives. Dans ce chapitre nous avons dit que le premier niveau est réalisé par l'utilisation des fonctions de hachage globale et locale. L'utilisation d'une fonction de hachage pour réaliser aussi le deuxième niveau (le niveau mémoire) revient a considèrer la machine N-ARCH comme une machine parallèle à deux niveaux de hachage: Un niveau inter-noeuds, par les fonctions de hachage globale et locale, et un niveau intra-noeud, par la fonction de hachage qui simule la mémoire associative.

Pour mémoriser l'expression de nom E dans la mémoire associative, la fonction de hachage est appliquée sur E. La chaîne de caractères représentant E est donc considérée comme un entier (fig 4.11). Cet entier (la clé) est traité par la fonction de hachage afin de déterminer un indice dans le premier tableau de la mémoire associative. Si l'entrée correspondante est libre, il y a écriture de l'expression. Si non une méthode de gestion des collisions est alors utilisée pour déterminer une nouvelle zone où sera tentée de nouveau l'opération d'écriture.

Il existe dans la littérature plusieurs manières de construire la fonction de hachage. Il existe aussi plusieurs manières de résoudre le problème des collisions. On trouvera dans [Lewi88] une étude détaillée de l'utilisation des fonctions de hachage. Pour simuler la mémoire associative des programmes du noeud N-ARCH dans le noyau OCCAM, nous avons utilisé une fonction de hachage qui considère le nom de l'expression (la clé) comme un nombre exprimé en base K, où K est le nombre de caractères pouvant exister dans la chaîne représentant le nom de l'expression. La fonction de hachage convertira tout simplement ce nombre en base K dans un nombre en base 10. Afin d'assurer que le résultat de la fonction de hachage (qui est utilisé comme un indice dans le premier tableau) soit compris entre 0 et (T - 1), où T est la taille du premier tableau, les opérations de conversion de chacun des caractères sont réalisées modulo T.

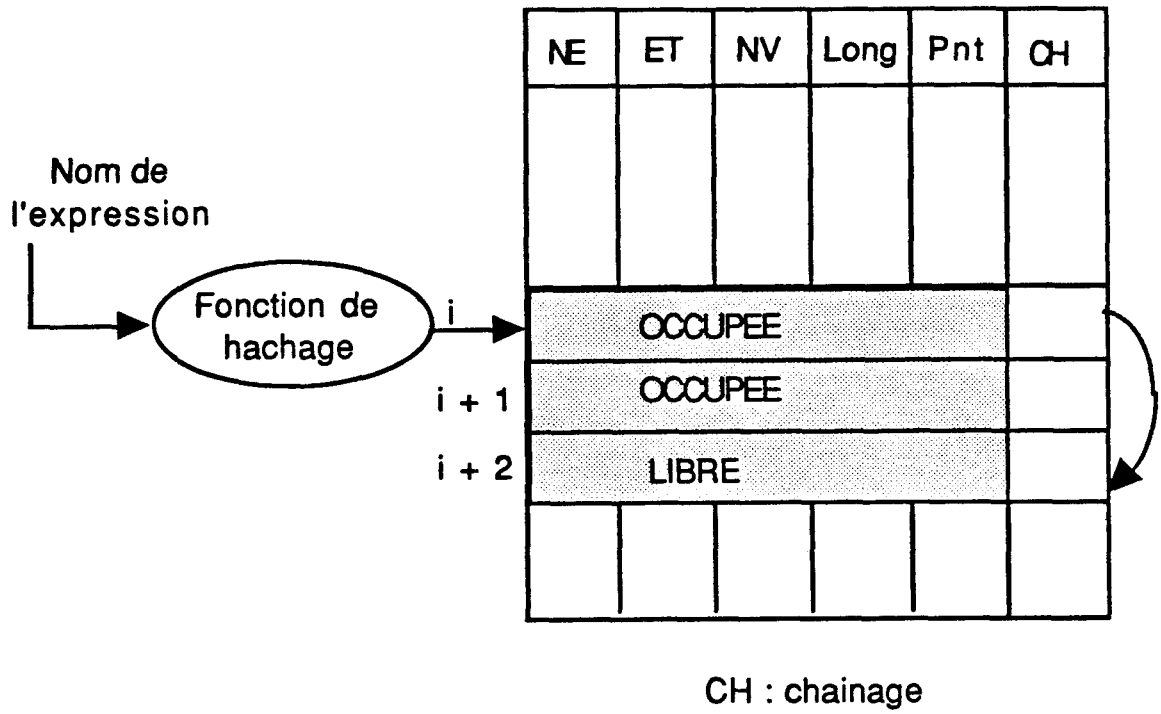


Fig 4.11 : Utilisation d'une fonction de hachage pour simuler une mémoire associative

D'après [Lewi88] cette technique produit très peu de collisions lorsque le nombre T est premier. La fonction de hachage peut être écrite en OCCAM:

```
[long] BYTE  chaîne RETYPES nom.expression:
SEQ
      K := 26                        --les 26 lettres de l'alphabet
      blanc := INT (' ')             --le code ASCII de l'espace
      Index:= 0
      SEQ i = 0 FOR long
            index := ( (K * index) + (chaîne[i] - blanc) ) REM T
            --REM est l'opérateur qui donne le reste de la division
```

Comme les mots ont une longueur de 32 bits, la variable long vaut 4.

Les résultats d'exécution de programmes ont montré que cette fonction produit de très bons résultats (voir chapitre V).

Quelle que soit la fonction de hachage utilisée, il est toujours nécessaire de prévoir une méthode pour résoudre les collisions. Une bonne fonction de hachage minimise le nombre de collisions mais ne l'élimine jamais.

Dans le noyau N-ARCH nous avons utilisé une méthode de résolution par chaînage.

Dans cette méthode les expressions pour lesquelles la fonction de hachage donne le même indice (les synonymes) sont mémorisées dans une liste chaînée. Un champ supplémentaire est ajouté dans le premier tableau. Ce champ est à -1 s'il n'y pas de chaînage. Si non il contient l'indice dans le premier tableau du prochain synonyme. En cas de collisions sur la position i, on accède à la position (i+1) pour tester si elle est libre. Si c'est le cas il y a écriture de l'expression dans cette position et mise à jour du champ chaînage de la position i. L'avantage de la méthode de résolution des collisions par chaînage est l'exploitation de toute la mémoire avant de produire un dépassement de capacité ou overflow. En effet, en utilisant le chaînage, l'overflow se produit uniquement lorsqu'il n'y a plus d'espace dans les tableaux.

Lorsqu'un paquet a tous ses arguments évalués, son champ "état" est égal à calculable. Si plusieurs paquets calculables existent au niveau de la mémoire associative, ils sont alors chaînés. Cette méthode permet d'avoir un accès direct à l'ensemble des paquets calculables dans la mémoire des programmes et minimise ainsi le temps perdu par la simulation de la mémoire associative (ce temps n'est pas nul, car il y a gestion des pointeurs). Cette méthode de chaînage simule alors l'accès associatif par le champ état calculable.

Les résultats d'exécution de quelques programmes ont montré que cette approche est celle qui offre un temps d'overhead minimum. Par conséquent, dans la suite du projet, nous supposerons que c'est cette méthode qui est utilisée.

IV.4.5: Le processus de calcul

Ce processus est responsable de l'exécution des opérations arithmétiques et logiques. Lorsqu'une expression de la mémoire associative des programmes devient calculable, elle est recopiée puis envoyée au processus de calcul, et cela dès que ce dernier peut recevoir une nouvelle expression à évaluer. L'expression sélectionnée est mise à l'état "en cours d'évaluation" et son indice dans le premier tableau de la mémoire associative des programmes est mémorisé pour être utilisé dans

l'opération de mise à jour. Une fois l'évaluation de l'expression terminée, le résultat est envoyé, aussi bien vers la mémoire associative des expressions, pour mettre à jour l'expression dont l'indice a été mémorisé, que vers la mémoire associative des paquets, afin de compléter le ou les paquets réponse en attente de cette valeur.

La structure générale de ce processus:

```
WHILE TRUE
    SEQ
        vers.mémoire.programmes ! requête.lecture
        réception.expression.exécutable (de.mémoire.programme,
                                         expression)
        évaluation (expression, valeur)
        vers.mémoire.programme ! mise.à.jour
        envoie.valeur (vers.mémoire.programme, valeur)
        envoie.paquet (vers.mémoire.paquet, nom.expression,
                                            valeur)
```

IV 4 6: Le processus mémoire associative des paquets

Lorsqu'une requête arrive au processus de contrôle, il y a génération et envoi d'un paquet réponse vers la mémoire associative des paquets si l'expression demandée n'est pas à l'état calculée.

Ce paquet est appelé *paquet réponse incomplet* (ou en *attente*) car il contient uniquement les champs suivants:

* Le type
* Le nom de l'expression réceptrice
* Le numéro du noeud destination
* Le nom de l'expression évaluée.
* Un champ valeur vide, qui sera mis à jour ultérieurement.

Au niveau de la mémoire associative des paquets, les paquets réponse en attente sont organisés en groupe. Un groupe contient tous les paquets qui ont le même champ "nom de l'expression évaluée" et il est représenté par une entrée dans la mémoire associative des paquets. Cette notion de groupe de paquets réponse est nécessaire car plusieurs requêtes peuvent être reçu par le noeud pour la même expression.

Lorsque le processus de calcul a réalisé l'évaluation d'une expression, un paquet contenant le nom de l'expression et la valeur calculée est envoyé vers la mémoire des paquets (comme on vient de le voir dans le paragraphe précédent). Le nom de l'expression est utilisé par ce processus comme clé, pour réaliser un accès associatif sur l'ensemble des paquets réponse en attente.

Dans le mode de réduction de graphes si une même expression est demandée par plusieurs autres expressions elle n'est évaluée qu'une seule fois. Tous les paquets réponse en attente d'une même expression sont mémorisés dans la même entrée au niveau de la mémoire des paquets (groupe).

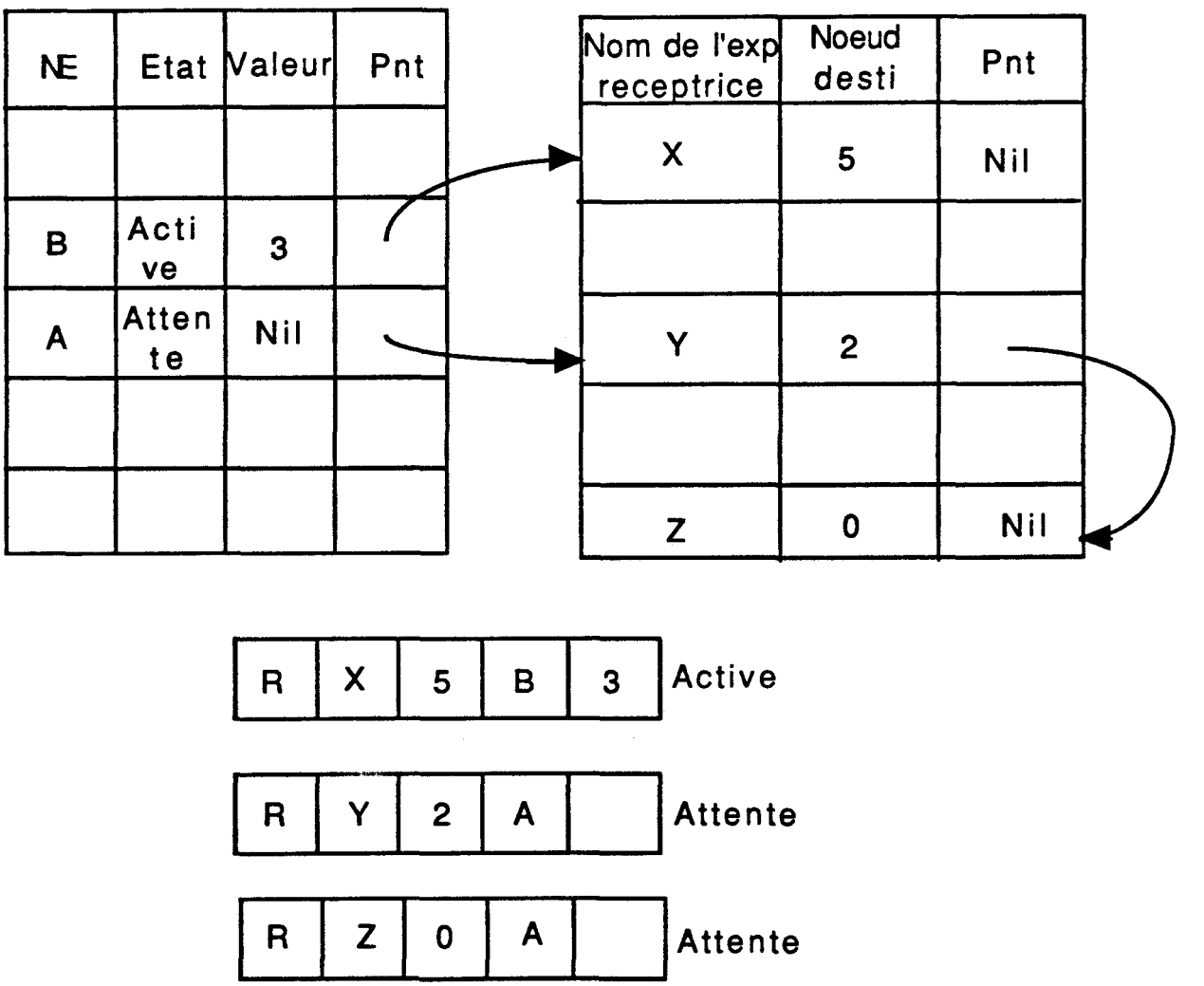


Fig 4.12 : Les paquets réponse dans la mémoire associative des paquets

Lorsque l'expression demandée a été calculée et que sa valeur a été envoyée par le processus de calcul, l'ensemble des paquets en attente pour cette expression devient actif et ils peuvent être expédiés. La liste chaînée des paquets en attente pour cette expression est alors accédée paquet par paquet. Pour chaque paquet réponse en attente lu, le processus mémoire associative des paquets met à jour le champ "valeur" par la valeur de l'expression reçue du processus de calcul, puis expédie le paquet réponse complet vers le buffer de sortie (fig 4.12).

Une fois que tous les paquets en attente d'une même expression ont été envoyés, l'entrée de cette expression dans la mémoire des paquets est éliminée. Comme pour la mémoire associative des programmes, cette mémoire est représentée par 2 tables. La première table contient le descripteur du groupe. Ce descripteur est formé des informations communes à tous les paquets du groupe comme: le nom de l'expression évaluée, un champ valeur initialisée à -1 , un champ état du groupe, et pointeur vers le deuxième table. Celle-ci contient les éléments du groupe. Ces éléments sont des paquets en attente organisés en listes chaînées. L'émission de 2 paquets complets consécutifs (appartenant à un même groupe ou non) peut être intercalée par la réception d'un paquet réponse en attente, ou une demande de mise à jour par le processus de calcul.

```
ALT
  du.processus.control ? nom.expression.évaluée
    SEQ
      recherche.expression (nom.expression.évaluée, trouve)
      réception (nom.expression.évaluée, noeud.recepteur)
      IF
        trouve
          insertion.table2 (nom.expression.évaluée,
                    nom.expression.receptrice, noeud.recepteur)
      TRUE
          SEQ
            création.entrée.table1(nom.expression.évaluée)
            insertion.table2(nom.expression.évaluée,
                    nom.expression.receptrice, noeud.recepteur)
```

```
du.processus.calcul ? nom.expression.évaluée
  SEQ
    du.processus.calcul ? valeur
    mise.à.jour.table1 (nom.expression.évaluée, valeur)
    ensemble.prêt := ensemble.prêt + 1
(ensemble.prêt > 0) & module.interface ? any
  SEQ
    emission.paquet.réponse(vers.module.émission,
                nom.expression, dernier)
    IF
      dernier
        SEQ
          ensemble.prêt := ensemble.prêt - 1
          eliminer.entrée.table1(nom.expression)
      TRUE
        SKIP
```

La procédure émission.paquet.réponse forme un paquet réponse, puis l'envoie vers le buffer de sortie. Si le paquet est le dernier qui se trouve dans la chaîne, la variable ensemble.prêt qui compte le nombre d'entrées mises à jour dans le tableau1 est décrémentée et l'entrée en question est récupérée (garbage collector).

Comme expliqué dans l'introduction du paragraphe IV.4, les 2 processus mémoire des paquets et buffer de sortie représentent des unités esclaves. La communication entre ces 2 unités est possible grâce à l'introduction d'un processus maître (l'interface). L'utilisation de cet interface d'émission permet d'éviter, comme on le verra dans le paragraphe suivant, que le processus buffer de sortie reste bloqué sur une demande de lecture dans la mémoire associative des paquets (Fig 4.13)

Remarque: Comme pour la mémoire associative des programmes, les accès associatifs ont été réalisés de 3 manières différentes (accès séquentiel, méthode dichotomique, fonction de hachage). Celle qui offre le plus d'avantages est la méthode par utilisation d'une fonction de hachage.

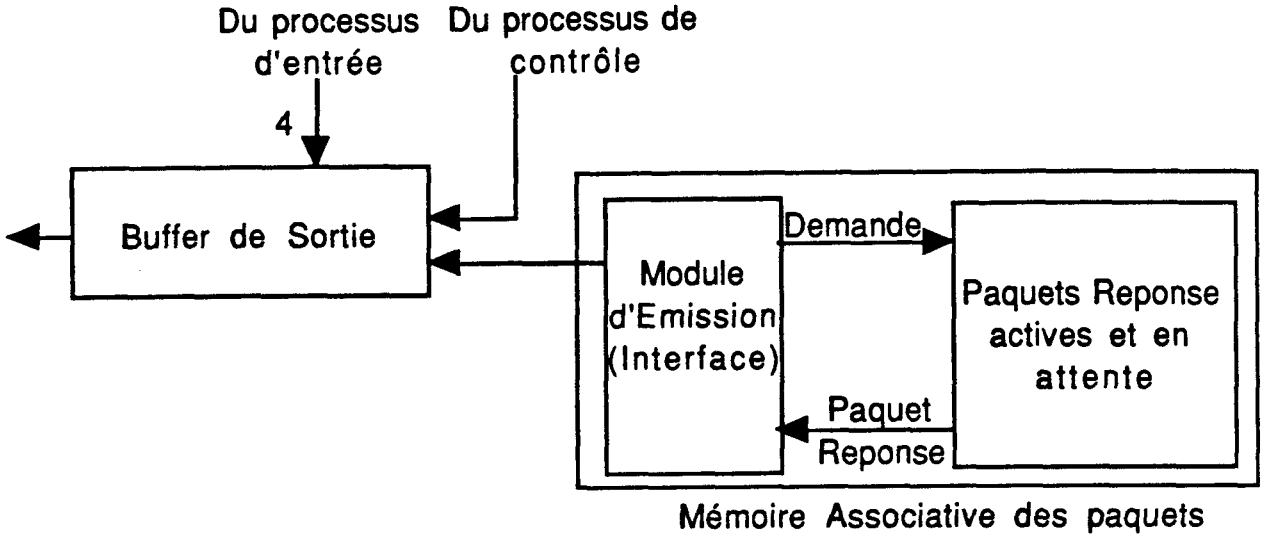


Fig 4.13 Le processus mémoire associative des paquets

IV.4.7: Le processus buffer de sortie

La fonction de ce processus est la mémorisation des paquets à émettre vers les noeuds voisins. Il permet par la même occasion aux différentes unités qui lui sont connectées de fonctionner à des vitesses différentes.

Comme nous l'avons vu, dans le chapitre précédent, la mémoire des paquets peut contenir, aussi bien, des paquets à destination d'autres noeuds que des paquets à destination du noeud lui-même.

Par conséquent ce processus doit aussi gérer les paquets à destination du buffer d'entrée (paquets qui réalisent un traitement local). Le processus buffer de sortie peut être décomposé en 2 sous processus parallèles (figure 4.14).

*L'unité de mémorisation.

*L'unité de gestion des canaux d'entrée.

L'unité de mémorisation est composée de 5 queues. Chaque queue correspond à un canal de sortie, et contient les paquets à émettre soit vers le buffer d'entrée soit vers les 4 sous processus parallèles du processus de sortie (Cf paragraphe suivant). Les 5 queues sont gérées de manière parallèle afin d'éviter les attentes actives et, d'exploiter le parallélisme dans l'émission des paquets sur les liens externes.

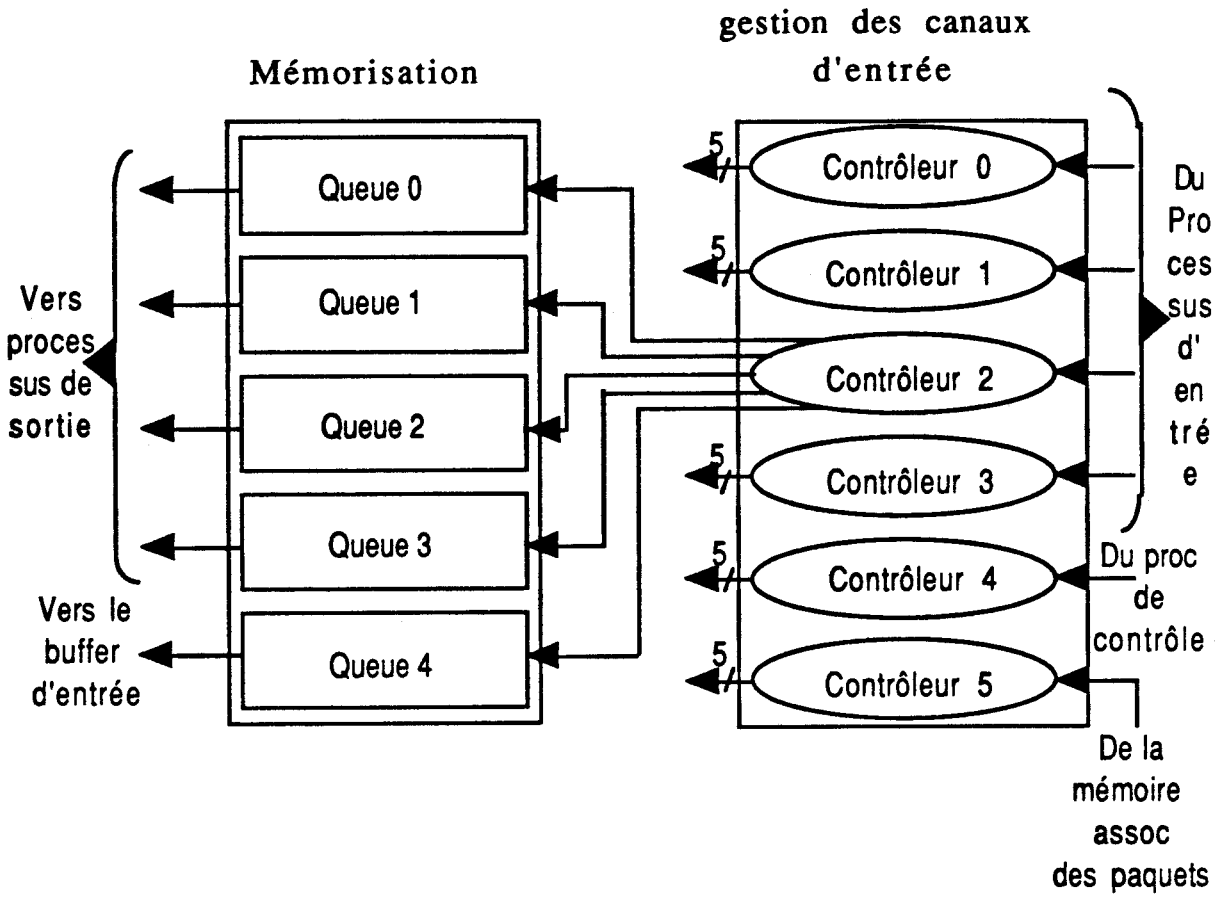


Fig 4.14 Le processus buffer de sortie

L'unité de gestion des canaux d'entrée a pour fonction essentielle d'assurer le routage des paquets qui sont transmis au buffer de sortie sur l'une des 5 voies (les queues). Les paquets reçus par le buffer de sortie peuvent provenir de 6 processus différents: les 4 sous processus d'entrée parallèles, le processus de contrôle, et le processus mémoire des paquets. Par conséquent, l'unité de gestion doit gérer 6 canaux différents (1 canal par processus). Lorsqu'un paquet arrive sur l'un de ces canaux, une fonction de routage est utilisée par cette unité pour déterminer la queue qui devra mémoriser le paquet reçu. La fonction de routage utilise pour cela le champ "numéro du noeud destination" du paquet à émettre. Ce numéro est considéré comme indice de colonne dans une table à 2 dimensions. L'autre indice utilisé (l'indice de ligne) est le numéro du noeud qui exécute le noyau. Si ce numéro est i, et si le numéro du noeud destination est j, le numéro de la queue qui va contenir le paquet - et par conséquent le numéro du canal qui sera utilisé pour envoyer le paquet

vers l'un des voisins - est table.routage[i, j]. Ce numéro est compris entre 0 et 4. Les numéros de 0 à 3 correspondent aux 4 canaux vers le processus de sortie, et le numéro 4 correspond au canal du buffer d'entrée.

La figure 4.15 montre un exemple simple de réseau avec 4 noeuds et la table de routage correspondante. Comme on peut le voir sur l'exemple précédent, la méthode de routage utilisée est une méthode statique (routage non adaptatif).

Ainsi, quelque soit l'état du réseau et quelque soit l'instant où est appliquée la fonction de routage, le résultat de cette fonction pour un noeud destination donné est toujours le même. Durant l'exécution d'un programme, certains noeuds peuvent avoir un plus grand nombre de paquets à transmettre que d'autres. La méthode statique peut donc ralentir l'exécution d'un programme et détériorer les performances de la machine. Néanmoins, elle évite l'introduction de nouveaux types de paquets (paquets de contrôle) utilisés pour informer tous les noeuds du réseau de la charge en nombre de paquets dans les buffers de sortie et d'entrée au niveau de chaque noeud. La manipulation de ces paquets de contrôle produirait aussi un overhead du point de vue temps qui n'existe pas dans la méthode statique.

En réalité le champ "numéro du noeud destination" du paquet à transmettre n'est utilisé directement que si le paquet est à l'état routage. Lorsque ce dernier est en collision le noeud désigné par le champ "numéro du noeud destination" a été atteint, et par conséquent, il n'est plus nécessaire d'y revenir.

Dans ce cas, il y a utilisation de la fonction de hachage locale qui détermine un noeud voisin où sera envoyé le paquet. Le numéro du noeud, ainsi calculé, est utilisé comme indice de colonne dans la table de routage.

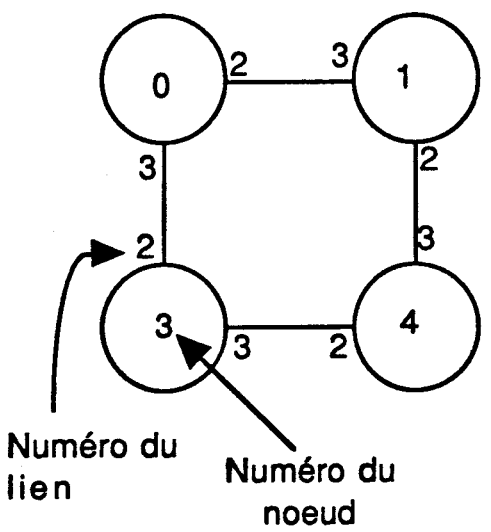
L'unité de gestion des canaux d'entrée peut être représentée par 6 processus parallèles (un processus par canal d'entrée), qui ont chacun la forme suivante:

```
SEQ
  réception.paquet (canal.d'entrée, paquet)
  IF
    type = routage
      SEQ
```

```
        routage (paquet, numéro.destination, numéro.noeud,
                                        numéro.canal )
        émission.paquet (vers.queue[numéro.canal], paquet)
TRUE    -- type = collision
  SEQ
        hachage.locale (paquet, numéro.noeud, numéro.voisin)
        routage (paquet, numéro.voisin, numéro.noeud,
                                              numéro.canal)
        émission.paquet (vers.queue[numéro.canal], paquet)
```

Remarque:

Les paquets à transmettre du buffer de sortie vers le buffer d'entrée transitent d'abord par un processus d'émission du buffer de sortie. Comme expliqué dans le paragraphe précédent les processus buffer d'entrée et buffer de sortie sont des unités esclaves. Pour réaliser une communication entre ces deux processus, il est nécessaire d'intègrer un processus maître (l'interface).



Noeud départ \ Noeud arrivée	0	1	2	3
0	4	2	3	2
1	3	4	2	2
2	2	2	4	3
3	2	3	2	4

Table de routage (numero de liens)

Fig 4.15 : Un hypercube d'ordre 2 avec sa table de routage

IV.4.8: Le processus de sortie

Le dernier processus du noyau que nous allons voir est le processus de sortie. Ce processus a pour fonction d'envoyer les paquets se trouvant dans le buffer de sortie vers les canaux physiques externes. Afin d'exploiter les possibilités matérielles des interfaces de liens, ce processus est divisé en 4 processus parallèles. Chacun de ces processus a comme tâche :

1) La génération d'une requête vers la queue du buffer de sortie qui lui correspond.

2) La réception d'un paquet à émettre.

3) L'émission du paquet reçu sur le canal dont il a le contrôle et revenir à 1.

Cet algorithme peut être traduit en OCCAM par:

```
PAR i = 0 FOR 4
  WHILE TRUE
    SEQ
      vers.queue[i] ! any
      réception.paquet (de.queue[i], paquet)
      émission.paquet (canal.externe[i], paquet)
```

IV.4.9: Conclusion:

Pour terminer ce paragraphe nous dirons, que le noyau de l'émulateur N-ARCH a été conçu de façon à représenter tout le parallélisme qui existe dans le noeud de la machine tel que nous l'avons vu dans le chapitre III.

Dans ce chapitre, nous avons vu que l'exécution de processus parallèles induit un overhead nécessaire aux changements de contexte. Par conséquent lorsque la charge affectée à chaque noeud du réseau (nombre de paquets à traiter) est faible, le rapport entre le temps d'exécution utilisé pour réaliser le traitement sur le temps total consommé par les changements de contexte est faible. Ce qui revient à dire que les performances du noyau ne sont intéressantes que si un grand degré de parallélisme existe dans le programme à exécuter. Cette conclusion est confirmée par les résultats que nous présenterons dans le chapitre V.

Du point de vue implémentation, les changements dans les différentes versions du système de développement du transputer (TDS) nous ont poussé à écrire une partie du noyau en OCCAM1 et une autre partie en OCCAM2 et à adapter à chaque fois la partie déjà écrite pour suivre les différentes versions de TDS.

L'existence dans la dernière version de TDS (reçue par le laboratoire au mois d'octobre 88) d'un debugger symbolique pour un réseau de Transputers, a énormément facilité la conception et la mise au point des dernières parties du noyau. Nous pouvons regretter, bien sûr, que ce debugger n'existait pas au moment où nous avons commencé la conception du noyau.

IV 5 LES FONCTIONS DE HACHAGE DANS L'EMULATEUR

L'un des points originaux du projet N-ARCH est l'utilisation de fonctions de hachage pour répartir ou rechercher les objets dans le réseau. Comme expliqué dans le chapitre III les objets du programme sont répartis dans le réseau en appliquant une fonction de hachage globale sur les noms de ces objets (les noms d'expressions).

Cette fonction de hachage globale est aussi utilisée par le processus de contrôle lorsqu'un objet est recherché. La répartition des expressions sur le réseau est donc faite de manière statique. En d'autres termes, les expressions ne sont pas déplacées pendant l'exécution. Si durant l'opération de chargement du programme, la mémoire associative des programmes d'un noeud est remplie, les prochains paquets initialisation à destination de ce noeud seront aiguillés vers les voisins en utilisant la fonction de hachage locale. Le même procédé est utilisé au niveau du voisin jusqu'à obtention d'un noeud qui accepte le paquet.

Nous allons dans ce paragraphe étudier chacune de ces 2 fonctions, et présenter celles qui ont été retenues pour la réalisation du noyau.

IV.5.1 : La fonction de hachage globale

Cette fonction de hachage joue, comme on le verra dans le chapitre V, un rôle important dans le fonctionnement du système. Un mauvais choix de cette fonction provoquera, sans un aucun doute, de très mauvaises

performances pour le système. Pour obtenir une bonne répartition du programme la fonction de hachage doit vérifier les 2 conditions suivantes:

1) Répartir les expressions du programme sur le maximum de noeuds de manière équitable. Ceci permettra un maximum de parallélisme dans le traitement des expressions.

2) Regrouper sur le même noeud les expressions dépendantes (localité des données). Cette localité permettra de réaliser le minimum de communication diminuant ainsi, l'overhead nécessaire aux opérations d'émission et de réception de paquets.

Afin de réaliser ces 2 conditions (parfois incompatibles), une possibilité à étudier serait de réaliser une analyse du programme avant exécution. Cette phase d'analyse aura comme résultat pour chaque nom d'expression dans le programme initial (fourni par l'utilisateur) un nouveau nom qui le remplacera durant l'exécution du programme

Cette phase d'analyse est réalisé par un module du processeur hôte. Il prendra en compte pour cela:

* Le programme initial.

* Le nombre de processeurs dans le réseau ainsi que la topologie du réseau.

* La fonction de hachage globale utilisée par le processeur hôte et les processeurs du réseau.

La figure 4.17 montre une transformation réalisée sur un programme. La fonction de hachage globale utilisée dans l'exemple consiste à prendre la valeur ASCII du deuxième caractère du nom de l'expression modulo le nombre de processeurs dans le réseau.

Dans la figure 4.18a nous avons représenté dans un cercle les différents noms d'expressions. Ce cercle contient aussi le numéro du noeud où est stockée l'expression correspondante. Lorsque 2 expressions reliées par un arc de l'arbre de dépendance sont mémorisées sur le même noeud, elles sont alors regroupées dans le même cercle. C'est ainsi que A0 et B0 ont été regroupées dans un cercle. Dans la figure 4.18b, les expressions dont les noms sont D2, E3, G0, et F1 peuvent être traitées en parallèle et il existe 4 arcs qui vont d'un cercle à un autre. Ces arcs représentent la quantité de communication entre les noeuds nécessaire à l'évaluation du programme.

La figure 4.18a montre la répartition du programme sans transformation du programme. Comme on peut le voir la fonction de hachage utilisée n'est pas adaptée à ce programme.

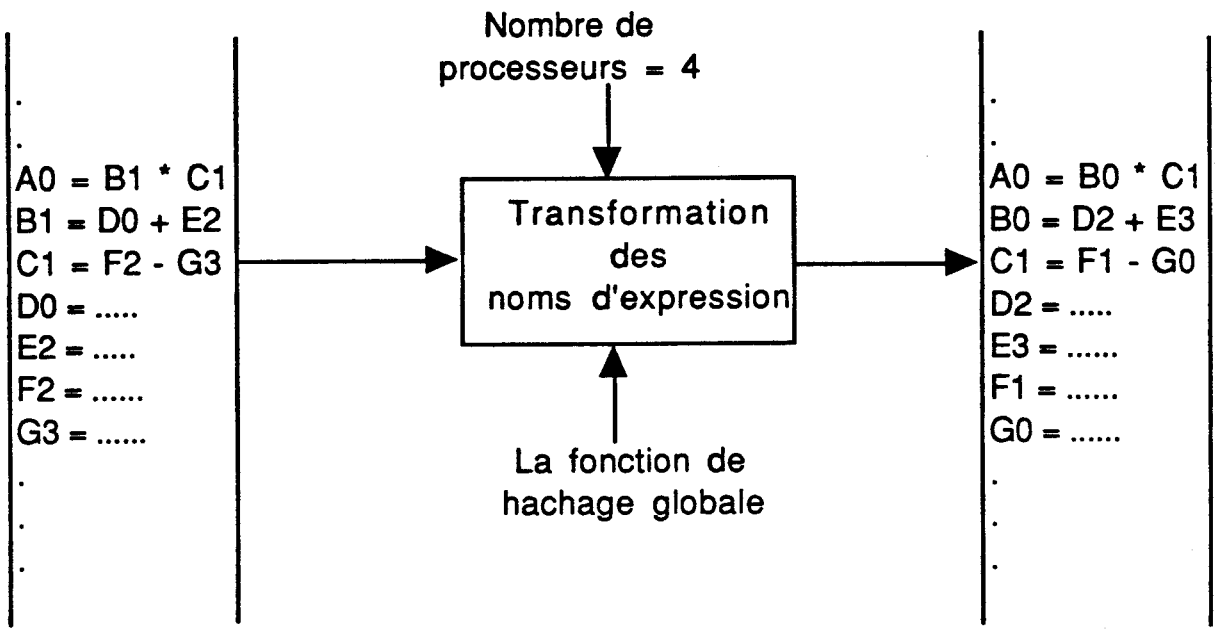


Fig 4.17 : Transformation des noms d'expression d'un programme

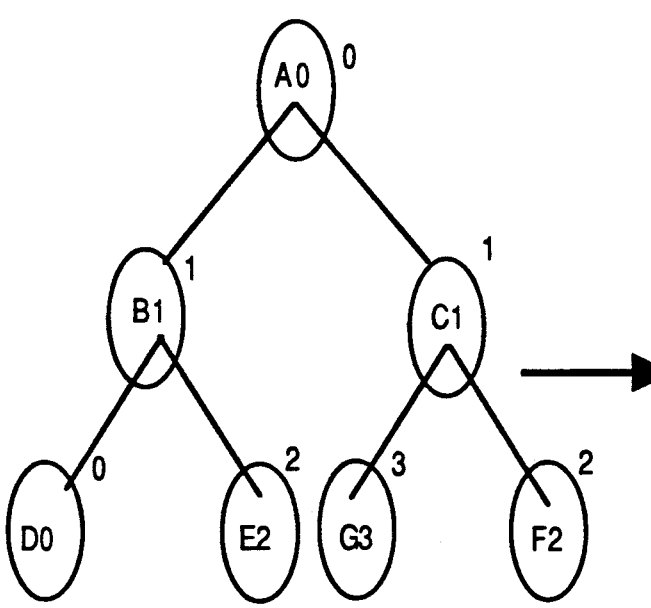


Fig 4.18a : L'arbre de dépendance sans transformation

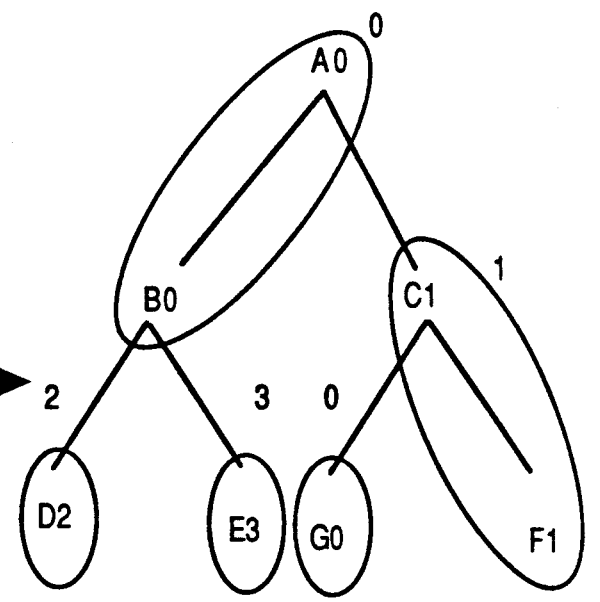


Fig 4.18b : L'arbre dépendance avec transformation

En effet le nombre maximum d'expressions pouvant être évaluées en parallèle dans la figure 4.18a est de 3 (dont les noms sont D0, G3 et E2 ou F2), de plus le nombre d'arcs entre les cercles est de 6. Cette répartition exploite mal les possibilités de la machine (parallélisme réduit) et produit un coût élevé de communication. Ces deux exemples montrent l'importance de la fonction de hachage globale.

Une autre alternative que l'ont peut utiliser pour faire face à une mauvaise fonction de hachage globale est d'utiliser une méthode de répartition dynamique. En effet d'après [Wats87], il apparait qu'une allocation dynamique des tâches à effectuer (les expressions à évaluer) est en général préférable à une allocation statique dans les langages déclaratifs. Une distribution statique a très peu de chances d'être la distribution idéale - c'est à dire celle qui permet d'avoir une charge distribuée uniformément-.

Dans une méthode de répartition dynamique [Iqba86], chaque noeud (à des instants réguliers ou sur événement) génère des paquets de contrôle informant les autres noeuds de sa charge. De cette manière, tous les noeuds du réseau disposent d'une structure de données qui contient pour chaque noeud la charge associée. Lorsque la différence entre la charge d'un noeud et celle du noeud le moins chargé dans le réseau dépasse un certain seuil, une partie de la charge du premier noeud est envoyée vers le deuxième. Cette opération est réalisée jusqu'à ce que la différence entre les charges soit inférieure au seuil. L'inconvénient dans une méthode de répartition dynamique est l'existence d'un overhead en temps nécessaire au traitement des paquets de contrôle.

Est ce que la méthode de transformation du programme est facilement réalisable? ou bien faut-il implémenter une méthode de répartition dynamique?. Ces questions font l'objet de travaux de recherche faits par les autres membres de l'équipe N-ARCH.

Pour le moment aucune de ces deux alternatives n'a été implémentée dans le noyau. La méthode de répartition que nous avons utilisée pour les programmes de test que nous allons présenter au chapitre V, est une méthode de répartition statique qui repose uniquement sur l'utilisation d'une fonction de hachage (sans transformation). Les 2 conditions énumérées précédemment ne sont réalisées que partiellement (charge

moyennement répartie, coût de communication moyen) et cela pour un type particulier de programmes.

La technique que nous pensons implémenter prochainement pour remédier à une mauvaise distribution - c'est à dire lorsque la fonction de hachage globale utilisée n'est pas adaptée au programme à exécuter- est une méthode que nous pouvons classer entre la méthode statique et la méthode dynamique. Nous l'avons nommée "méthode de répartition par taux de remplissage".

La méthode de répartition par taux de remplissage

L'utilisation de la méthode de répartition par taux de remplissage vise à pallier une mauvaise répartition du programme obtenue par une fonction de hachage globale non adaptée aux noms d'expression du programme.

Dans cette méthode de répartition, le processus mémoire associative utilise durant les opérations de stockage une constante K (0 < K <= 1) identique pour tous les noeuds. Cette constante est appelée *taux de remplissage* et désigne le pourcentage de la mémoire associative des programmes qui peut être utilisé pour stocker les expressions durant la phase primaire comme nous allons le voir.

Un paquet initialisation peut avoir, dans ce cas, 4 types différents:

1) *Initialisation primaire en routage.*
2) *Initialisation primaire en collision.*
3) *Initialisation secondaire en routage.*
4) *Initialisation secondaire en collision.*

Le programme soumis à l'exécution est traité par le processeur hôte afin de répartir l'ensemble des expressions sur le réseau. Les paquets initialisation générés par l'hôte sont dans ce cas du type *"initialisation primaire en routage"*. Lorsqu'un paquet initialisation primaire en routage a atteint le noeud destination calculé par la fonction de hachage globale, le processus mémoire associative des programmes compare la valeur de (K * N), où K est le facteur de remplissage et N est le nombre total d'expressions pouvant être mémorisées (nombres d'entrée dans la mémoire des programmes), au nombre d'expressions déjà stockées dans la mémoire (noté P). Si (K * N) > P, ceci implique que le taux de remplissage n'a pas été encore atteint. Le paquet est alors mémorisé et la variable P est incrémentée de 1.

Si par contre (K * N) < ou = P le taux de remplissage a été atteint (collision primaire), il faut alors reporter la mémorisation de ce paquet sur un des voisins. A ce moment le type du paquet passe en initialisation primaire en collision et il est expédié vers le voisin désigné par application de la fonction de hachage locale sur le nom de l'expression. Au niveau du noeud voisin l'opération de comparaison de (K * N) par rapport à P est de nouveau réalisée et le même traitement est effectué. Cette opération se répète jusqu'à ce que le paquet soit dans une de ces 2 situations:

* Un noeud du réseau a son (K * N) supérieur à P, il peut donc mémoriser le paquet.

* Le paquet primaire en collision a atteint le dernier noeud du réseau sans qu'il ait été mémorisé (tous les noeuds ont dépassé leur taux de remplissage).

Dans le deuxième cas, le processus de contrôle de ce dernier noeud a la tâche de changer le type du paquet à *initialisation secondaire en routage.* Ce paquet est alors envoyé de nouveau vers le noeud désigné par son champ noeud destination en utilisant l'algorithme de routage. Lorsque ce noeud reçoit le paquet initialisation secondaire en routage, le processus mémoire associative compare cette fois N par rapport à P. C'est donc toute la mémoire associative qui est testée cette fois pour trouver une nouvelle entrée pour la paquet reçu.

Si N > P l'expression est mémorisée. Si par contre N = P (collision secondaire) le paquet est aiguillé vers l'un des voisins en utilisant la fonction de hachage locale, et le type devient alors *initialisation secondaire en collision.* Le même mécanisme se répète au niveau du noeud voisin (toujours en comparent N à P) jusqu'à ce qu'un noeud accepte de mémoriser l'expression (on suppose, bien sûr, qu'il y a dans le réseau suffisamment d'espace pour mémoriser tout le programme).

Au début de ce paragraphe nous avons introduit le facteur K sans préciser de quelle manière il était calculé.

Nous pensons que la meilleure manière de le faire c'est d'utiliser les indications de l'utilisateur. En générale un utilisateur a une idée sur le degré de parallélisme du programme qu'il va présenter à la machine (largeur de l'arbre de dépendance). Afin de simplifier la tâche de l'utilisateur, 4 valeurs peuvent être autorisées pour K.

	K	Degré de parallélisme
1er cas	1	très peu ou pas de parallélisme
2em cas	0,75	peu de parallélisme
3em cas	0,50	quantité moyenne de parallélisme
4em cas	0,25	beaucoup de parallélisme

En utilisant la valeur de K indiqué par le programmeur un paquet de contrôle est diffusé à tous les noeuds. Ce paquet est envoyé avant le chargement du programme et contient la valeur de K.

Les études que nous avons entreprises montre que l'utilisation de la méthode du taux de remplissage permet une bonne répartition même lorsque la fonction de hachage globale est mal adaptée au programme.

Le prix à payer dans cette méthode de correction de la fonction de hachage, c'est bien sûr l'implication du programmeur dans un détail du fonctionnement de la machine et une augmentation du nombre de paquets en collision. Pour cela un mécanisme de communication rapide des paquets en collision est nécessaire (le temps de calcul de la fonction de hachage locale ne doit pas être important).

IV.5.2: La fonction de hachage locale [Bour88]

Dans un noeud N-ARCH lorsque le taux de remplissage de la mémoire associative des programmes est atteint, les prochains paquets initialisation qui arrivent à ce noeud seront aiguillés vers l'un des voisins. Ce noeud est appelé noeud racine. Le choix du noeud suivant parmi les noeuds physiquement reliés se fait en appliquant la fonction de hachage locale sur le nom de l'expression contenu dans le paquet initialisation reçu.

Le même procédé se réitère si nécessaire au niveau du noeud voisin jusqu'à obtention d'un noeud capable de contenir l'expression. L'ensemble des noeuds accédés à partir d'un noeud racine par ce mécanisme peuvent

être représentés sous forme d'un arbre. Cet arbre est appelé *"arbre de collision"* relatif à ce noeud.

Chaque noeud du réseau doit figurer une et une seule fois dans toute branche de l'arbre de collision. Ceci permet d'utiliser toutes les capacités mémoire de la machine en cas de collisions et évite des accès aux noeuds déjà parcourus.

Le type d'arbre qui convient le mieux avec la topologie du réseau ici choisi (l'hypercube), et qui satisfait aux conditions précédentes sont les arbres d'exploration de graphe.

Dans ce genre d'arbre, chaque branche est un chemin Hamiltonien de point de départ le noeud racine de l'arbre de collision. La figure 4.19 donne l'arbre d'exploration de graphe de racine 0 dans un hypercube d'ordre 2.

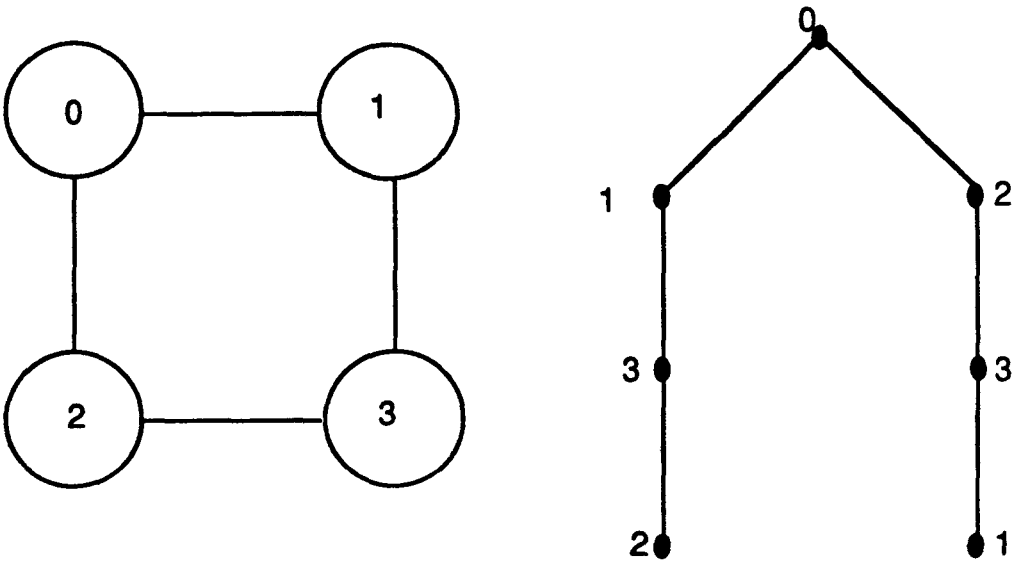


Fig 4.19 : Un hypercube d'ordre 2 et son arbre de recouvrement de racine 0

Contrairement à d'autres types d'arbres, le choix du noeud suivant en cas de collision se fait, comme on peut le voir dans la figure, sur tous les noeuds physiquement connectés au noeud. De plus, lorsqu'une collision se produit dans un noeud feuille de l'arbre de collision, il n'est plus nécessaire d'envoyer le paquet à un nouveau noeud, car tous les noeuds ont été accédés.

Remarque: Comme on peut le voir sur l'exemple précédent les arbres d'exploration de graphes ne permettent pas d'avoir des arbres équilibrés. En réalité, il est impossible de construire des arbres équilibrés sur un réseau en hypercube .

Le calcul des différents chemins hamiltoniens est réalisé dans le noyau à l'aide des codes GRAY (CG). Pour définir un CG nous avons besoin d'un certain nombre de définitions que nous allons introduire.

a- Le rang partiel

Soit $G=\{g_1, g_2, \ldots\ldots, g_n\}$ où g_i est pris dans N* et $g_i <> g_j$ pour tout i et j distincts. On définit le rang partiel r_i de g_i comme étant le rang de g_i dans $\{g_1, g_2, \ldots\ldots, g_i\}$ quand G est trié dans l'ordre croissant.

Exemple: $\{g_1, g_2, g_3, g_4\} = \{ 4, 2, 3, 1\}$

$r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 1$

b- Noeuds adjacents

Deux noeuds sont adjacents si et seulement si leurs numéros (exprimés en binaire) différent d'un seul bit (leur distance de Hamming est de 1).

Notation:

Soit A un ensemble de chaînes binaires de longueur (n-1) , n > 1. On note $A^{b/k}$, où b est pris dans {0, 1}, l'ensemble des chaînes binaires obtenues en insérant le bit b dans chaque chaîne de A *après* la k-ième position. Les positions dans la chaîne binaire sont comptées à partir de la droite (position de poids faible). Si k = n, le bit b est mis dans la position du bit de poids faible de la chaîne.

Exemple:

A = { 00, 01, 11, 10} n-1 = 2 => n = 3

$A^{1/1}$ = { 0$\underline{1}$0, 0$\underline{1}$1, 1$\underline{1}$1, 1$\underline{1}$0}

$A^{0/3}$ = { 00$\underline{0}$, 01$\underline{0}$, 11$\underline{0}$, 10$\underline{0}$}

On note aussi A*, l'ensemble des chaînes binaires obtenues en inversant l'ordre des chaînes de A. En reprenant l'exemple précédent on a

A*= {10, 11, 01, 00} et $(A^{1/1})^*$= {110, 111, 011, 010}

On peut remarquer que $(A^*)^{i/j} = (A^{i/j})^*$.

Définition: Soit $\{g_1, g_2, ..., g_n\}$ une permutation de $Z_n = \{1, 2, 3, ..., n\}$, le code de GRAY est défini par :

$G_1 = \{ 0, 1\}$

$G_k = \{ G_{(k-1)}{}^{0/r_k} , G^*_{(k-1)}{}^{1/r_k} \}$

ou r_k est le rang partiel de g_k.

Les valeurs $\{r_1, r_2, ..., r_k\}$ sont appelées dans la suite du chapitre les paramètres du CG.

Exemple:

$\{g_1, g_2, g_3\} = \{ 3, 1, 2\}$

$\{r_1, r_2, r_3\} = \{1, 1, 2\}$

$G_1 = \{ 0, 1\}$ $\quad G_1^* = \{ 1, 0\}$

$G_2 = \{ G_1{}^{0/r_2}, (G_1^*)^{1/r_2}\} = \{ 00, 01, 11, 10\}$

$G_2^* = \{10, 11, 01, 00\}$

$G_3 = \{ 000, 001, 011, 010, 110, 111, 101, 100\}$

Nous venons de voir comment est calculé un chemin Hamiltonien cyclique de racine le noeud 0. Ce chemin est noté Gn. Les autres chemins Hamiltoniens (cycliques et non cycliques) peuvent être obtenus en réalisant des inversions des chaînes binaires comprises dans Gn à partir d'un noeud adjacent avec l'extrémité droite de ce chemin. Le numéro du noeud à partir duquel se fait cet inversion est appelé numéro d'inversion.

Exemple:

$G_3 = \{ 000, 010, 011, 001, 101, 111, 110, 100\}$

$H_1 = \{ 000, 100, 110, 111, 101, 001, 011, 010\}$, numéro d'inversion = 000

$H_2 = \{ 000, 010, 011, 001, 101, 100, 110, 111\}$, numéro d'inversion = 101

$H_3 = G_3$, numéro d'inversion = 110

Les chemins Hamiltoniens de racine un noeud quelconque M et pour les mêmes paramètres r_i, sont obtenus en faisant des "ou exclusifs" entre le numéro du noeud M et les numéros des noeuds constituant les chaînes binaires du chemin Hamiltonien de Gn.

Exemple:

Le chemin Hamiltonien de racine 2 (010_b) et de paramètres = {1, 1, 2} est :

{ 000, 010, 011, 001, 101, 100, 110, 111}

oux 010

{010, 000, 001, 011, 111, 110, 100, 101}

Un chemin Hamiltonien est donc défini à partir des informations suivantes:

* La racine du chemin Hamiltonien, c'est à dire la racine de l'arbre de collision. Cette information est obtenue à partir du champ noeud destination du paquet (initialisation ou requête) reçu.

* Les paramètres {r1, r2, ...rk} qui définissent le code GRAY, dont le nombre est égal au degré (d) de l'hypercube. Ces paramètres sont obtenus à partir du nom de l'expression définie (s'il s'agit d'un paquet initialisation) ou du nom de l'expression demandée (s'il s'agit d'un paquet requête), en utilisant une fonction de hachage quelconque. Dans la version actuelle du noyau la fonction de hachage que nous avons utilisée, consiste à diviser le nom de l'expression en champ de longueur 3 bits (à partir des poids faibles). Seuls les d premiers champs sont utilisés comme paramètres r_i. Le choix du chiffre 3 (qui limite le degré de l'hypercube à 7 et qui donne un réseau de 128 processeurs) peut être augmenté.

* Le numéro du noeud à partir duquel se fait l'inversion (le numéro d'inversion), qui est aussi obtenu à partir du nom de l'expression définie ou demandée. Ce numéro de noeud doit être, comme expliqué, adjacent à l'extrémité de Gn. On trouvera dans [Bour88] la méthode selon laquelle ce numéro est calculé, ainsi que le code de la procédure OCCAM qui implémente la fonction de hachage locale dans le noyau N-ARCH.

Cette procédure utilise 3 paramètres en entrée:

- Le numéro du noeud racine de l'arbre de collision.
- Le nom de l'expression définie ou demandée.
- Le numéro du noeud actuel.

Les premier et deuxième paramètres sont utilisés pour calcul le chemin Hamiltonien comme expliqué précédemment. Le troisième paramètre est lui utilisé pour sélectionner le noeud suivant dans le chemin.

La procédure donne comme résultat:

- Un booléen qui est à vrai s'il y a un noeud suivant dans le chemin, ou à faux si le noeud actuel est une feuille de l'arbre de collision (dernier noeud du chemin).
- Le numéro du noeud suivant.

Remarques:

1- La fonction de hachage locale présentée, permet de calculer dynamiquement le numéro du noeud successeur. Ceci n'était pas possible si on avait utilisé une table de collision (méthode statique). De plus pour avoir une répartition uniforme sur tous les voisins avec cette table de collision, il aurait fallu réserver une zone de taille importante.

2- Le numéro du noeud suivant calculé par la fonction de hachage locale est 1 parmi d s'il s'agit de la première collision (numéro du noeud racine = numéro du noeud actuel). Si non ce numéro est 1 parmi (d-1).

Afin de simplifier les calculs et permettre un gain de temps à l'exécution, la procédure OCCAM qui réalise la fonction de hachage ne calcule pas tout le chemin à chaque collision, mais uniquement le prédécesseur et successeur dans le CG du noeud où la collision a eu lieu. En fonction du numéro du noeud à partir duquel se fait l'inversion, le résultat sera ou bien l'un ou l'autre.

Malgré cette amélioration, les tests réalisés ont montré que le temps de calcul de la fonction de hachage locale sont très longs (pour un hypercube de degré 4 le temps de calcul peut atteindre jusqu'à 400 micro-secondes). Ce temps de calcul a une grande influence sur les performances de la machine (augmentation du coût des communications pour les paquets en collision). De plus pour implémenter la méthode de répartition par taux de remplissage, il est important de réduire ce temps de calcul car il y aura un très grand nombre de paquets en collision dans le réseau.

Pour cela une étude a été menée afin de réaliser le calcul de la fonction de hachage locale par matériel. Aprés avoir conçu le circuit qui réalise ce calcul à l'aide de portes logiques et un réseau de PLA, une simulation de ce circuit a été réalisée en utilisant le langage de description de matériel LIDO [Bako87].

Dans le cas le plus défavorable, le temps de calcul de la fonction de hachage en utilisant ce simulateur est de 40 cycles. L'utilisation d'une

fréquence de 5 MHZ permet de réduire ce temps de calcul jusqu'à 8 micro-secondes. Ceci représente un gain de 50.

Références citées dans le chapitre :

[Atki87], [Bako87], [Bour88], [Burn86], [Burn88], [Gonc87], [Heid87], [Inmos87], [Iqba86], [Lewi88], [Munt88], [Niar87], [Pout87], [Silb88], [Tour86], [Wats87].

CHAPITRE V
TESTS ET MESURES DE PERFORMANCES DU NOYAU N-ARCH

Après avoir développé le noyau de l'émulateur en langage OCCAM pour la machine N-ARCH, nous avons réalisé des opérations de tests et de mesures de performances sur un ensemble de programmes. Ces programmes ne comportent pas de fonctions ni de références à des éléments d'une liste, mais uniquement des expressions composées d'un certain nombre d'arguments. Exemple: A = B * (C + 1).

Par conséquent, il n'est pas possible de fournir des résultats concernant l'exécution de programmes de tests (ou "benchmark") standards comme le programme "Quicksort".

Le matériel utilisé afin de réaliser ces tests est composé :

* D'un micro-ordinateur PC-AT contenant une carte Inmos B04. Cette carte contient un Transputer et joue le rôle d'interface avec le réseau. C'est aussi sur cette carte que se fait le développement et la mise au point des programmes OCCAM qui seront chargés plus tard dans le réseau. Dans la suite de ce paragraphe, cette carte est appelée *processeur Hôte* (Fig 5.1).

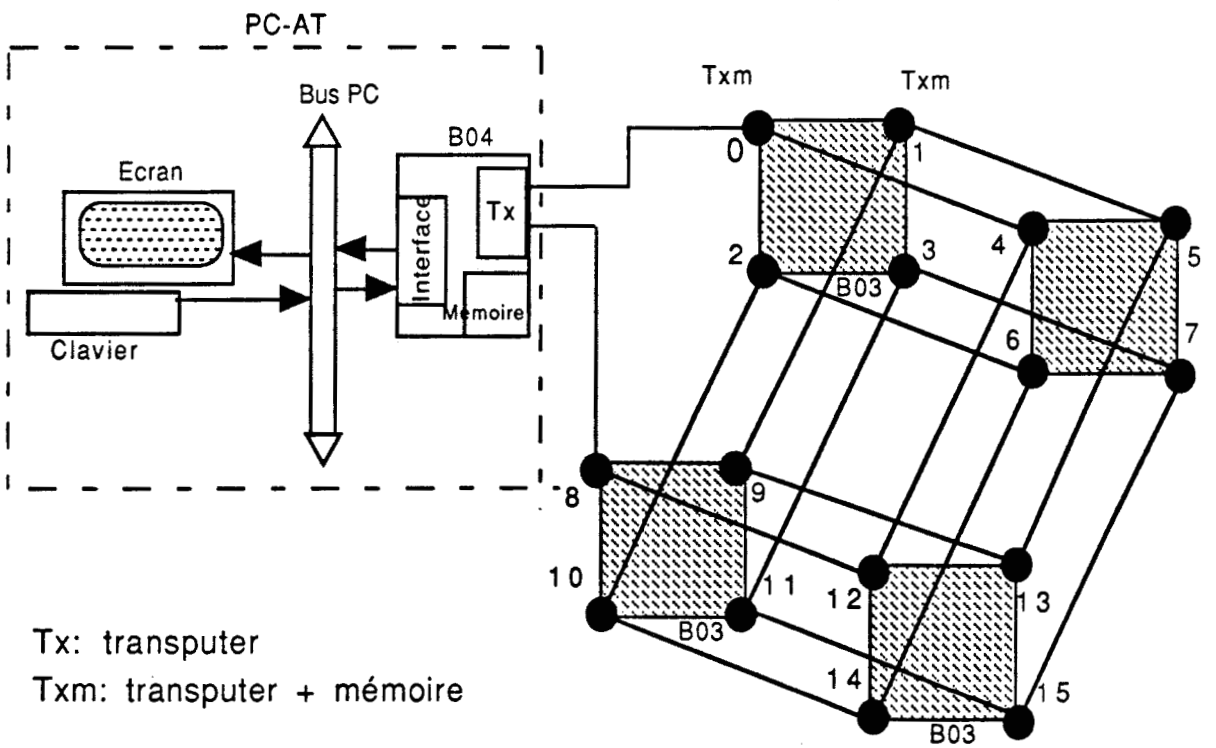


Fig 5.1 : La configuration matérielle utilisée

* De 4 cartes Inmos B03. Chacune de ces cartes contient 4 Transputers ainsi que 256 K-octets de mémoire privée pour chaque Transputer. Les 4 Transputers d'une même carte sont configurés en anneau (ou hypercube de degré 2). De ce fait, nous disposons dans chaque carte de 8 liens libres (2 par Transputer). Ceci nous a permis ainsi, de construire des hypercubes de degré:

- 0 en utilisant un seul Transputer.
- 1 en utilisant 2 Transputers d'une même carte.
- 2 en utilisant les 4 Transputers d'une carte.
- 3 en utilisant les 8 Transputers de 2 cartes.
- 4 en utilisant les 16 Transputers des 4 cartes.

V.1 Les programmes testés

V.1.1- Le langage de simulation

Afin de réaliser des opérations de test et voir quel est le comportement macroscopique de l'émulateur durant l'exécution d'un programme de nature déclarative, nous avons défini un langage de simulation. Ce langage permet de construire des programmes composés uniquement d'expressions.

Sa syntaxe est la suivante:

<programme> ::= {<expression>}

<expression> ::= <nom> = <paramètre délai> {<argument>} | <paramètre délai> <entier>

<argument> := <nom>

<nom> ::= chaîne de caractères

<paramètre délai> ::= entier.

De cette manière, à chaque expression du programme sont associés :

* Un ensemble d'arguments. Ces arguments représentent les paramètres nécessaires au calcul de l'expression.

* Un facteur de temps. Ce facteur représente le temps nécessaire à la production du résultat après que les valeurs des arguments aient été reçues. Il est appelé *paramètre délai* associé à l'expression.

Exemple : A = 3 B C D E et E = 3 4 sont des expressions.

L'expression de nom A fait référence aux quatre arguments B, C, D, E et nécessite 3 unités de temps pour le calcul de A une fois que la valeur de

chacun des arguments est connue. Afin de simplifier l'écriture nous appellerons les expressions par leur nom. En d'autres termes au lieu de dire l'expression dont le nom est A, on dira simplement l'expression A.

Le paramètre délai permet de voir l'importance du niveau de grain du parallélisme sur les performances de la machine. Ainsi une valeur importante pour ce paramètre correspond à un gros niveau de grain.

Pratiquement, nous avons utilisé l'horloge temps réel du Transputer pour implémenter ce délai. Sur le Transputer T414, cette horloge est incrémentée toutes les 64 micro-secondes et reprend la valeur 0 au bout de 74 heures (pour les processus de basse priorité).

Afin d'exécuter des programmes de taille importante, nous avons conçu un processus OCCAM qui permet de générer automatiquement un programme de test que nous pouvons charger et exécuter sur le réseau. Ce processus est exécuté par le processeur hôte et utilise les paramètres suivants :

1- Le nombre d'expressions que nous désirons mettre dans le programme à exécuter sur le réseau (nombre de noeuds et de feuilles de l'arbre de dépendance).

2- Le degré de l'hypercube sur lequel aura lieu l'exécution du programme.

3- Le paramètre délai qui est commun à toutes les expressions générées.

La figure 5.2 donne l'arbre de dépendance équivalent au programme

```
A0 = 1 B0 C0 D0 E0
B0 = 1 A1
C0 = 1 A2
D0 = 1 A3
E0 = 1 A4
A1 = 1 2
A2 = 1 67
A3 = 1 9
A4 = 1 4
```

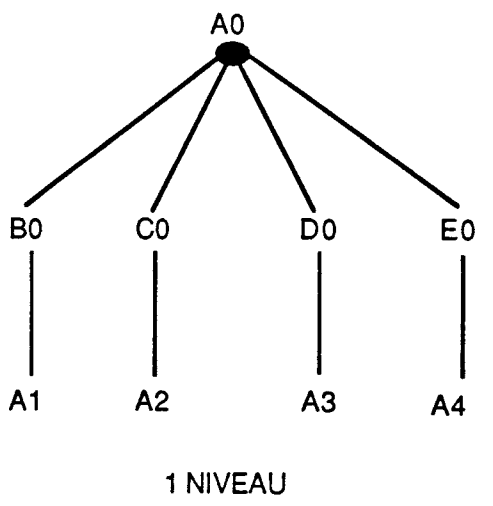


fig 5.2 Arbre de dépendance pour un programme de 9 expressions

généré par le processeur hôte avec les paramètres suivants: nombre d'expressions = 9, degré de l'hypercube = 2, délai = 1.

Nous appellerons dans la suite de ce chapitre, *boule* le sous arbre constitué de la racine Ai et des feuilles {Bi, Ci, Di, ...} si Ai n'est pas une feuille de l'arbre, ou bien seulement de Ai, si ce dernier se trouve comme feuille de l'arbre de dépendance. Dans la figure 5.4 nous avons regroupé dans un cercle les noeuds qui constituent une boule.

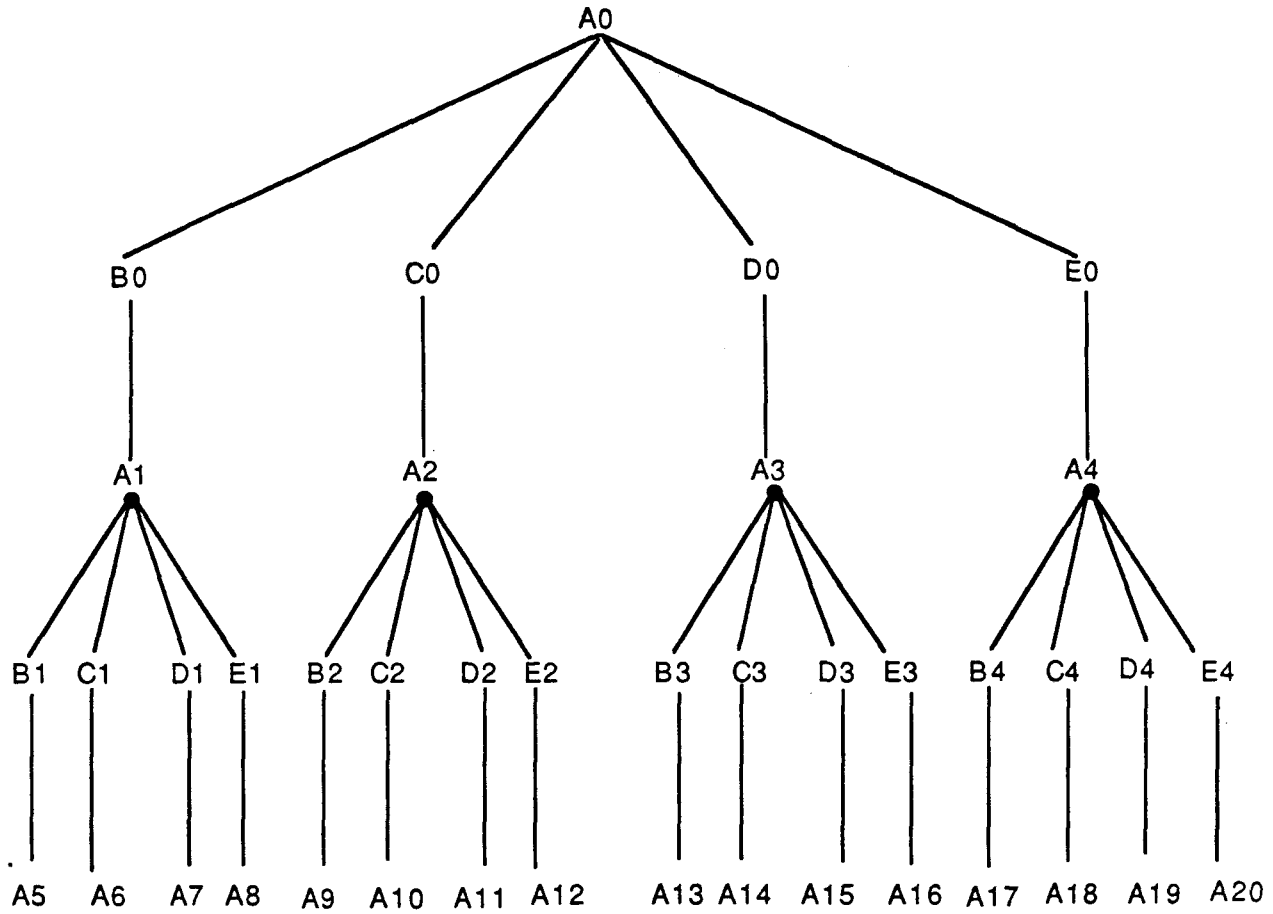


Fig 5.3 Arbre de dépendance équivalent au programme composé de 41 expressions

V.1.2 - La répartition des programmes

Lorsque le nombre de processeurs dans le réseau est supérieur à 1, il est possible de répartir le programme ainsi constitué, de plusieurs manières. Deux méthodes ont été testées sur l'émulateur. Ces deux méthodes correspondent, en fait, à deux fonctions de hachage globale.

La première fonction de hachage globale utilisée répartit le programme dans le réseau boule par boule. Ainsi, si une boule contient plusieurs

expressions, toute la boule sera stockée sur le même noeud si ce dernier dispose d'espace mémoire suffisant. Cette méthode est schématisée dans la figure 5.4 (nous avons supposé que le nombre de processeurs est 8). Chaque boule est représentée par un cercle qui est accompagné (en haut à droite) du numéro du noeud destination calculé par cette première fonction de hachage globale. C'est ainsi que les expressions du programme ont été réparties de la manière suivante:

Numéro du processeur	0	1	2	3	4	5	6	7
Nombre d'expressions	7	7	7	7	7	7	2	2

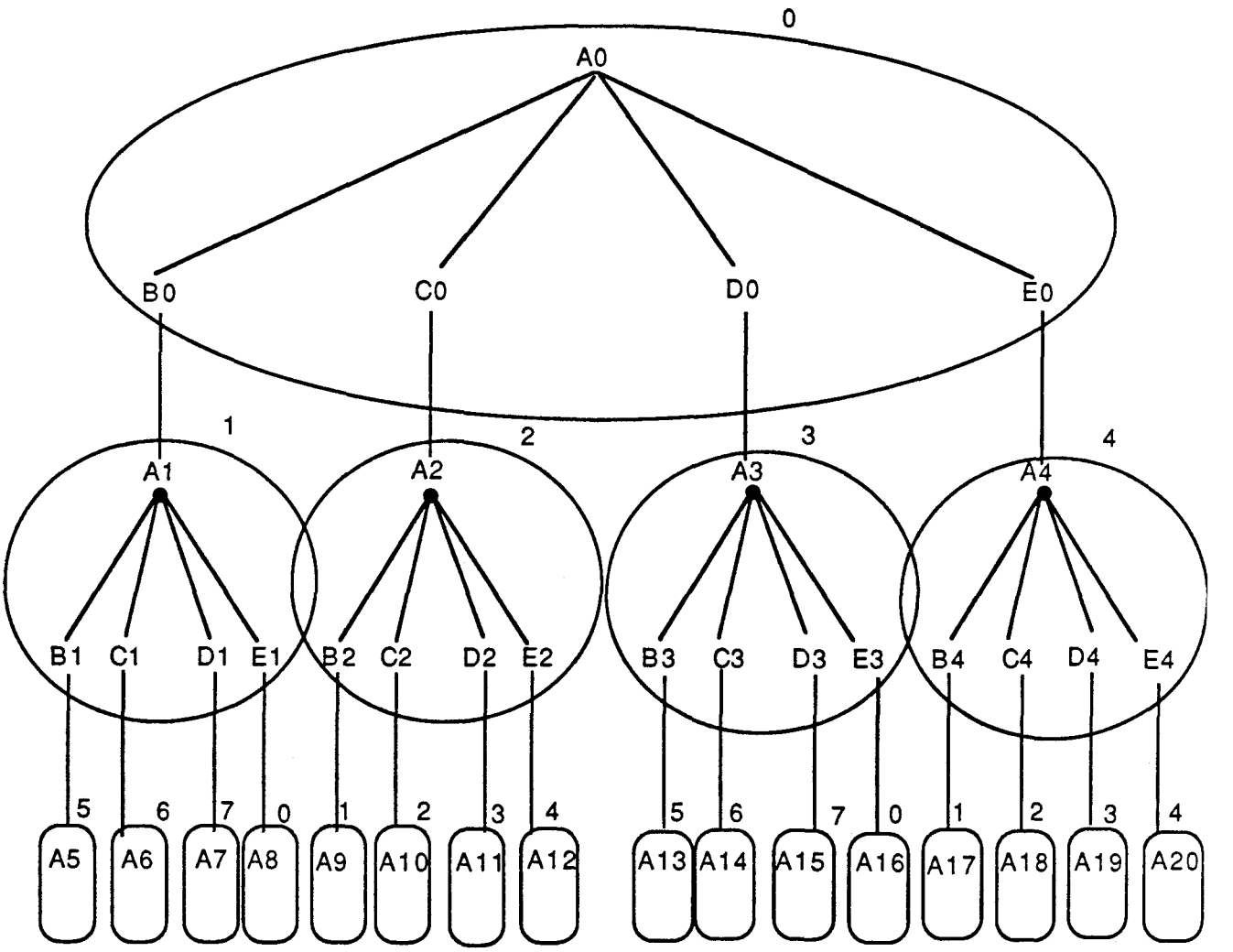


Fig 5.4 : Répartition d'un programme composé de 41 expressions avec la première fonction de hachage

Comme on peut le voir, cette méthode de répartition permet d'avoir un traitement local (séquentiel) pour les expressions d'une même boule et un traitement parallèle des différentes boules.

La deuxième méthode de répartition permet, elle, de distribuer les expressions d'une boule sur des processeurs différents (fig 5.5) sauf pour la racine Ai et le fils gauche {Bi} qui seront mis sur le même noeud.

De cette manière, si les expressions Ai et Bi sont stockées dans le noeud j, les expressions Ci, Di, Ei , par exemple, seront mises dans les noeuds j+1, j+2, j+3 , modulo le nombre de processeurs dans le réseau.

Cette méthode de répartition permet un traitement parallèle des expressions à l'intérieur d'une boule. Lorsque le nombre de processeurs dans le réseau est important, cette méthode ne répartit pas les expressions de manière équilibrée.

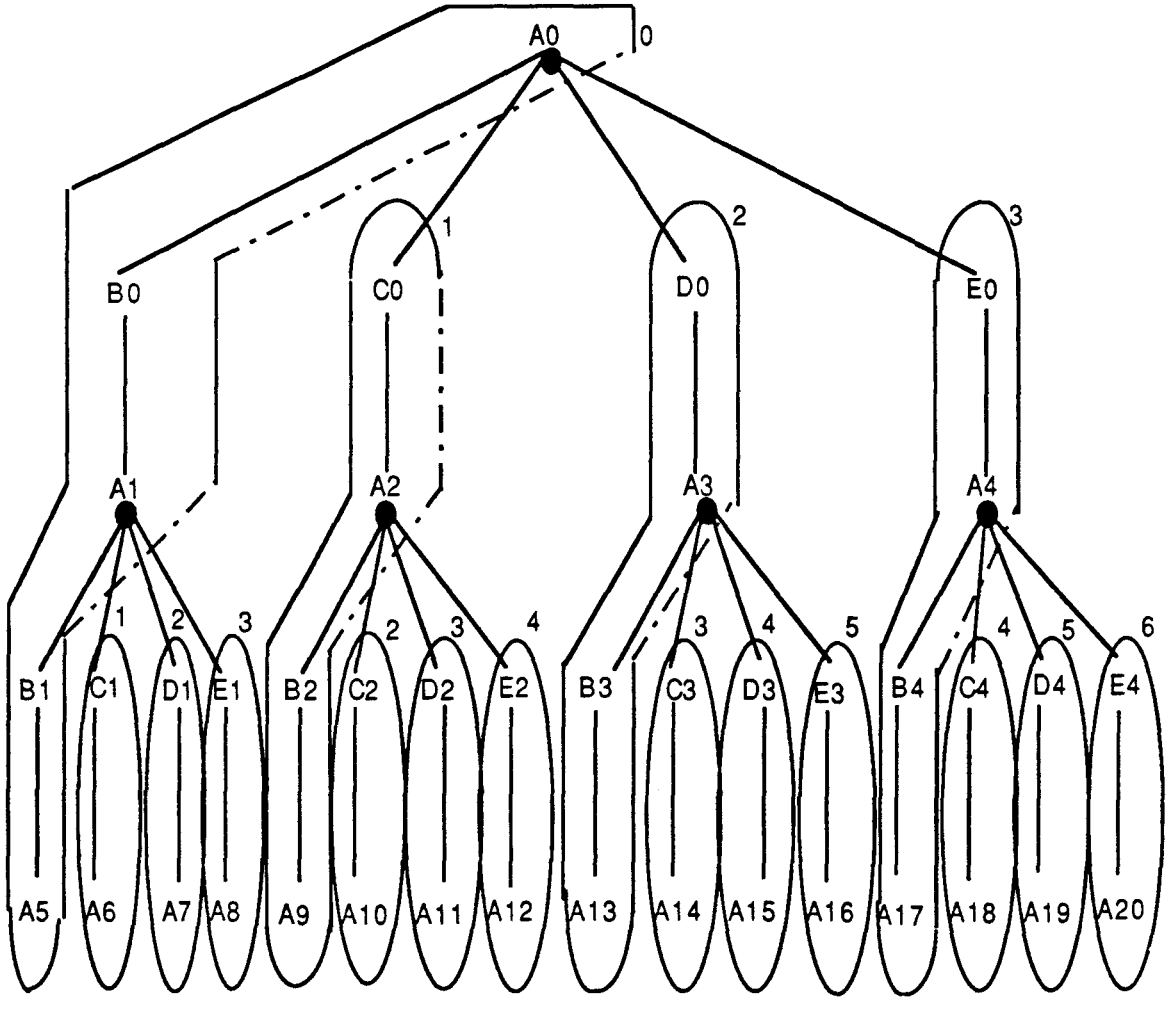


Fig 5.5 : Répartition d'un programme composé de 41 expressions en utilisant la deuxième méthode de répartition

Le tableau ci-après donne la répartition de la charge obtenue dans le cas du programme composé de 41 expressions exécuté sur un réseau de 8 processeurs. nous pouvons remarquer que le noeud 3 contient 10 expressions alors que le noeud 7 n'en contient aucune.

Numéro du processeur	0	1	2	3	4	5	6	7
Nombre d'expressions	5	6	8	10	6	4	2	0

La première méthode, par contre, est relativement intéressante lorsqu'on utilise un nombre important de processeurs. Dans la suite de ce chapitre c'est donc cette méthode qui est utilisée, car le projet tend à intégrer un grand nombre de processeurs.

Il est peut être nécessaire à ce niveau de préciser quelles sont les étapes de l'exécution d'un programme et ce qu'on l'on entend par le terme temps d'exécution.

V.1.3 - L'exécution d'un programme

L'exécution d'un programme se fait en 3 étapes :

1- Le processeur hôte génère les paquets initialisation et les transmet vers le noeud 0 ou le noeud 8 (si la valeur du champ noeud destination est supérieure ou égale à 8). Ces paquets seront ensuite diffusés dans le réseau si c'est nécessaire.

2- Le processeur hôte génère un paquet requête dont le champ nom de l'expression demandée est le nom de l'expression racine de l'arbre de dépendance à évaluer. Ce champ est généralement égal à A0. Dès que ce paquet est envoyé, il y a déclenchement d'une horloge.

3- Le hôte reçoit le paquet réponse correspondant à la requête précédemment envoyée. L'horloge est alors arrêtée et sa valeur représente le temps d'exécution. Le temps de chargement du programme n'est donc pas compris dans le temps d'exécution.

Remarques :

a: Une fois la phase 1 réalisée, seuls des paquets requête ou réponse peuvent transiter dans le réseau.

b: D'après la figure 5.3, il apparaît que dans les programmes que nous allons exécuter il n'y a pas de partage de résultat d'une expression. Par conséquent, toute expression recevra une et une seule requête et générera une et une seule réponse. De ce fait le nombre total de paquets requête et réponse qui seront émis dans le réseau est 2 fois le nombre d'expressions constituant le programme.

Dans le paragraphe IV.4 nous avons étudié la structure du noyau, et nous avons passé en revue les 3 approches utilisées pour simuler les mémoires associatives des programmes et des paquets. Celle qui produit le minimum d'overhead en temps est la méthode par utilisation d'une fonction de hachage. Chaque fois que nous ferons référence aux deux mémoires associatives, nous supposerons que c'est cette méthode qui est utilisée.

V.2 L'accélération théorique [Niar89]

Les courbes que nous allons présenter dans les paragraphes suivants montrent la diminution du temps d'exécution des différents programmes testés au fur-à-mesure que le nombre de transputers dans le réseau augmente.

V.2.1 Définitions

Nous appelerons "accélération réalisée par un réseau de N transputers sur un programme contenant P expressions", le rapport entre le temps d'exécution du programme composé de P expressions sur un transputer, et le temps d'exécution du même programme sur un réseau contenant N transputers. Comme le réseau que nous allons utiliser a toujours la forme d'un hypercube, ce chiffre N est une puissance entière de 2. L'accélération est notée " ACC_m (N, P) ".

$ACC_m (N, P) = \frac{T_{exec1}}{T_{execn}}$ en supposant que c'est le même programme qui est exécuté dans les deux cas.

Nous appelerons *accélération théorique*, l'accélération calculée en faisant une analyse du système. Dans cette analyse un nombre important de détails ne peuvent pas être pris en compte tel que: le temps pris par les changements de contexte, le temps de communication, le temps nécessaire à l'initialisation des contrôleurs de liens, le temps de routage,

...etc. De plus le calcul de l'accélération théorique se fait en supposons une répartition uniforme de la charge (la différence entre les charges des différents processeurs est nulle).

V.2.2 Le calcul de l'accélération théorique

Calculons maintenant cette accélération théorique pour un programme composé de P expressions et exécuté sur un réseau contenant N processeurs.

Le schéma fonctionnel de la figure 5.6 montre que le noyau peut être décomposé en deux grandes unités. Ces unités sont l'unité de traitement et de mémorisation, et l'unité de communication.

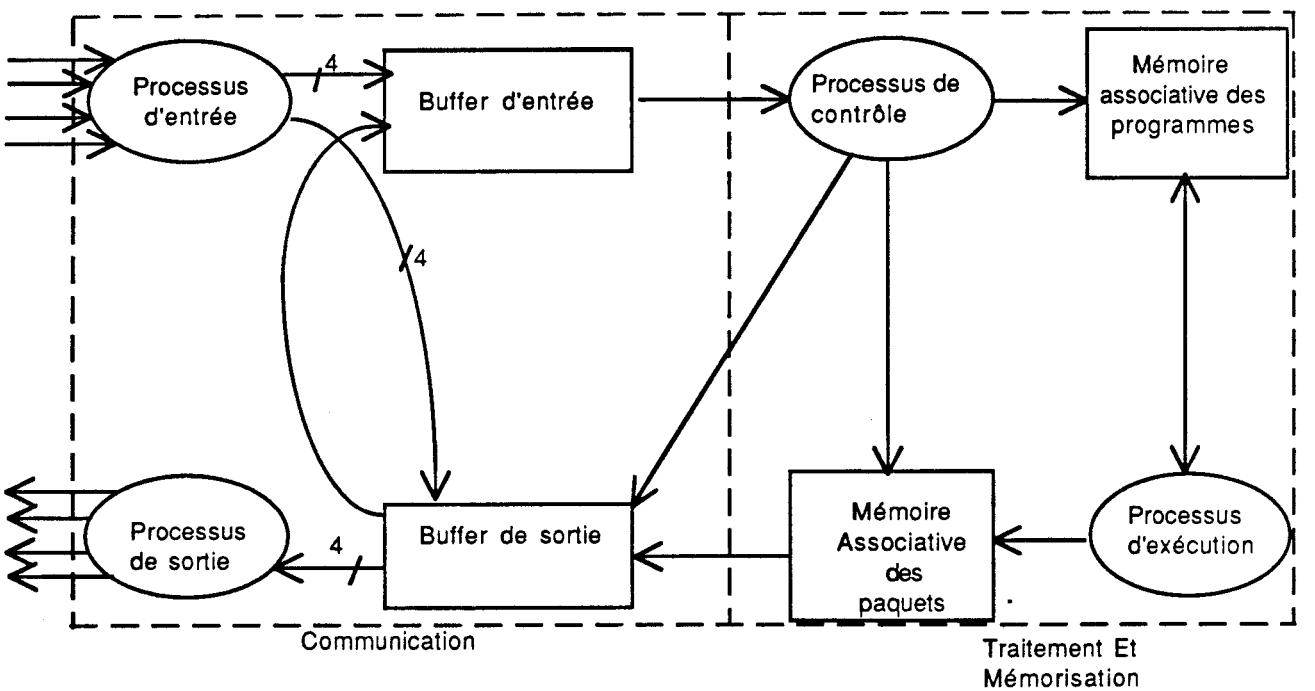


Fig 5.6 : Noyau = Traitement et Mémorisation + Communication

Le temps d'exécution d'un programme par un processeur (noté T_{exec1}) peut aussi être divisé en deux parties: un temps de traitement et de mémorisation et un temps de communication.

On peut donc écrire dans le cas où l'exécution se fait sur un seul processeur

$T_{exec1} = T_{trait1} + T_{comm1}$

Le temps T_{comm1} est composé du temps pris par la l'unité centrale (CPU) du transputer pour réaliser des émissions et réceptions de

messages (T_{trans1}) plus le temps nécessaire aux opérations de lecture et d'écriture dans les buffers d'entrée et de sortie (T_{buff1}).

Ce temps T_{buff1} correspond à la gestion de P paquets requêtes et (P-1) paquets réponses par le buffer d'entrée, et de P paquets requêtes et P paquets réponses par le buffer de sortie.

Lorsque tout le programme se trouve mémorisé sur un seul noeud le temps T_{trans1} est nul car les liens externes ne sont pas utilisés.

T_{trait1} est le temps nécessaire au processeur qui contient le programme pour réaliser le traitement des paquets. Ce temps est lui aussi composé du temps nécessaire aux opérations de lecture et d'écriture dans les mémoires associatives (T_{am1}) et le temps correspondant à l'exécution des opérateurs (T_{e1}). Le temps T_{am1} correspond à P accès à la mémoire des programmes en recherche d'expressions, (P-1) accès à la mémoire des programmes en mise à jour d'argument, P accès à la mémoire des paquets en écriture de paquets en attente et enfin P accès à la même mémoire en mise à jour de paquets réponse. Le temps T_{e1} peut être approximé par

T_{e1} = P * délai

En conclusion nous pouvons écrire:

$T_{comm1} = T_{buff1}$ et on peut écrire

$T_{trait1} = T_{am1} + T_{e1}$

Pour simplifier les calculs, le délai associé à chaque expression est nul. Par conséquent nous avons: $T_{e1} = 0$

$T_{exec1} = T_{comm1} + T_{trait1}$

$T_{exec1} = T_{am1} + T_{buff1}$

Lorsqu'un programme est découpé en tâches pour être exécuté sur un ensemble de processeurs, son temps d'exécution est égal alors au temps d'exécution de la plus grande tâche. Ce temps Texecn est composé de 4 temps (T_{en}, T_{transn}, T_{buffn}, T_{amn}).

Dans ce cas, le temps T_{en} est toujours nul pour les mêmes raisons qu'en haut, mais T_{transn} ne l'est plus car dans ce cas il y a des paquets à émettre (ou à recevoir) vers les (ou des) voisins.

Si le programme composé de P expressions est réparti de manière équilibrée sur les N processeurs, la charge au niveau de chacun d'eux sera de P/N. De ce fait le temps d'exécution du programme est égal au temps d'exécution des P/N expressions sur un des N processeurs.

Si nous supposons que le temps d'accès à une donnée dans les deux mémoires associatives est constant (pas de collision mémoire), nous pouvons dire que le temps d'accès aux mémoires associatives T_{amn} est égale à T_{am1} divisé par N. Car chaque noeud contient P/N expressions, il recevra P/N paquets requêtes et (P-1)/N paquets réponses, et générera P/N paquets réponses. Ceci implique donc (2P-1)/N accès en mémoire des programmes et 2P/N accès en mémoire des paquets. Le temps nécessaire à la lecture et à l'écriture des paquets dans le buffer d'entrée et de sortie T_{buffn} est aussi divisé par N, car P/N paquets requête et P/N paquets réponses transiteront à travers chacun des buffers d'entrée et de sortie.

Remarque:

En réalité le temps d'accès aux processus mémoires associatives est fonction du nombre d'expressions mémorisées. Lorsque le nombre d'expressions mémorisées est important, il se produit un grand nombre de collisions (comme nous allons le voir après). Ceci a pour effet d'allonger le temps d'accès.

L'existence de contrôleurs de liens autonomes décharge la CPU du transputer de la gestion des opérations d'entrée et de sortie sur les canaux externes. Il y a donc un parallélisme au niveau interne entre les opérations de traitement et de mémorisation, et les opérations de transmission et de réception des messages. On peut dire que le temps d'exécution (T_{execn}) , dans le cas où le nombre de processeurs est supérieur à 1, est égal à:

$$T_{execn} = MAX\ [\ \frac{(T_{buff1} + T_{am1})}{N},\ T_{transn}] = MAX\ [\ \frac{T_{exec1}}{N}\ ,\ T_{transn}]$$

Concernant cette formule, nous pouvons faire plusieurs remarques:

1) Le temps Ttrans est généralement faible devant le temps (T_{buff1} + T_{am1}) / N, même si N est important. A cela il y a plusieurs raisons:

a- Une vitesse de transmission importante au niveau des canaux externes (10 méga-bits/seconde).

b- L'existence de 4 contrôleurs de liens indépendants et autonomes. Les 4 contrôleurs peuvent fonctionner en parallèle et permettent donc, de diminuer le temps T_{transn}.

c- La longueur des paquets transmis entre les noeuds est faible, ne dépasse pas 6 mots de 32 bits.

2) L'opération d'émission ou de réception d'un message sur un canal n'est réalisée, comme nous venons de le voir, que lorsque la CPU du

transputer a initialisé ce dernier en lui fournissant un pointeur vers l'espace de travail du processus qui désire réaliser l'opération de transfert, un pointeur vers l'adresse où est mis (ou sera mis) le message à envoyer (ou à recevoir), et enfin un compteur d'octets initialisé par le nombre d'octets à émettre ou à recevoir.

3) Dans la dernière formule nous n'avons pas pris en compte le temps nécessaire au routage des paquets arrivés au noeud, mais qui sont destinés à un autre noeud. Nous avons en quelque sorte supposé qu'il y a un maillage complet entre les noeuds.

Ces remarques nous conduisent à dire que le temps T_{transn} est faible par rapport au temps $\frac{T_{exec1}}{N}$.

En conclusion, nous pouvons dire que le temps d'exécution théorique d'un programme de P expressions sur un réseau de N processeurs est :

$$T_{execn} = \frac{(T_{buff1} + T_{am1})}{N} = \frac{T_{exec1}}{N}$$

L'accélération théorique est donc $ACC_m(P, N) = N$ (1)

V.2.3 L'accélérations théorique matérielle et l'accélération théorique logicielle

A ce niveau, il est intéressant de remarquer que la formule (1) ne prend pas en compte le nombre d'expressions qui existent dans le programme et qui peuvent être évaluées en parallèle.

En d'autres termes cette formule donne l'accélération théorique matérielle (appelée aussi limite matérielle ou hardware bound) [Eage89]. Nous pouvons définir de la même manière l'accélération théorique logicielle (appelée aussi limite logicielle ou software bound notée $ACC_L(P)$) qui exprime l'accélération obtenue en exécutant un programme contenant en moyenne M expressions pouvant être évaluées en parallèle et en supposant que le nombre de processeurs dans le réseau est infini.

Si on représente le programme par un arbre de dépendance (fig 5.7), on peut définir le nombre M qui représente le taux de parallélisme moyen dans le programme et qui est égale à la somme des expressions dans chaque niveau de l'arbre divisé par le nombre de niveau de l'arbre de dépendance. En d'autres termes, M est égale au nombre totale d'expressions dans le programme divisé par le nombre de niveau de

l'arbre. Pour le programme équivalent à l'arbre de dépendance de la figure 5.7, le taux de parallélisme moyen est = $\frac{1+3+5}{3} = \frac{9}{3} = 3$.

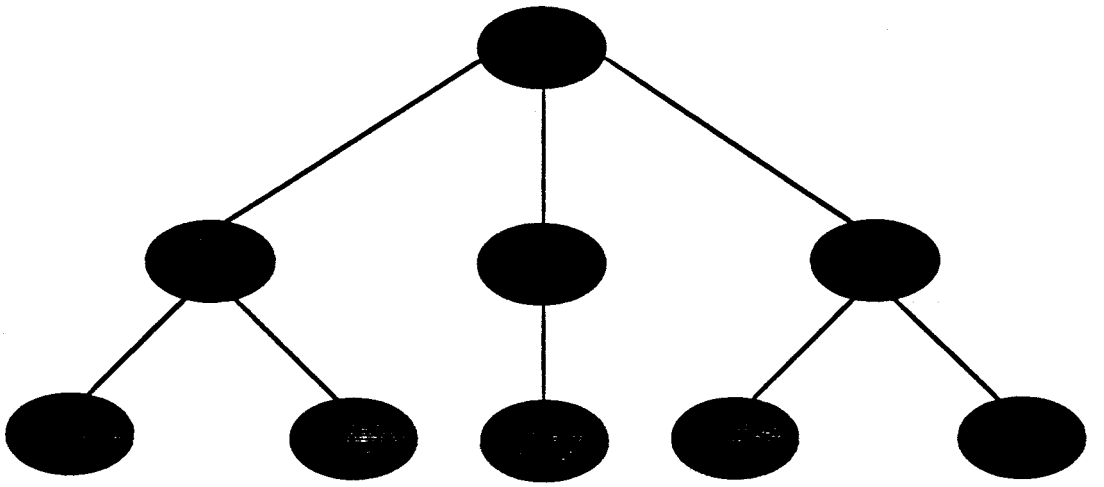


Fig 5.7 Un arbre de dépendance représentant un programme contenant 9 expressions

Si on suppose que chaque expression consomme pour son exécution le même temps CPU, l'accélération théorique logicielle est égale au nombre total d'expressions dans le programme divisé par le nombre d'expressions se trouvant dans la branche la plus longue de l'arbre de dépendance. Ce dernier est égal aussi au nombre de niveau de l'arbre de dépendance. Par conséquent, on peut dire que l'accélération théorique logicielle d'un programme contenant en moyenne M expressions pouvant être évaluées en parallèle est égale à M.

Pour les programmes testés, dont la forme générale à été donnée dans le paragraphe précédent, nous pouvons dire que l'accélération théorique logicielle est une fonction strictement croissante de P.

L'accélération réelle (notée ACC (P, N)), celle calculée en faisant des exécutions de programmes sur une architecture physique, est dans tous les cas inférieure à l'accélération théorique matérielle et l'accélération théorique logicielle. De ce fait, la courbe de l'accélération réelle est limitée par l'aire se trouvant entre les accélérations théoriques matérielle et logicielle et l'axe des abscisses.

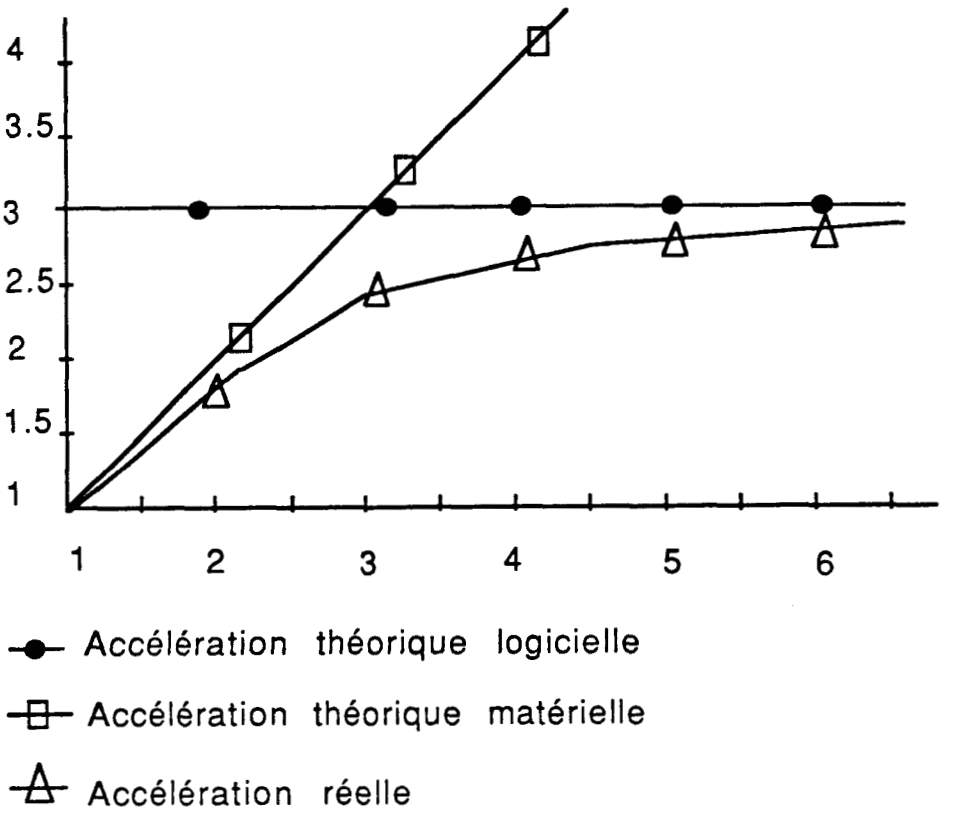


Fig 5.8 Les accélérations théoriques et réelles

Les courbes de la figure 5.8 représente l'accélération théorique matérielle et logicielle pour l'arbre de dépendance de la figure 5.7, ainsi qu'un exemple d'accélération réelle.

Dans le cas des systèmes multiprocesseurs, une autre entité qui est largement utilisée (pour caractériser le coût de la machine), mais que nous n'utiliserons pas souvent, est l'efficacité Eff(P, N). L'efficacité d'un système contenant N processeurs est définie comme étant le taux moyen d'utilisation de ces N processeurs.

Pratiquement l'efficacité est calculée par $\text{Eff}(P, N) = \frac{\text{ACC}(P, N)}{N}$.

L'efficacité pour un système monoprocesseur est 1, et cette valeur de l'efficacité décroît lorsque le nombre de processeurs augmente. Ceci en raison des temps de communications et l'impossibilité de toujours fournir du travail au processeurs pour les occuper entièrement.

V.3 Les temps d'exécution [Gonc89]

Dans ce paragraphe nous allons présenter un certain nombre de courbes. Sur chacune de ces courbes, nous avons représenté sur l'axe des abscisses le nombre de processeurs de l'hypercube sur lesquels se fait l'exécution du programme. Sur l'axe des ordonnées nous avons reporté l'accélération réelle.

Les programmes que nous avons testé sont constitués de boules contenant un noeud racine A_i et de 4 fils {Bi, Ci, Di, Ei}.

V.3.1 Les temps d'exécution sans collision

Dans ce cas, nous avons prévu dans le processus mémoire associative des programmes une taille suffisante pour éviter toute collision entre les noeuds. Par conséquent, la fonction de hachage locale n'est jamais appliquée et le temps de communication est diminué car les paquets requêtes vont atteindre leurs noeuds destinations en un minimum de temps. La figure 5.9 présente les accélérations réelles mesurées pour les 5 programmes testés avec un paramètre délai nul.

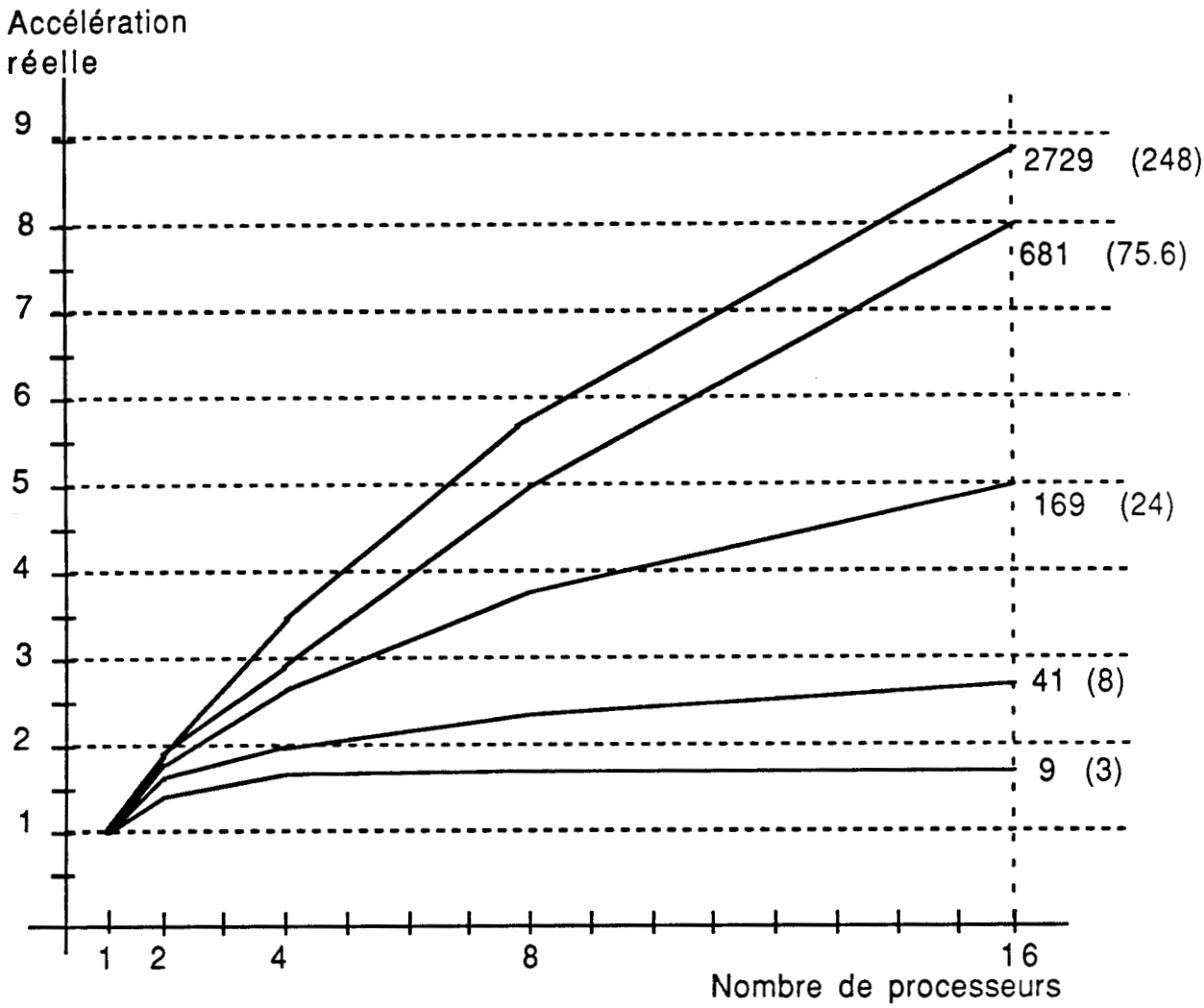
Ces 5 programmes correspondent à des arbres de dépendances de niveaux de profondeur différents. Nous avons indiqué aussi pour chaque programme testé quelle était son accélération théorique logicielle (le chiffre mis entre parenthèses).

Nous pouvons alors faire les constatations suivantes:

1- Lorsque la taille du programme est importante, le nombre moyen d'expressions pouvant être évaluées en parallèle devient important. De ce fait, si suffisamment de processeurs existent dans le réseau, l'accélération obtenue croit avec la taille (en nombre d'expressions) du programme à exécuter.

2- Quelque soit le programme exécuté il existe un seuil (au niveau du nombre de processeurs) au-delà duquel l'accélération réelle devient constante (ou presque).

A titre d'exemple pour le programme contenant 41 définitions ACC (41, 4) = 2, ACC(41, 8) = 2,8 , et ACC(41, 16) = 3. De ce fait l'ajout de 8 processeurs (le passage de 8 à 16 processeurs) n'a produit qu'une très faible augmentation (de 0,2). On peut dire ainsi que le seuil de l'accélération pour le programme comportant 41 définitions est atteint lorsque le nombre de processeurs est 8.



N.B : Le chiffre mis devant la courbe représente la taille du programme exécuté alors que celui entre parenthèses exprime l'accélération théorique logicielle

Fig 5.9 : Les accélérations obtenues pour différentes tailles de programmes

De plus lorsque N = 8 l'accélération théorique matérielle est égale à l'accélération théorique logicielle. De ce fait, si des processeurs supplémentaires (en nombre supérieur à 8) sont utilisés, le programme ne contient pas assez de parallélisme pour les occuper. Ceci provoque, par conséquent, une mauvaise exploitation des processeurs (efficacité réduite). En effet, comme on peut le voir sur le tableau 5.1, l'efficacité décroît très rapidement lorsque le nombre de processeurs est supérieur à 8.

Nombre de processeurs	1	2	4	8	16
Efficacité	1	0.85	0.55	0.47	0.26

Tab 5.1 : L'efficacité pour le programme contenant 41 expressions.

Pour cela il est intéressant d'utiliser des réseaux dont le nombre de processeurs est égale aux taux de parallélisme des programme à exécuter. Ceci permet d'avoir un rapport d'efficacité important et aussi une accélération réelle proche du maximum (lire conclusion).

3- Il faut remarquer aussi que les accélérations réelles obtenues sont en général inférieures aux accélérations théoriques logicielles.

Pour les programmes à 9 et 41 expressions par exemple, les accélérations réelles sont 2,6 et 2,8 (respectivement), alors que les accélérations théoriques logicielles sont égales à 3 et 8 (respectivement).

Ceci est dû à 2 raisons:

* Les opérations d'émission et de réception de messages imposent des temps d'attente.

* La méthode de répartition du programme choisie.

En effet, si nous tenons compte de la répartition du programme contenant 41 expressions (fig 5.4), nous constatons qu'il n'y pas 41 tâches pouvant être exécutées en parallèle, mais uniquement 21. Ce nombre correspond au nombre de boules dans le programme. Comme le nombre de niveau est toujours égal à 5, on peut dire que l'accélération théorique logicielle en tenant compte de la méthode de répartition n'est que de $\frac{21}{5}$ = 4. Le même calcul pour le programme contenant 9 expressions, donne une accélération théorique (en tenant compte de la méthode de répartition) égale à $\frac{5}{3}$ = 2,66 .

Remarque:

Pour avoir les seuils des programmes de 169, 681 et 2729 définitions, il faudrait faire des mesures avec un nombre de processeurs supérieur à 16.

Afin de donner une idée sur l'ordre de grandeur des temps d'exécution obtenus, nous présentons dans la table 5.2 les temps d'exécution pour le programmes à 2729 expressions. Notons que l'accélération théorique

logicielle pour ce programme est de $\frac{2729}{11} = 248$. De ce fait le temps d'exécution peut encore être réduit en augmentant le nombre de processeurs.

Nombre de processeurs	1	2	4	8	16
Temps d'exécution en milli-sec	6395	3570	2062	1240	730
Efficacité	1	0.9	0.77	0.64	0.54

Tab 5.2 : Temps d'exécution et efficacité pour le programme de 2729 exp avec pour un délai nul

La 2ieme série de mesures que nous avons réalisée (toujours en éliminant le phénomène de collision entre les noeuds) était d'augmenter le niveau du grain afin d'évaluer l'importance de ce dernier sur les performances de la machine.

La figure 5.10 donne les accélérations réelles pour le programme composé de 2729 expressions avec des paramètres délai de 0, 20, et 70 unités. Nous pouvons ainsi voir que l'augmentation du niveau de grain des opérations permet de réaliser des accélérations de plus en plus importantes.

Lorsque le niveau de grain des opérations est grand, la quantité de traitement local est importante. Dans ce cas le coût des communications devient très faible sur le temps d'exécution. Un délai de 70 unités permet, comme on peut voir, d'avoir une accélération de 11,5 au lieu de 8,77 pour un délai nul. Comme expliqué dans l'introduction de ce chapitre, un facteur de délai égal à une unité correspond à une attente active de 64 micro-secondes.

L'addition de 2 variables en mémoire par un transputer, ayant un cycle de 67 nano-secondes, se fait en 2,04 micro-secondes. Par conséquent un délai égale à 70 unités (4,48 milli-secondes) est équivalent à un processus OCCAM réalisant 2191 opérations d'addition sur deux variables en mémoire. Ce qui représente un niveau de grain assez élevé.

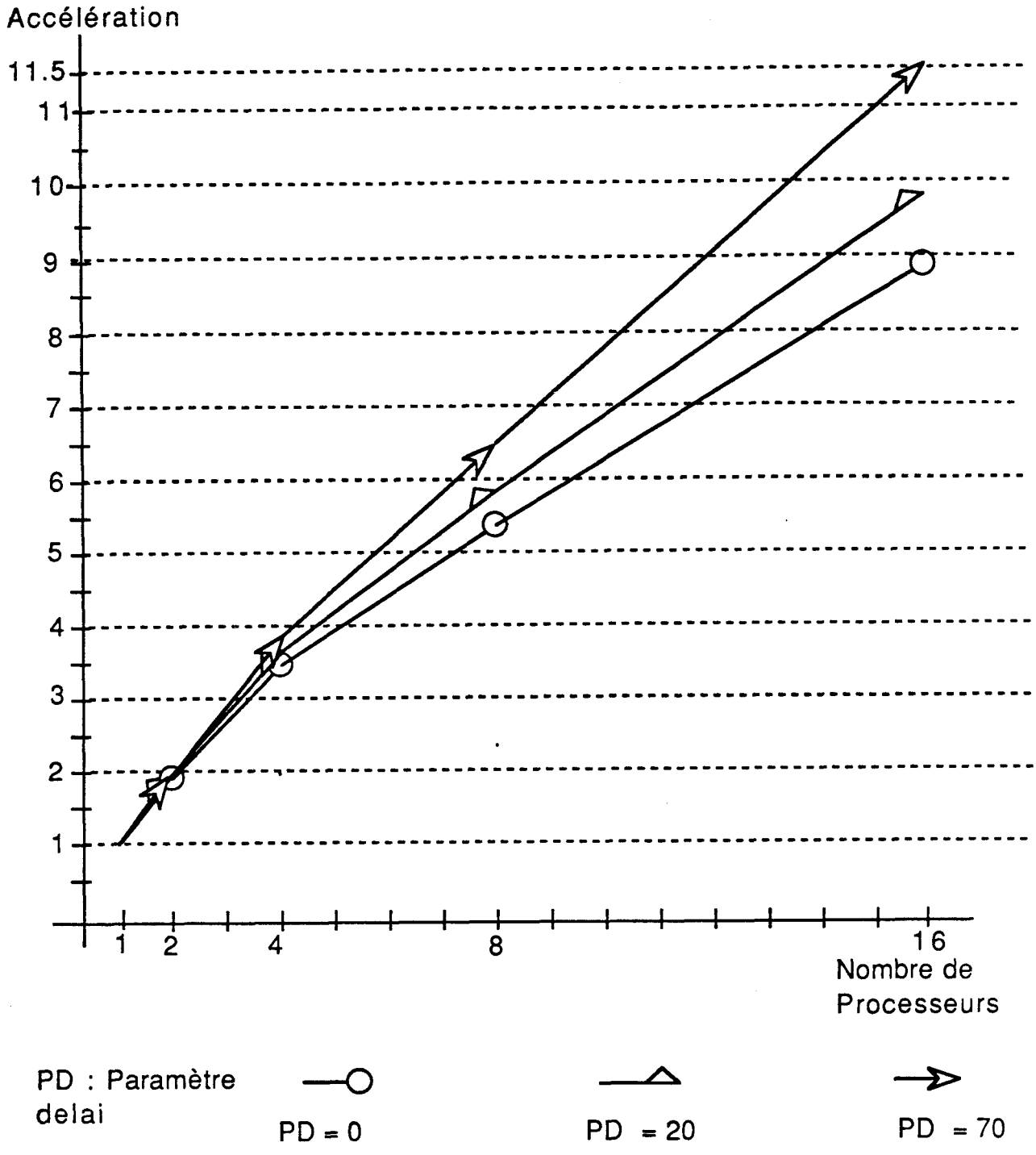


Fig 5.10 : Influence du niveau de grain sur l'accélération

De plus augmenter le niveau de grain des tâches parallèles n'est pas la solution qui permet d'avoir dans tous les cas les meilleurs performances en temps d'exécution. En effet, un niveau de grain important peut

diminuer le degré de parallélisme du programme à exécuter. Des tâches élémentaires parallèles peuvent exister dans une opération de niveau de grain élevé, et seront, par conséquent, exécutées de manière séquentielle. Ceci n'autorise pas la machine à utiliser toutes ces possibilités de traitement. Il faut noter aussi, qu'une répartition équilibrée de la charge (statique ou dynamique) est difficile à réaliser pour des programmes ayant un niveau de grain élevé. Les écarts entre les charges peuvent être très important dans ce cas.

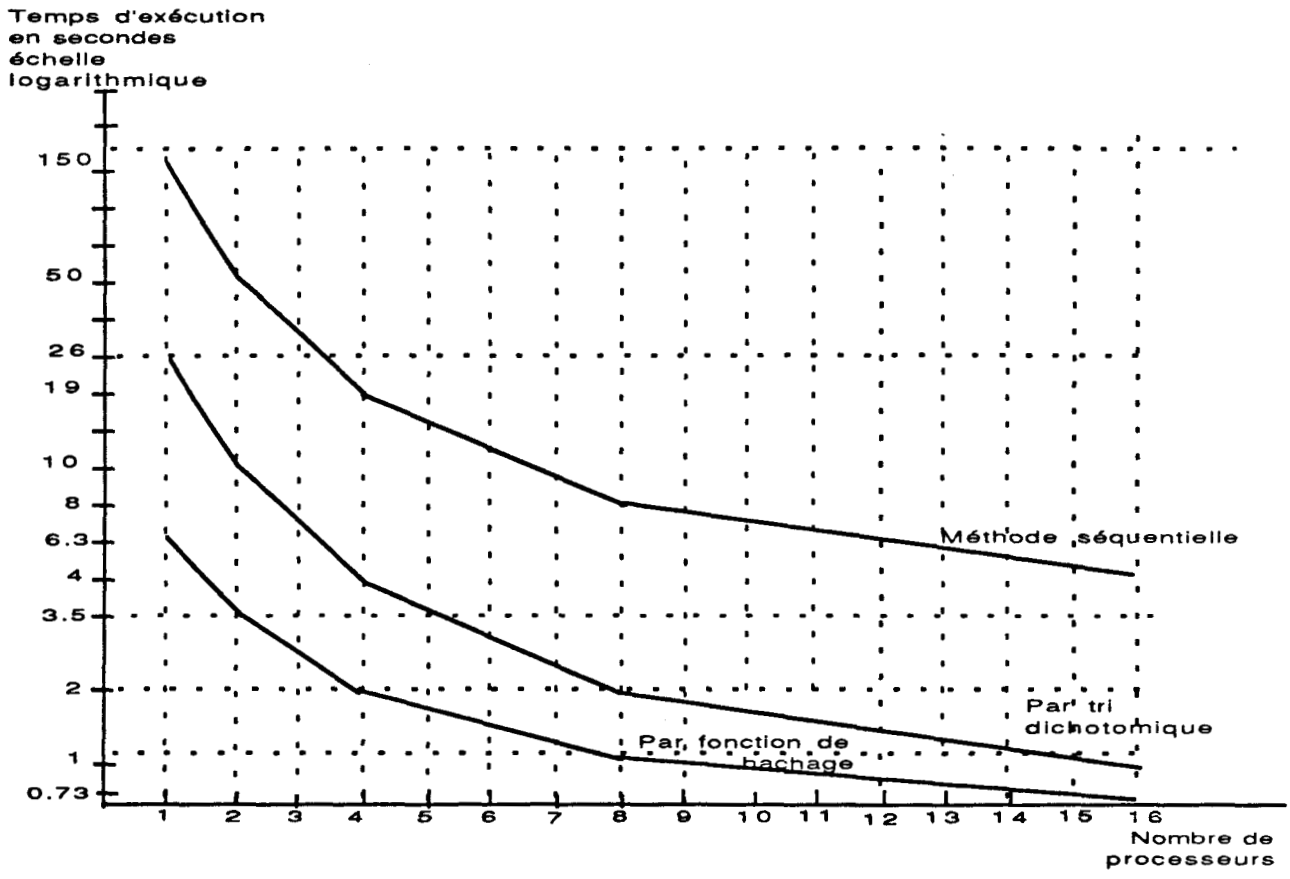


Fig 5.11 : Les temps d'exécution pour les
3 implémentations de la mémoire associative

Pour conclure ce paragraphe, nous donnons dans la figure 5.11 les courbes représentant les temps d'exécution pour le programme de 2729 expressions en utilisant les 3 méthodes pour simuler les mémoires associatives des programmes et des paquets.

Comme on pouvait s'attendre, c'est la méthode utilisant une fonction de hachage qui offre le plus petit temps d'exécution.

Concernant cette méthode, les essais réalisés ont montré que le temps de calcul de la fonction de hachage des mémoires associatives est de 23 micro-secondes, et qu'elle est calculée en moyenne 4 fois pour toute expression du programme (2 fois en moyenne dans la mémoire des programmes et 2 fois dans la mémoire des paquets).

L'overhead dû à l'utilisation de cette méthode est donc proche de 100 micro-secondes. Dans le tableau 5.3 nous donnons toujours pour le programme à 2729 expressions le nombre moyen d'expressions dans un processeur ainsi que le nombre moyen de collisions par processeurs dans la mémoires associatives des programmes (collisions intra-noeud). Ces collisions sont, donc, dûes à la simulation des mémoires associatives par des processus OCCAM.

Nombre de processeurs	1	2	4	8	16
Nombre moy d'expressions par processeur	2729	1367	682	341	170
Nombre moy de collisions par processeurs	1314	165	29	7	0

Tab 5.3: Nombre moyen de collisions mémoire
dans une table de 2729 entrées.

Comme on peut le voir le nombre de collisions produit par la fonction de hachage de la mémoire associative n'est pas très important, sauf lorsque le nombre d'expressions est proche de la taille de la mémoire elle même.

V.3.2 Les temps d'exécution avec collisions

Afin de voir l'influence des collisions sur les performances du réseau, nous avons réduit la taille de la mémoire des programmes pour provoquer des collisions dès que le nombre d'expressions stockées dans la mémoire a atteint une certaine limite. Cette limite T_m peut être fixé au choix de l'utilisateur à condition que $N * T_m >= T_{pr}$, où N est le nombre de processeurs et T_{pr} est la taille du programme. Le programme est représenté par un certain nombre d'expressions, et chaque expression est

représentée par une entrée dans le tableau 1 de la mémoire associative. Ainsi pour provoquer des collisions au niveau d'un noeud, il suffit, par exemple, d'allouer au tableau 1 de la mémoire associative des programmes un nombre d'entrées inférieur au nombre d'expressions affectées à ce noeud.

Nombre de processeurs adressés par la FHG	1	2	4	8	16
nombre total de processeurs utilisés	5	8	4	8	16
nombre de processeurs contenant uniquement des expression en collision	4	6	0	0	0
nombre moyen d'expressions en colli par noeud	30,7	12,8	0	0	0
temps d'exécution en milli-seconde	307	233	144	101	74
gain en temps par la FHG	1	1,31	2,13	3,07	4,14
temps d'exécution sans collisions	376 **	229 *	144	101	74

Tab 5.4: Résultats d'exécution du programme avec 169 expressions

Nous présentons dans le tableau 5.4 et 5.5 les résultats d'exécution de 2 programmes (169 et 681 expressions). Lors de ces essais 46 entrées ont été allouées au tableau 1 du processus mémoire associative des programmes. En conséquence, un noeud peut comprendre au maximum 46 expressions. Dans tous les essais que nous avons réalisés, c'est un hypercube d'ordre 4 qui est utilisé (16 processeurs).

Dans la première ligne nous avons reporté le nombre de processeurs sur lequel la fonction de hachage globale répartit le programme. Si une expression est stockée sur un noeud autre que celui calculé par la

fonction de hachage globale, on dira que l'expression est en collision. De cette manière, si la fonction de hachage globale répartit le programme sur un nombre réduit de processeurs (sur 1 ou 2 processeurs) la quantité d'expressions en collision peut être importante si le programme contient un grand nombre d'expressions.

nombre de processeurs adressés par la FHG.	1	2	4	8	16
nombre total processeurs utilisés	16	16	16	16	16
nombre de processeurs contenant uniquement des expressions collision	15	14	12	8	0
nombre moyen d'expressions en collision par noeud	42	42	42	39	0
temps d'exécution en milli-seconde	1076	954	719	460	191
gain en temps par la FHG	1	1.12	1.49	2.33	5.60
temps d'exécution sans collision	1542 **	879 *	512 *	314 *	191

Tab 5.5: Résultats d'exécution du programme avec 681 expressions

Dans ce cas on dira que la fonction de hachage globale est mauvaise. Si par contre le nombre de processeurs sur lequel la fonction de hachage globale répartit le programme est grand (8 ou 16 processeurs), il y aura très peu (ou pas) d'expressions en collision, et on dira que la fonction de hachage globale est bonne.

La deuxième ligne des 2 tableaux précédents donne le nombre de processeurs qui contiennent au moins une expression. Ce nombre peut être divisé en 3 parties:

- Des processeurs qui ne possèdent aucune expressions en collision.

- Des processeurs qui détiennent uniquement des expressions en collision.

- Des processeurs qui possèdent des expressions dont certaines sont en collision et d'autres ne le sont pas.

Dans la troisième ligne nous avons indiqué le nombre de processeurs qui contiennent uniquement des expressions en collision. On peut remarquer que le nombre total de processeurs utilisés (ligne 2) est égale au nombre de processeurs utilisés par la fonction de hachage globale (ligne 1) plus le nombre de processeurs qui contiennent uniquement des expressions en collision (la ligne 3).

Le nombre moyen d'expressions dans les processeurs ne contenant que des expressions en collisions est donné dans la ligne 4. Comme on peut le noter, plus la fonction de hachage est bonne, moins grand est le nombre moyen d'expressions en collision par noeud.

Dans la ligne 5 et 6 nous avons donné respectivement le temps d'exécution en milli-secondes, et le gain relatif en temps d'exécution des programmes testés par l'amélioration de la fonction de hachage. Nous remarquons que, lorsque la fonction de hachage globale est bonne, le temps d'exécution est réduit et le gain relatif est important. Ceci se produit même lorsque le nombre total de processeurs impliqués dans le traitement diminue. A titre d'exemple, nous pouvons comparer la colonne 2 du tableau 5.4 où le nombre de processeurs adressés par la FHG est 2 mais le nombre total de processeurs est de 8 (dont 6 contiennent uniquement des expressions en collision) et la colonne 3 du même tableau où le nombre total de processeurs est de 4. Malgré cette diminution en nombre de processeurs, le gain a augmenté (de 1,31 il est passé à 2,13).

Dans la dernière ligne nous avons indiqué le temps d'exécution obtenu dans le cas ou il n'y pas de collisions (mémoire associative assez grande pour contenir tout le programme). Le nombre de processeurs effectivement utilisés pour réaliser les calculs est dans ce cas, celui présentés dans la ligne 1.

Ceci montre que de façon générale, l'existence des paquets en collision provoque un allongement du temps d'exécution (colonnes marquées par une *), et montre aussi l'importance de la fonction de hachage globale.

V.4 CONCLUSIONS

Les résultats présentés dans ce chapitre nous ont permit de tirer un certain nombre de conclusions concernant le fonctionnement de l'émulateur de N-ARCH. Ces conclusions sont:

A) Afin de réduire la quantité de charges supplémentaires que le noyau doit réaliser, il est nécessaire d'intégrer au niveau du matériel un certain nombre d'unités clés. Nous pouvons proposer pour cette opération d'intégration les processus suivants : les contrôleurs des buffers d'entrée et de sortie, les mémoires associatives des programmes et des paquets et les deux fonctions de hachage.

De l'autre côté on peut aussi ajouter que les programmes exécutés par le noyau N-ARCH sont interprétés. Une exécution du programme après compilation du code permettra sûrement d'avoir des temps plus faibles. Cette approche a été, par ailleurs utilisée depuis bien longtemps pour construire des machines séquentielles exécutant des langages déclaratifs après compilation du code comme la machine SECD [Hend80], ou récemment pour réaliser des machines parallèles comme la machine MARS [Cast86] ou GRIP [Peyt87] par utilisation des combinateurs (ou supercombinateurs).

L'utilisation d'un émulateur comme outil de tests ne permet, malheureusement, pas l'implémentation de cette méthode.

B) Les contrôleurs de liens ont un effet très important sur les performances de la machine. Il est donc indispensable de prévoir au niveau du noeud N-ARCH des unités d'entrée-sortie autonomes qui permettront de décharger l'unité de contrôle des tâches liées à la gestion des communications et garantiront un plus grand degré de parallélisme au niveau interne. Ceci est d'autant plus nécessaire que le niveau de grain des opérations est faible, car dans ce cas les programmes contiennent un grand nombre d'opérations d'entrée-sortie (nombre élevé de paquets à émettre ou à recevoir).

C) La répartition du programme par utilisation du phénomène de collision (c'est à dire par fonction de hachage locale) est moins équilibrée que celle obtenue par une bonne fonction de hachage globale. Les temps d'exécution obtenus par utilisation d'une bonne fonction de hachage globale (sans collision) sont nettement inférieurs par rapport à ceux

obtenus quand le nombre de collisions est élevé. Il y a 2 raisons essentielles pour cela:

1- La méthode de répartition par effet de collision ne permet pas d'exploiter tout le parallélisme dans le traitement des différentes branches de l'arbre de dépendance. Contrairement à la fonction de hachage globale qui répartit les boules de l'arbre sur tous les processeurs du réseau, la fonction de hachage elle, stocke toutes les expressions en collision dans les voisins (uniquement) jusqu'à occupation de toute la mémoire des voisins. Ceci ne permet, par conséquent, qu'un faible parallélisme dans le traitement des boules (tab 5.4 et 5.5).

2- Le temps de communication d'un paquet requête en collision est plus grand que celui d'un paquet requête en routage (le calcul des temps se fait dans les mêmes conditions: mêmes chemins, mêmes charges de noeuds, ...etc). En effet la transmission d'un paquet requête par un noeud à son voisin (fig 5.12) nécessite le passage par le buffer d'entrée, l'accès à la mémoire associative des programmes et enfin l'envoi du paquet par l'unité de contrôle vers le buffer de sortie.

Dans la version où les mémoires associatives sont simulées par utilisation d'une fonction de hachage, les tests ont montré que le temps séparant la réception d'un paquet requête en collision par le processus d'entrée et son émission vers le voisin, est égale à 500 micro-secondes en ayant les buffers d'entrée et de sortie vides, l'unité de contrôle au repos, et pas de collisions au niveau de la mémoire des programmes (1 seul accès). Pour un paquet requête en routage ce temps n'est que de 45 micro-secondes.

Néanmoins, il est parfois très intéressant de provoquer des collisions dans le réseau afin de mieux répartir le programme. Ceci est le cas lorsque:

- La fonction de hachage globale est très mauvaise. (les colonnes marquée par ** dans les tableaux 5.3 et 5.4).
- Le temps de traitement est important devant le temps de communication. Ceci peut se produire dans 2 cas distincts : soit lorsque le niveau de grain des opérations est élevé, ou (et) lorsque le temps de communication des paquets requêtes en collision est faible (fonction de hachage locale par hardware, mémoire associative rapide, contrôleurs d'E/S autonomes).

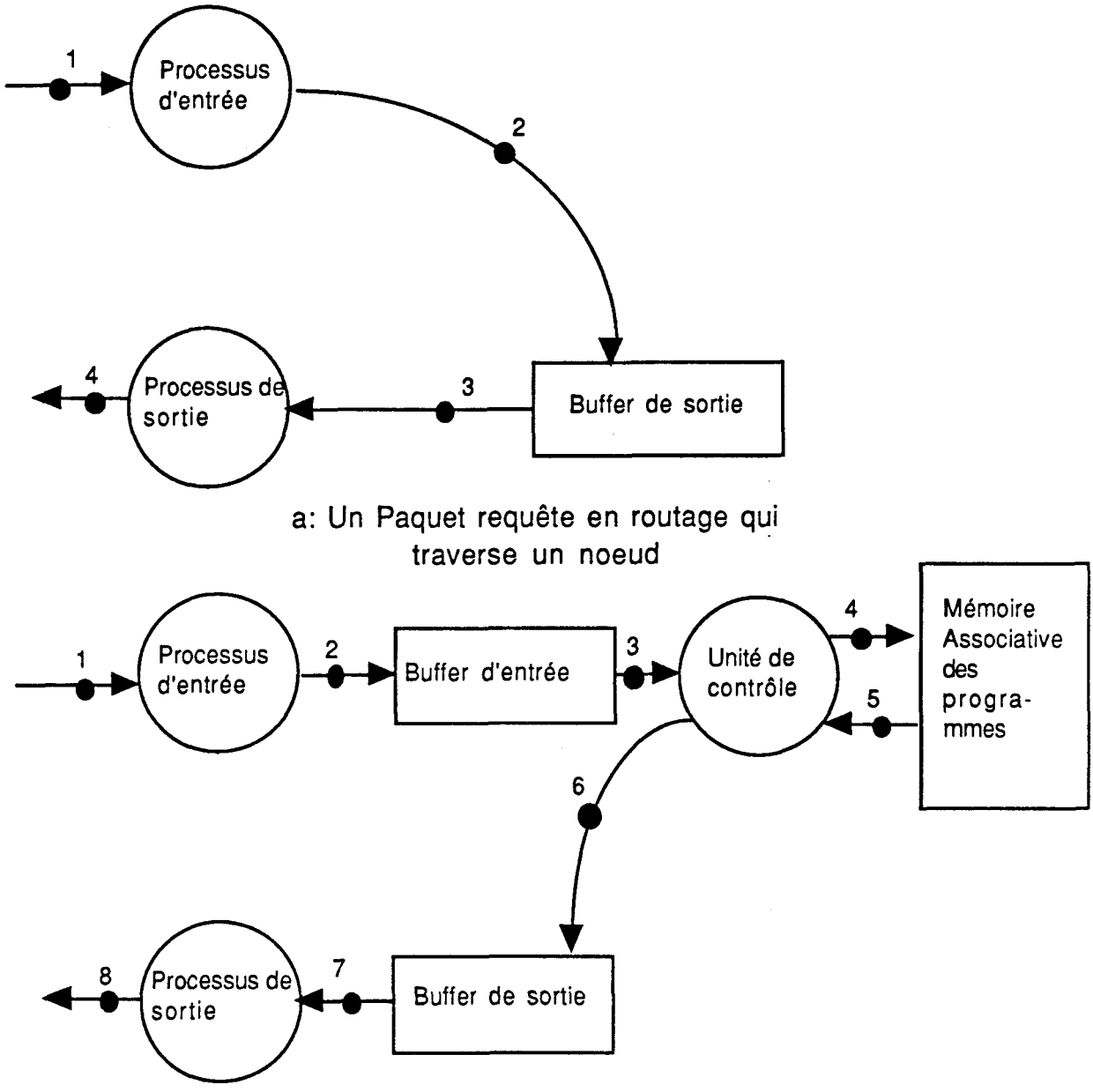


Fig 5.12 : Un noeud requête qui traverse un noeud

La méthode de répartition des expressions par taux de remplissage, que nous avons présenté dans le chapitre IV peut être utilisé lorsque l'une de 2 conditions est réalisées.

Pour cela, il est nécessaire de réaliser le calcul de la fonction de hachage locale par matériel et d'intégrer au niveau du noeud une

mémoire associative réelle. L'intégration des contrôleurs des buffers d'entrée et de sortie permettra aussi de décharger la CPU d'une charge supplémentaire. D'autant plus que les algorithmes de contrôle réalisés dans chacun des 2 buffers sont très simples (algorithmes FIFO) et peuvent donc facilement être réalisés par matériel.

3- Quelque soit le programme testé il y a toujours un seuil en nombre de processeurs, au delà duquel l'accélération devient fixe. Dans tous les cas, ce seuil ne peut jamais dépasser l'accélération théorique logicielle. Il faut noter, toute fois, que l'accélération réelle obtenue pour un nombre de processeurs égal à l'accélération théorique logicielle est très proche de ce seuil.

Néanmoins, pour les programmes déclaratifs qui sont en général récursifs, le nombre de tâches parallèles, et par conséquent l'accélération théorique moyenne, dépend de la donnée sur la quelle s'exécute le programme [Germ89].

Références citées dans le chapitre :

[Eage89], [Germ89], [Gonc89], [Hend80], [Niar89], [Peyt87].

CHAPITRE VI
EXTENSION DU NOYAU A L'EVALUATION DES FONCTIONS

VI.1 Introduction

Nous décrivons dans ce chapitre, l'extension que nous avons apportée au noyau tel qu'il a été défini jusqu'à présent. Il s'agit de la prise en compte des fonctions au niveau du langage de simulation.

La possibilité d'évaluer des fonctions permet à l'émulateur d'exécuter des programmes fonctionnels réels. Plusieurs autres possibilités peuvent être également intégrées dans le noyau pour augmenter sa puissance et enrichir ces possibilités (tel que le traitement des données structurées de différents types, la manipulation de fonctions de haut niveau, la récupération dynamique d'espace libre, ...etc).

Partant de la définition du noyau présenté dans le chapitre IV, nous avons imaginé un modèle de traitement de fonctions. Ce modèle repose sur l'utilisation de la structure d'un noeud N-ARCH et sa décomposition en 8 unités fonctionnelles, comme ceci à été expliqué dans le paragraphe IV.2 .

Dans cette extension le modèle d'évaluation est toujours celui de la réduction de graphes. Une expression sera évaluée que lorsque ceci est nécessaire à la poursuite d'une évaluation (lazy evaluation). Pour permettre cela, le passage des paramètres des fonctions doit se faire par références. Dans le modèle que nous proposons une fonction est représentée sous forme de lambda-expressions (sans utilisation de combinateurs) et les différents appels de la fonction se font par duplication du corps de celle-ci.

Dans une première étape, on supposera que les opérateurs élémentaires (comme les opérateurs arithmétiques, le IF, et les opérateurs sur les listes: Car, Cdr, Cons) sont tous stricts. On montrera ensuite comment on peut évaluer les expressions utilisant des opérateurs élémentaires de manière paresseuse.

Dans le modèle de traitement des fonctions que nous proposons, il n'y pas de notion de variables globales. Par conséquent, toutes les variables référencées dans le corps de la fonction sont locales. Cette restriction ne pose pas un grand problème car pour exécuter une fonction qui utilise

une variable globale, il suffit d'introduire dans la définition de cette fonction un nouveau paramètre d'appel lié à la variable globale. A titre d'exemple, reprenons la fonction qui calcule la factorielle d'un entier par la méthode "diviser pour conquérir" (page 17). Les variables "min" et "max" sont des variables globales pour la fonction "mid". Pour éliminer ces variables, il suffit d'écrire le programme de la façon suivante:

```
Par_Fact n = Dec 1 n
Dec min max =   If  (min = max)  Then  min
                    Else  (Dec  min  (Mid min max) )   *
                          (Dec ( (Mid min max)+1)   max)
Mid x y  =  (x + y) IntDiv 2
```

VI.2 Le graphe d'exécution des fonctions

Dans une machine exécutant un langage fonctionnel, on peut distinguer 4 type d'expressions :

1) Les constantes.

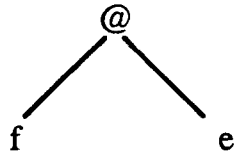
2) Les variables : comme les arguments d'une fonction.

3) Les application d'une fonction à des arguments. Une application est notée : @ <expression> {<expression} , comme @ f 2 3.

4) Les lambda-expressions; notée λ{<variable>}.<expression>.

Comme exemple de lambda-expression: λx. x+1

De façon générale l'application d'une fonction "f" à un argument "e" peut être représentée par l'arbre suivant :



Si la fonction f utilise plusieurs arguments, nous pouvons, par exemple, prendre comme convention de représenter l'application f sur plusieurs arguments par une suite de noeuds @ (fig 6.1) et (fig 6.2).

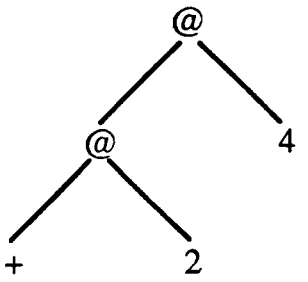


fig 6.1 : première représentation de l'expression
+ 2 4 = (+ 2) 4

On considère dans ce cas que chaque opérateur est unaire. De cette façon, une application d'une fonction f à n arguments est transformée en une série de n applications de fonctions à 1 argument. Cette opération de transformation de la fonction f est appelée *currification*. La figure 6.2 montre un exemple de représentation d'une fonction utilisant cette méthode.

f = λx. x * 2
f 4 = (λx. x * 2) 4

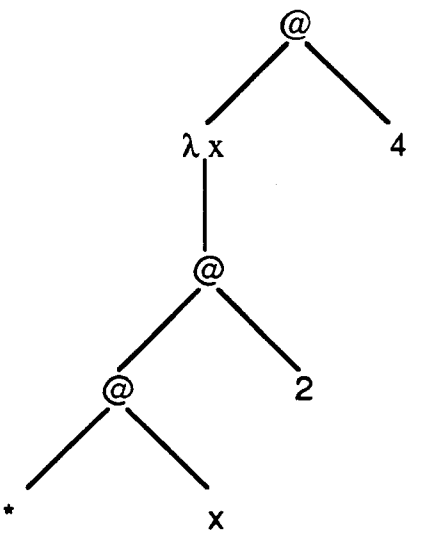


fig 6.2 : le graphe représentant l'application d'une fonction

Les cellules mémoires représentant le graphe ont, dans ce cas, des tailles fixes. Cette représentation permet d'avoir une gestion simple de l'espace mémoire libre.

Une autre alternative serait de donner la possibilité aux noeuds @ d'avoir plusieurs arcs fils (fig 6.3).

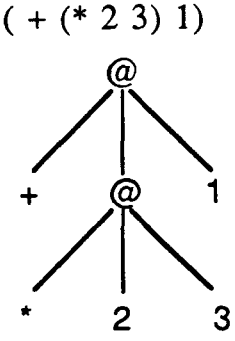


fig 6.3 : Deuxième représentation d'une expression

Les cellules mémoires doivent avoir dans ce cas des longueurs variables. Si les noeuds du graphe d'exécution sont répartis sur un réseau de processeurs, cette méthode permet d'avoir une exécution rapide des opérateurs car il y a moins d'opération d'émission/réception de messages à faire. En effet dans cette méthode, à partir d'un noeud @ on peut accéder directement à tous les paramètres nécessaires à l'exécution de l'opérateur. C'est cette deuxième méthode qui est utilisée dans le modèle que nous proposons.

VI.3 La représentation mémoire du graphe

Le graphe d'exécution d'une fonction est représenté en mémoire par un ensemble de cellules. Chaque cellule est équivalente à un noeud du graphe et elle est composée du nom de la cellule, une marque indiquant si la cellule correspond à un noeud application (@), une lambda-expression (λ), ou une constante (N), et un ou plusieurs champs arguments.

Un champ argument peut être :

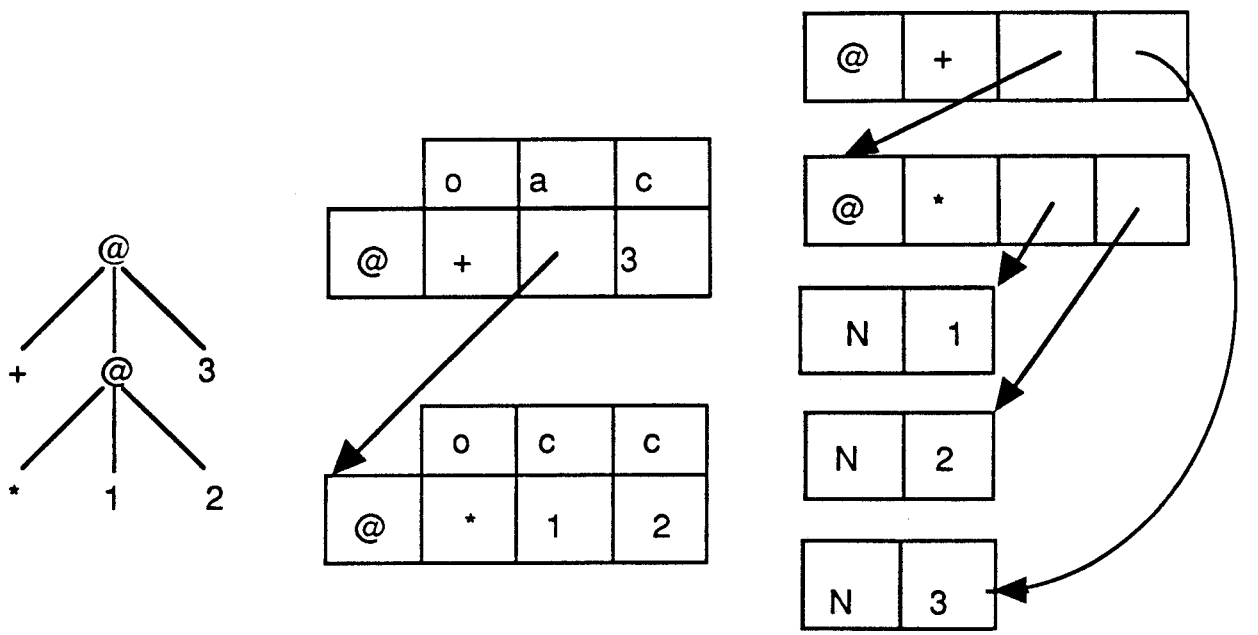
- Le nom d'une fonction à appliquer.
- Le nom d'une expression. Dans la suite de ce chapitre on utilisera aussi le terme "référence" pour désigner le nom d'une expression.
- Le nom d'une variable formelle dans une lambda-expression ou dans le corps d'une fonction.
- Le code d'un opérateur élémentaire.

- La valeur d'un paramètre réel (c'est à dire une constante).

Pour distinguer chacun de ces cas, à chaque champ argument de la cellule est associé une marque appelée "marque de l'argument". D'après ce qui a été dit précédemment, on peut distinguer 5 types de marque d'arguments:

- Fonction (f)
- Nom d'expression, c'est à dire une référence (a)
- Variable (v)
- Opérateur élémentaire (o)
- Constante (c).

La représentation des fonctions que nous avons adoptée permet de minimiser l'utilisation de références et offre ainsi un gain d'espace mémoire. Dans cette représentation, si les arguments d'une fonction ou d'un opérateur élémentaire sont du type constante ou variable, ils sont représentés au niveau de la même cellule que la fonction (fig 6.4 a).



a : représentation sans références (unboxed)

b : représentation avec des références (boxed)

fig 6.4 : la représentation du graphe en mémoire

Remarquons qu'il est aussi possible d'utiliser une autre représentation, où tous les arguments d'un noeud @ sont du type références (fig 6.4 b)*.

* Cette dernière représentation est appelée en anglais "boxed representation", alors que la première est appelée "unboxed representation".

Comme on peut le noter, la première représentation (à gauche) utilise des marques d'argument et nécessite pour l'exemple montré 2 cellules et 6 marques d'argument. L'autre représentation n'utilise pas de marque d'argument (car tous les arguments sont des références) mais nécessite 5 cellules. En général, nous pouvons dire que les marques d'argument sont implémentées par une zone mémoire de taille réduite par rapport à la taille d'une cellule. Nous pouvons dire, par conséquent, que la méthode de représentation choisie permet un gain d'espace.

En résumé, nous avons les représentations mémoires suivantes:

A = @<fonc> {<arg1>...<argn>}

			f	a/c/v	...	a/c/v
A	@	n+1	fonc	arg1	..	argn

B = @<opér> {<arg1>...<argn>}

			o	a/c/v	...	a/c/v
B	@	n+1	opér	arg1	.	argn

C = λ{<var1> <var2>..<varn>}.corps

			v	v		a
C	λ	n+1	var1	var2	..	corps

Dans certains cas (voir plus loin), il est nécessaire de représenter dans des cellules indépendantes les paramètres réels d'une fonction. A ce moment, la cellule a la forme suivante.

D = <const>

		c
D	N	const

VI.4 Les types de paquets utilisés

Afin de supporter l'évaluation des fonctions dans l'émulateur, il est nécessaire d'ajouter d'autres types de paquets, et de modifier le format de certains types de paquet déjà existant.

Nous allons voir dans ce paragraphe l'ensemble des paquets qui sont nécessaires à l'évaluation des fonctions.

VI.4.1 les paquets d'initialisation

Ce paquet réalise la même fonction que précédemment, à savoir le chargement du programme dans les différentes "unités mémoires associatives des programmes". Les champs existant dans ce type de paquet ont changé. Ces champs sont :

- Un champ type, ayant comme la valeur : initialisation en routage ou initialisation en collision.
- Le nom de l'expression à mémoriser.
- La marque de l'expression spécifiant s'il s'agit d'une application, d'une lambda-expression ou d'une constante.
- Le numéro du noeud destination calculé par l'application de la fonction de hachage globale sur le nom de l'expression.

Les champs qui suivent dépendent du type de l'expression :

* Si l'expression correspond à un noeud @, nous aurons les champs suivants :
  - Le nombre de champs arguments.
  - La chaîne d'arguments. Chaque argument est accompagné d'une marque (un octet) spécifiant le type de l'argument correspondant.
* Si l'expression correspond à un noeud lambda, nous trouvons les champs suivants :
  - le nombre de champs arguments. Ce nombre correspond au nombre de variables liées dans la lambda-expression plus 1.
  - La chaîne d'arguments. Là aussi chaque argument est accompagné d'une marque. Tous les arguments d'une lambda-expression ont la marque "v" sauf le dernier qui correspond au corps de la fonction et qui peut avoir soit la marque "a", comme dans l'expression $\lambda x.\ x+1$, la marque c, comme dans l'expression $\lambda x.\ 2$, ou bien la marque v, comme dans l'expression $\lambda x.\ x$.
* Si l'expression correspond finalement à un noeud constante, il y a un seul champ argument qui contient la valeur de la constante (la variable réelle) avec la marque "c".

Remarquons ici, que pour une expression du type @, le premier argument a soit la marque "f" ou la marque "o", et que pour une expression du type λ , cette marque est toujours "v", et qu'en fin pour une expression du type "N", elle est toujours égale à "c".

VI.4.2 Les paquets requête et les paquets réponse

Ces paquets ont les mêmes formats que ceux présentés dans le paragraphe IV.3.

IV.4.3 les paquets requête-fonction

Ce type de paquet est généré par un noeud @ lorsque la marque du premier argument de l'expression est égale à "f". Par conséquent, un paquet requête-fonction est toujours destiné à un noeud "λ". Un paquet requête-fonction contient les champs suivants :

- Le type du paquet, qui est requête-fonction (en routage ou en collision).
- Le nom de la fonction à appliquer.
- Le numéro du noeud destination fourni par la fonction de hachage globale.
- Le nom de l'expression réceptrice.
- Le nombre d'arguments sur lesquels s'appliquera la fonction.
- La liste des références vers les paramètres réels ou directement les valeurs des paramètres réels. Chaque élément est, de ce fait, accompagné par une marque indiquant s'il s'agit d'une référence (marque égale à "a") ou d'une constante (marque égale à "c"). Dans la suite de ce chapitre on notera cette liste {(e1)m1, (e2)m2,..(en)mn}, où les ei sont les éléments de la liste et les mi sont les marques associées.
- Le nombre d'éléments dans la liste des liens. Cette liste représente l'environnement dans le quel seront évalués les paramètres de la fonction.
- La liste des liens représentée par des doublets (paramètre formel/paramètre réel). Le paramètre réel peut soit être représenté par une référence vers l'expression qui le définit soit être exprimé directement sous forme d'une constante. Pour cela, chaque doublet contient une marque, égale soit à "a" ou bien à "c", indiquant si le

deuxième élément du doublet est, respectivement, une référence ou une contante. Dans la suite, nous noterons cette liste par {(f1/r1)m1, (f2/r2)m3, ...}, où fi représente le nom du paramètre formel, ri la référence ou la valeur du paramètre réel et mi la marque associée au couple.

IV.4.4 Les paquets requête-copie

Comme il a été précédemment indiqué, les différentes évaluations d'une même fonction se font par duplication du corps de celle-ci. De ce fait, il est nécessaire de définir un paquet afin de demander la copie et l'évaluation d'une fonction.

A cet effet un paquet requête-copie est utilisé afin de demander une copie de l'expression destination et de commencer son évaluation. Cette opération est réalisée récursivement pour toutes les expressions du corps de l'expression. Un paquet requête-copie est généré soit durant l'évaluation d'un noeud λ-expression qui a reçu un paquet requête-fonction, ou bien d'un noeud application (@) qui a reçu un paquet requête-copie.

Les champs existants dans ce type de paquet sont les suivants :

- Le type qui est égal à requête-copie (en routage ou en collision).
- Le nom de l'expression destination à copier et à évaluer.
- Le numéro du noeud destination.
- Le nom de l'expression qui a généré le paquet requête-copie.
- Le numéro du noeud où se trouve cette expression.
- Le nombre de paramètres utilisés pour réaliser la copie de l'expression. Ce nombre de paramètres correspond soit au nombre de variables liées trouvé dans la λ-expression si le paquet a été généré par un noeud λ , ou bien au nombre de paramètres qui se trouvait dans le paquet requête-copie si le paquet a été généré par un noeud @.
- La liste des doublets (paramètre formel/paramètre réel). Le paramètre réel peut soit être représenté par une référence vers l'expression qui le définit soit être exprimé directement sous forme d'une constante. Pour cela, chaque doublet contient une marque, égale soit à "a" ou bien à "c", indiquant si le deuxième élément du doublet est, respectivement, une référence ou une contante. Cette liste est notée par {(f1/r1)m1, (f2/r2)m3, ...}, où fi représente le nom du paramètre formel,

ri la référence ou la valeur du paramètre réel et mi la marque associée au couple.

IV.5 Le traitement des paquets par l'émulateur

Afin de détailler le mode d'évaluation des fonctions par l'émulateur d'un noeud N-ARCH, nous allons décrire dans ce paragraphe quelles sont les opérations que devra réaliser le processus de contrôle. Les autres processus de l'émulateur gardent le même fonctionnement que précédemment. Si, toutefois, des opérations supplémentaires sont à réaliser, nous l'indiquerons au moment nécessaire.

IV.5.1 Sur réception d'un paquet initialisation

Lorsqu'un paquet initialisation est reçu par l'unité de contrôle du noeud N-ARCH, une demande d'écriture est envoyée vers la mémoire des programmes. Comme auparavant, s'il existe suffisamment d'espace dans cette mémoire, l'expression contenue dans le paquet est mémorisée. Sinon, il y a collision, et la demande de mémorisation est reportée sur un noeud voisin (par hachage local).

IV.5 2 Sur réception d'un paquet requête

Lorsque l'unité de contrôle reçoit ce type de paquet, une demande de lecture est envoyée vers la mémoire associative des programmes. Si l'expression recherchée ne s'y trouve pas, on se trouve dans une situation de collision, sinon l'unité de contrôle reçoit l'expression demandée. Deux cas peuvent se présenter, suivant le type de l'expression:

1- Si la marque de l'expression est N, l'expression est donc réduite, un paquet réponse est alors construit puis envoyé vers le buffer de sortie.

2- Si la marque de l'expression recherchée est par contre @ et si cette expression n'a pas été demandée, il y a test de la marque du premier argument de l'expression. Deux cas peuvent alors se présenter :

- Si cette marque est égale à f, il s'agit d'un noeud de type "fonction". Dans ce cas un paquet requête-fonction est envoyé vers la définition de la fonction (fig 6.5 a). Dans ce paquet nous trouvons en particulier, le nombre de paramètres utilisés par la fonction, ainsi que la liste des

paramètres réels (ou des références vers ces paramètres réels) avec les marques associées. Comme l'environnement d'évaluation est vide, la liste des liens ne contient aucun élément.

- Si la marque du premier argument de l'expression est "o", il s'agit d'une application d'un opérateur élémentaire. Dans ce cas, si l'expression est à l'état repos, un paquet requête est émis pour chacun des paramètres de l'opérateur (fig 6.5 b).

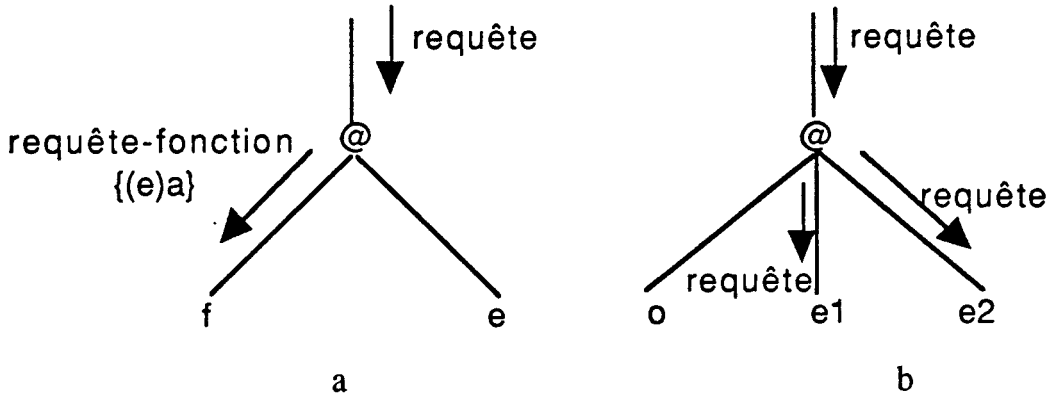


fig 6.5 : Interprétation d'un paquet "requête"

Dans tous les cas, un paquet réponse en attente est formé et déposé dans la mémoire associative des paquets.

VI.5.3 Sur réception d'un paquet requête-fonction

Ce paquet est toujours reçu par une expression qui correspond à un noeud "λ". Le traitement de ce paquet commence par la recherche de la fonction dont le nom est reçu dans le paquet. Si la définition de cette fonction (c'est à dire la λ-expression) est trouvée dans la mémoire associative des programmes, un paquet requête-copie est généré, puis envoyé vers le noeud qui contient le corps de la fonction. Le paquet requête-copie envoyé contient, comme expliqué précédemment, la liste des liens (paramètres formels/paramètres réels). Cette liste est construite en utilisant les paramètres réels (ou les références vers ces paramètres réels) reçus dans le paquet requête-fonction, et les paramètres formels trouvés dans la λ–expression (fig 6.6).

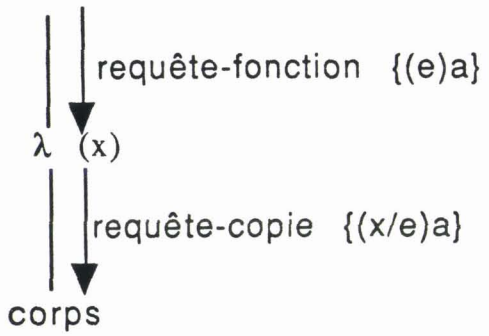


fig 6.6 : Interprétation d'une λ-expression

Si le paquet requête-fonction contient une liste de liens (environnement d'évaluation non vide), cette liste est ajoutée à la liste des liens précédente.

En plus de cette liste, le paquet requête copie généré contient le nom de l'expression émettrice. Afin de minimiser les temps d'attente des paquets réponse, ce champ n'est pas égal au nom de la λ-expression, mais il est égal au nom de l'expression émettrice du paquet requête-fonction. Par conséquent, il n'y a pas de génération de paquet réponse en attente par l'unité de contrôle lorsque celle-ci reçoit un paquet requête-fonction.

VI.5.4 Sur réception d'un paquet requête-copie

Ce type de paquet est toujours destiné à une expression représentant un noeud @ du graphe. Il contient la liste des liens {paramètres formels/paramètres réels}. Une demande de lecture est alors envoyée vers la mémoire associative des programmes. Si l'expression est trouvée, une copie de cette expression est réalisée. Dans cette copie les paramètres formels sont remplacés soit par les paramètres réels soit par les références vers les paramètres réels (fig 6.7).

Afin d'avoir un fonctionnement correct, il est nécessaire de distinguer les différents appels d'une même fonction. Pour cela, les demandes d'évaluation d'une fonction f sont envoyées vers l'unique expression qui définit le corps de la fonction f. Cette expression correspond à un noeud @. Elle est appelée *original de la fonction f.*

Toutes les expressions @ qui composent le corps de la fonction, contiennent, en plus des champs vus précédemment, un champ appelé *numéro de contexte.* Ce numéro de contexte fonctionne comme un compteur qui est initialisé à zéro (lors de l'écriture de l'expression en

mémoire), et qui s'incrémente de 1 à chaque nouvelle demande d'évaluation (c'est à dire pour chaque paquet requête-copie qui arrive à l'expression). Cette nouvelle valeur du compteur est alors copiée et mise dans le champ numéro de contexte se trouvant dans la copie de l'expression qui a été faite. Dans la suite du traitement, l'expression copiée est référencée par le couple (nom de l'expression, numéro de contexte). Ces 2 informations représentent respectivement la partie statique et dynamique du nom de l'expression, comme ceci a été expliqué dans le chapitre III §2.

Une fois la copie réalisée il faut, après cela, évaluer cette nouvelle expression. Si l'expression copiée correspond à l'application d'une fonction sur des arguments, un paquet requête-fonction est envoyé vers la λ–expression qui définit cette fonction. De ce fait une expression du type λ n'est jamais dupliquée.

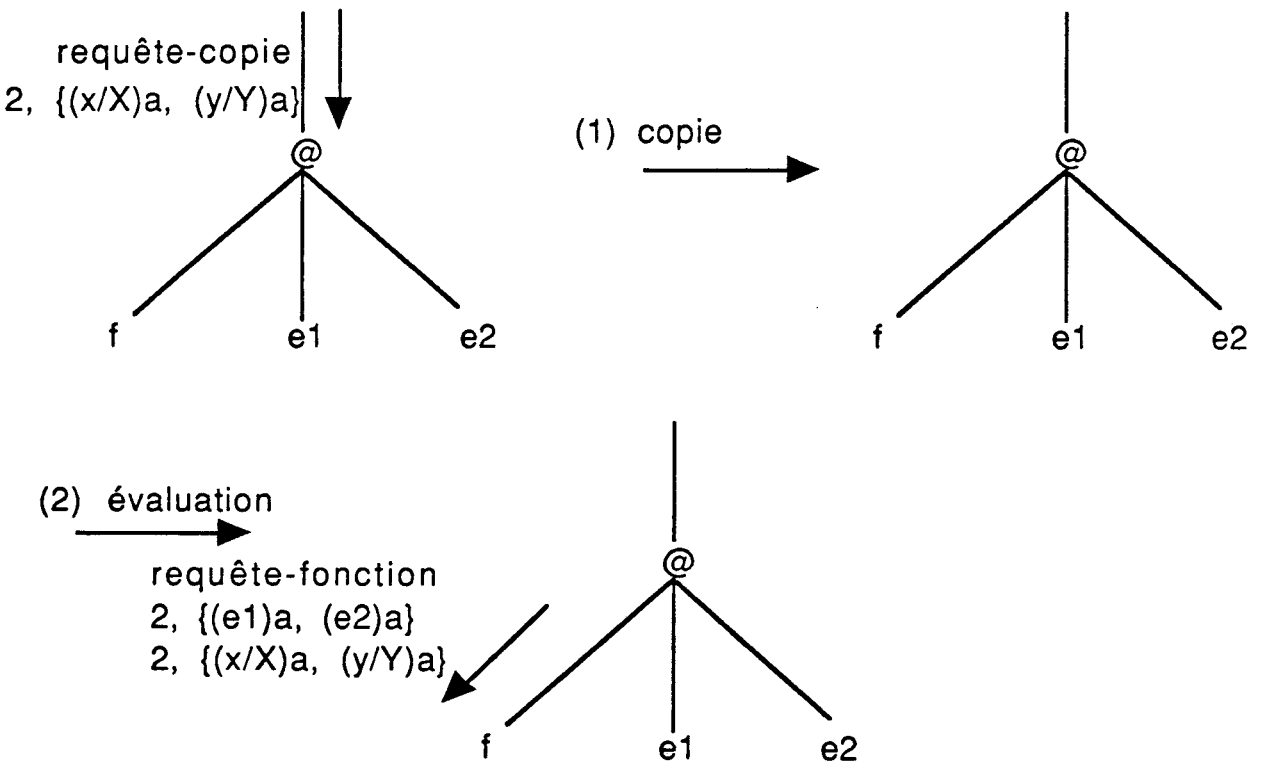


fig 6.7 : Interprétation d'un paquet requête-copie (cas d'une fonction)

Si l'expression copiée correspond par contre à l'application d'un opérateur élémentaire, des paquets requête-copie sont générés pour chacun des paramètres qui a sa marque égale à "a" (fig 6.8). Ces paquets requête-copie contiennent la liste des liens reçue précédemment.

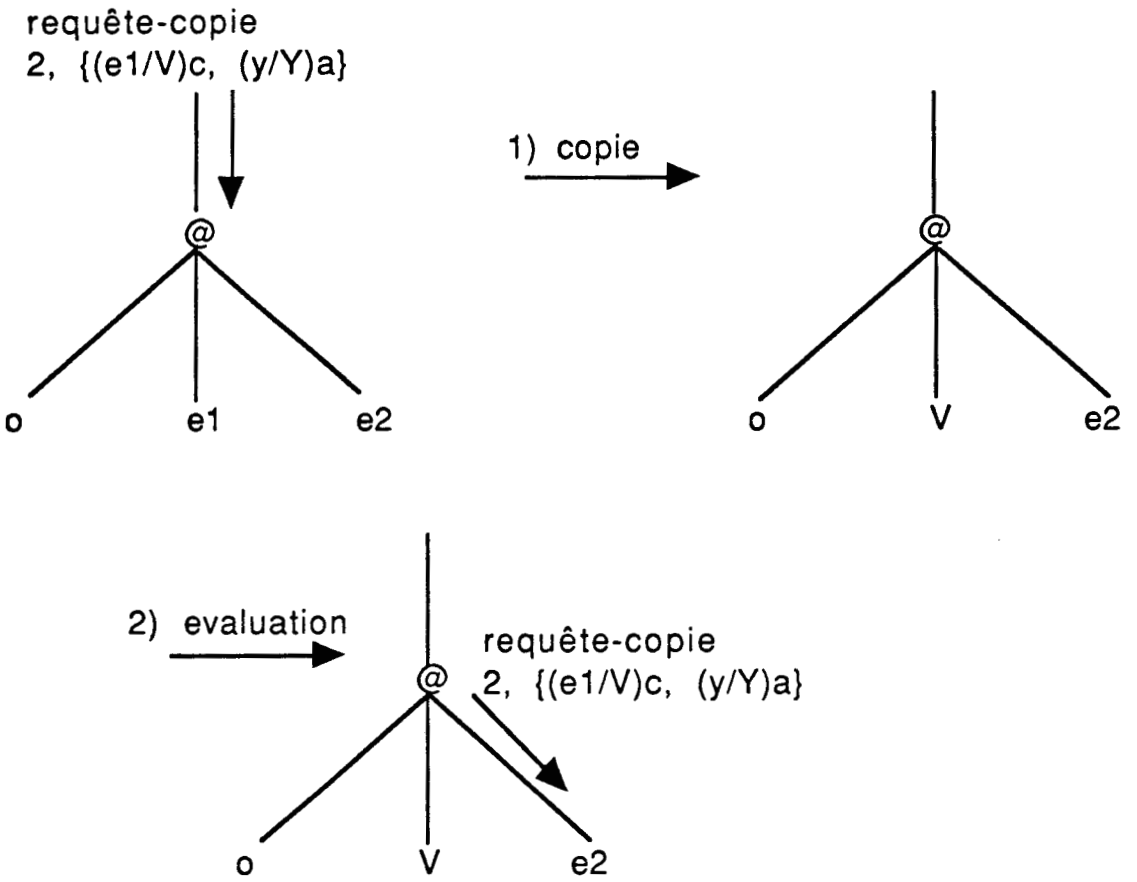


fig 6.8 : Interprétation d'un paquet requête-copie
(cas d'un opérateur élémentaire)

VI.5.5 Sur réception d'un paquet réponse

Un paquet réponse est toujours destiné à une expression qui correspond à un noeud @. Par conséquent, l'expression réceptrice est repérée par le couple (nom de l'expression, numéro de contexte).

Si la marque du premier argument au niveau de l'expression réceptrice est égale à "o", le nom de l'expression évaluée est remplacé par la valeur reçue dans le paquet réponse.

Si par contre, la marque du premier argument est "f", il s'agit d'une mise à jour de toute l'expression (retour du résultat de l'application de la fonction sur des arguments). Dans ce cas, l'expression est accédée, la marque de celle-ci est mise à "N" et la valeur reçue dans le paquet réponse est écrite. Afin de mettre à jour le paquet réponse qui est en attente du résultat de l'expression, l'état de l'expression est mis à calculable. L'unité d'exécution lira ensuite cette expression; comme la

valeur est déjà calculée, il y a seulement envoi de cette valeur à la mémoire des paquets.

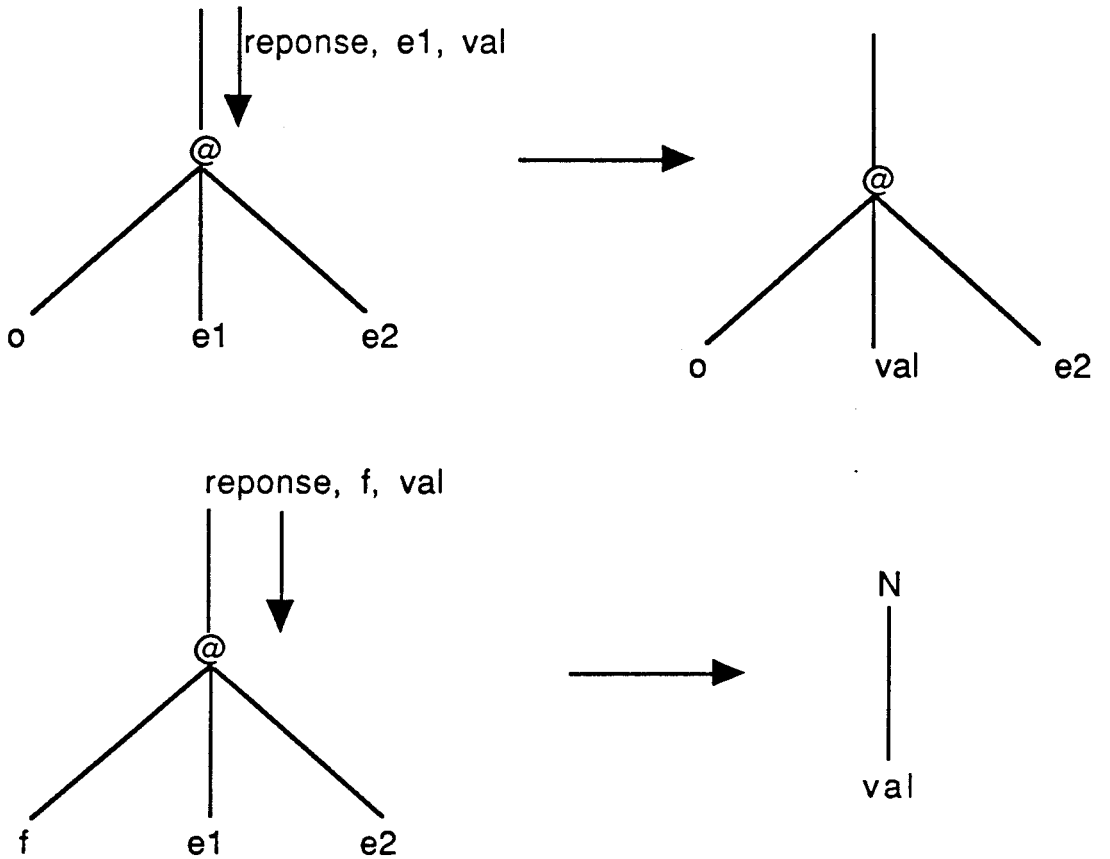


fig 6.9 : Interprétation d'un paquet réponse

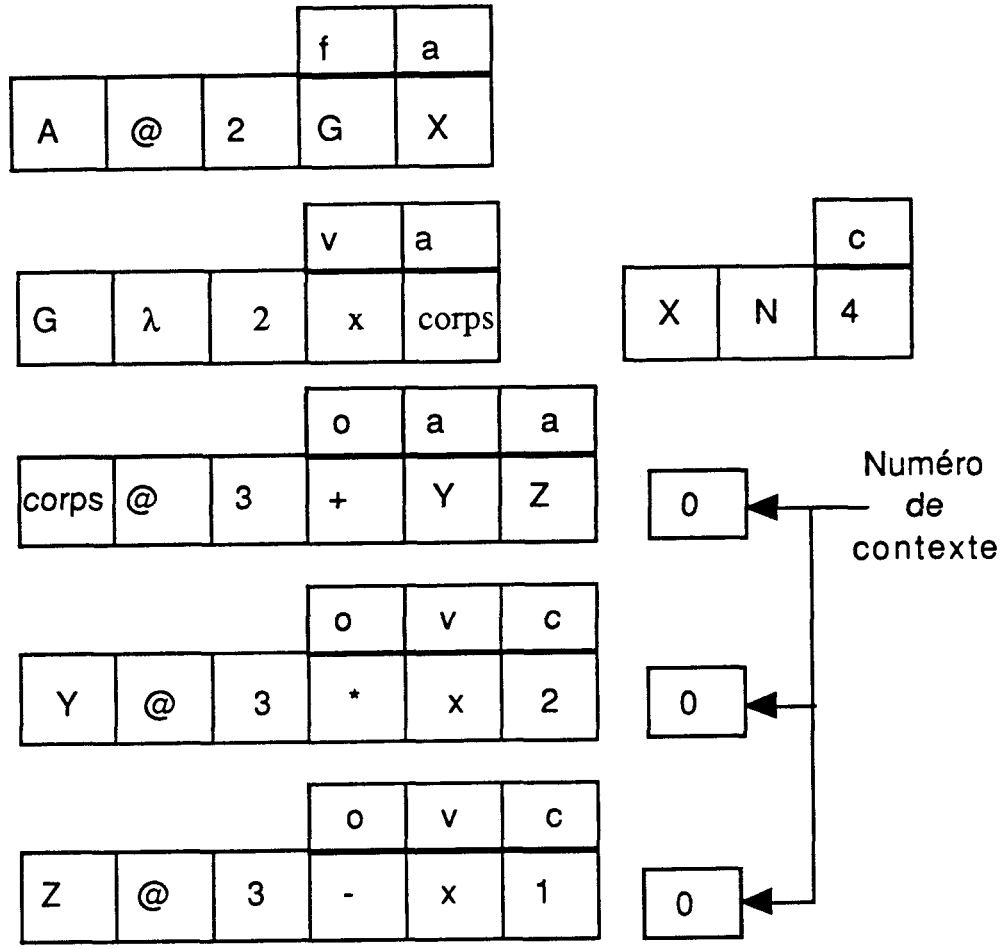
VI.6 Un exemple

Afin d'illustrer le fonctionnement de l'émulateur, nous allons donner un exemple de traitement de fonction :

Supposant que G = λx. (x * 2)+(x - 1) et calculant l'application

A = G 4

Le graphe de la fonction est représenté comme suit :

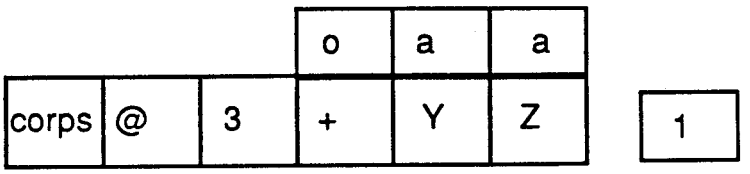


Le processeur hôte commence le traitement en envoyant un paquet requête vers l'expression A. Lorsque le noeud contenant cette expression a reçu le paquet requête pour A, un paquet requête-fonction est envoyé vers la définition de la fonction G (la λ-expression). Ce paquet contient le nom de l'expression émettrice du paquet requête-fonction (qui est A), et une référence vers le paramètre réel X. Comme l'environnement d'évaluation de la fonction est vide, la liste des liens ne contient aucun élément.

La λ-expression dont le nom est G va générer lors de la réception du paquet, un paquet requête-copie destiné à l'expression définissant le corps de la fonction. Ce paquet contient en plus du nom de l'expression demandée, qui est "corps", le nom de l'expression qui recevra le paquet réponse (c'est à dire "A") et la liste des liens {(x/X)a}.

Lorsque le noeud où se trouve la définition de "corps" reçoit le paquet requête-copie, le numéro de contexte de l'expression "corps" est incrémenté et une copie de l'expression est réalisée.

La copie est mise à l'état "en cours d'évaluation" et son contenu est le suivant :

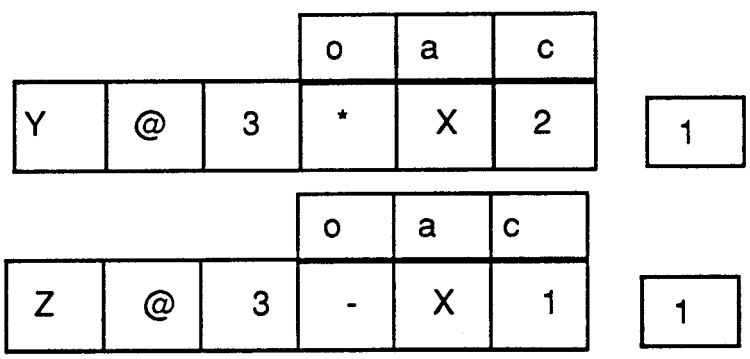


Un paquet réponse est mis en attente du résultat de "{corps, 1}" pour "A". Comme l'expression "corps" contient 2 références, 2 paquets requête-copie sont générés, un pour Y et un pour Z. Ces 2 paquets contiennent la liste des liens réduite à un seul élément ({x/X}a).:

requête-copie, Y, noeud-destination, {corps, 1}, noeud-émetteur, ({x/X}a)

requête-copie, Z, noeud-destination, {corps, 1}, noeud-émetteur, ({x/X}a)

Lorsque ces 2 paquets arrivent à leurs destinataires (là où sont définies les expressions Y et Z), une copie de chacune de ces 2 expressions est alors réalisée, après avoir incrémenté le numéro de contexte dans chacune de ces 2 expressions.



Deux paquets réponse en attente sont déposés dans la mémoire des paquets dans les (ou le) noeuds où se trouvent les 2 expressions Y et Z. Ces paquets réponse ont le format suivant :

Réponse, {corps, 1}, noeud-destination, {Y, 1}, ?

Réponse, {corps, 1}, noeud-destination, {Z, 1}, ?

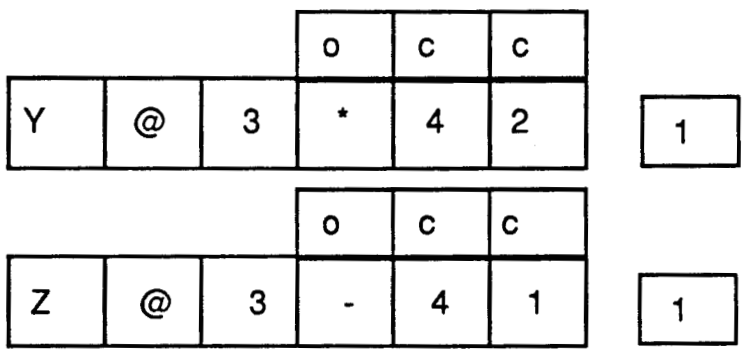
Le symbole "?" représente le champ valeur vide.

Après cette étape, on commence à évaluer chacune de ces 2 expressions. Pour cela 2 paquets requête sont envoyés vers l'expression de nom "X", un par {Y, 1} et un par {Z, 1}.

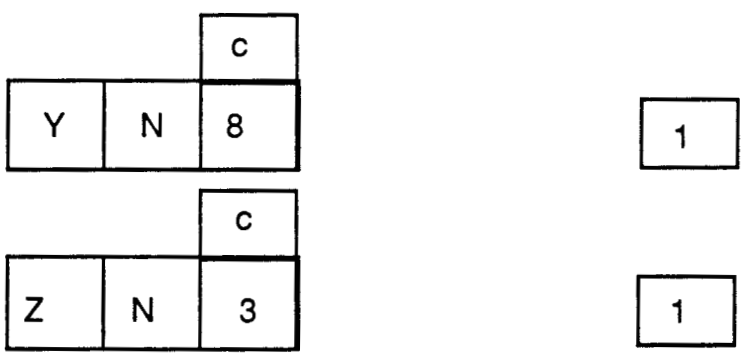
En effet afin de réduire la taille des paquets transmis, si l'argument de l'expression correspond à un paramètre réel, il y a émission d'un paquet requête au lieu d'émettre un paquet requête-copie.

Un paquet réponse est par la suite retourné à chacune des 2 expressions qui ont demandé la valeur de X.

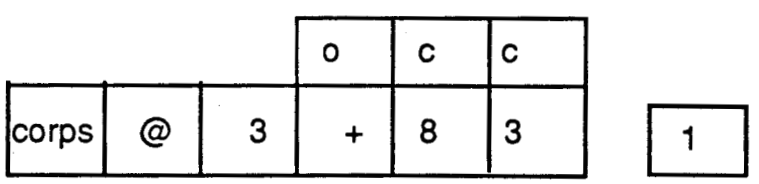
Ceci produit alors les 2 expressions calculables suivantes :



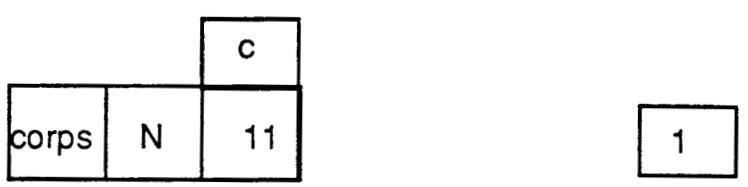
qui sont transformées en :



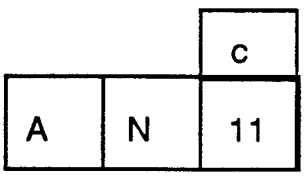
Les paquets réponse en attente pour ces 2 résultat sont complétés puis envoyés vers {corps, 1}. Cette expression devient alors



puis après évaluation



Le paquet réponse en attente de {corps, 1} pour l'expression A, est complété puis envoyé vers cette expression. Comme la marque du premier argument de l'expression a est "f", la valeur 11 est écrite et l'expression A devient



L'état de cette expression est mis à calculable. En utilisant le mécanisme décrit dans le paragraphe précédent, le paquet réponse en attente de la valeur de A est complété et envoyé vers le processeur hôte.

VI.7 L'évaluation de l'opérateur IF

Jusqu'à présent nous avons supposé que les opérateurs élémentaires utilisés dans le programme fonctionnel, étaient tous stricts (cf chapitre II §2). Néanmoins, il est intéressant, parfois même indispensable d'utiliser des opérateurs non stricts comme le IF.

En effet lorsqu'une expression contenant l'opérateur IF reçoit une demande d'évaluation par un paquet requête-copie, par exemple, c'est uniquement l'expression définissant la condition qui est d'abord évaluée

Lorsque la valeur (vrai ou faux) a été retournée à l'expression, on demande à ce moment l'évaluation de l'une des 2 alternatives.

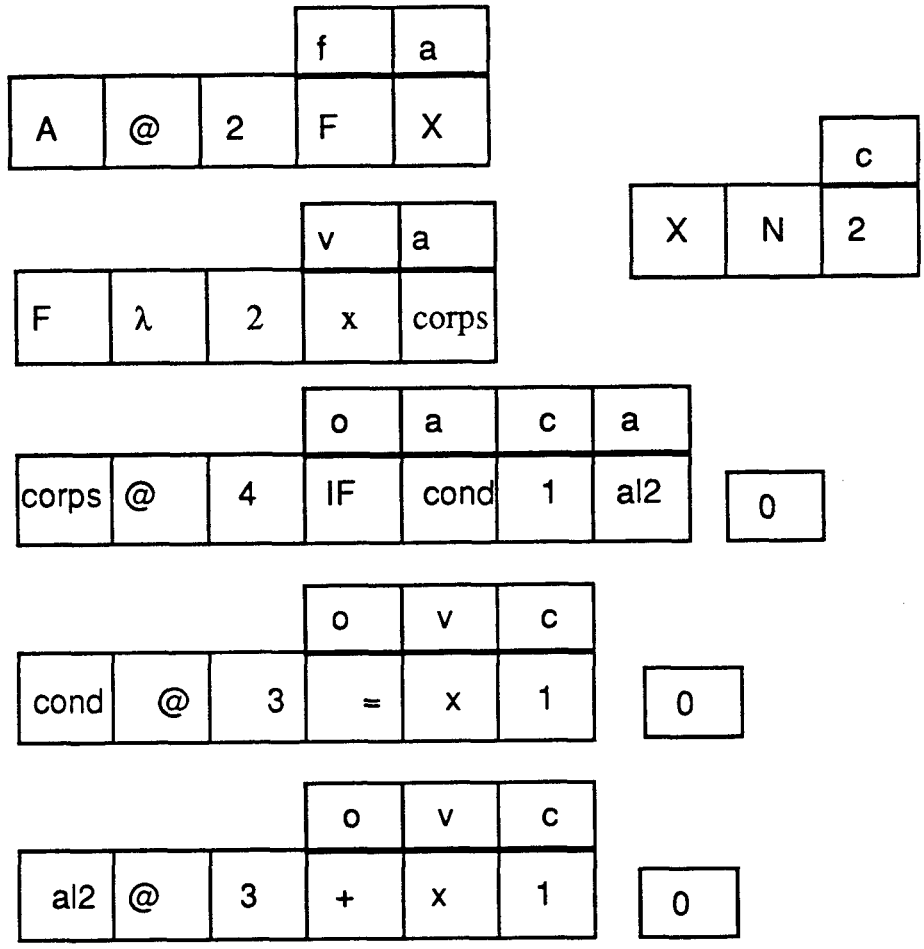
Voyons sur un exemple quels sont les problèmes posés par l'implémentation de cet opérateur.

Soit f une fonction définie par :

F = λx. If (x=1) Then 1 Else (x+1)

Calculons A=F 2

L'image du programme en mémoire est la suivante :



En utilisant le modèle de traitement décrit précédemment, l'expression "corps" reçoit un paquet requête-copie pour l'expression "A" avec la liste des liens {(x/X)a}. Le numéro de contexte de l'expression corps est alors incrémenté et une copie de l'expression est réalisée.

Après cette étape un paquet requête-copie est émis vers l'expression "cond" pour l'expression {corps, 1}. La liste des liens contenue dans ce paquet contient un seul élément qui est {(x/X)a}. A ce moment l'évaluation de l'expression est suspendue jusqu'à ce que la condition soit évaluée.

Par la suite l'expression "cond" peut être évaluée. Là aussi, le numéro de contexte est incrémenté et une copie de l'expression est réalisée. Dans cette nouvelle copie le paramètre formel x est remplacé par la référence X. Comme l'opérateur de comparaison "=" est strict, on demande l'évaluation de l'expression de nom "X". Un paquet requête est alors émis vers "X" pour {cond, 1}. Sur réception de la réponse X = 2, l'expression {cond, 1} peut poursuivre son évaluation. Ceci donne "faux" comme valeur à cette expression. Cette valeur est ensuite retournée à l'expression {corps, 1} qui peut, à son tour, poursuivre son évaluation.

Ainsi comme la condition est fausse, il faut évaluer la deuxième alternative. Un paquet requête-copie doit être émis vers l'expression "A12". Pour cela il est nécessaire d'envoyer dans ce paquet la liste des liens qui a été reçue dans le paquet requête-copie arrivé à l'expression {corps, 1}.

Afin que cela soit possible il est utile de mémoriser cette liste de liens dans une mémoire prévue à cette effet. Cette mémoire est appelée *mémoire des liens.*

Par conséquent, lorsqu'un paquet requête-copie arrive demandant l'évaluation de l'expression "corps", une structure de données dans la mémoire des liens est créée. Cette structure contient les champs suivants:

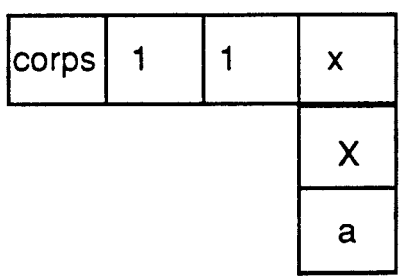
- Le nom de l'expression demandée.
- Le numéro de contexte.
- La longueur de la liste des liens.
- La liste des liens qui contient pour chaque paramètre formel les éléments suivant:

_Le paramètre formel

_Le paramètre réel

_La marque spécifiant s'il s'agit d'un paramètre réel (c) ou d'une référence vers ce paramètre (a)

Nom expression demandée = corps
Numéro de contexte = 1
Nombre de paramètres formels = 1
Liste des liens = {(x/X)a}



En utilisant cette structure, il est alors possible de reprendre l'évaluation de l'expression {corps, 1}. Ainsi, un paquet requête-copie est construit afin d'évaluer l'expression "a12". Dans ce paquet nous retrouvons la liste des liens lue à partir de la mémoire des liens.

Remarquons que les accès à cette mémoire sont du type associatif car ils se font en spécifiant une partie de la structure (le nom de l'expression et le numéro du contexte). Par conséquent cette mémoire pourrait être réalisée à l'aide d'une mémoire associative.

Une structure dans cette mémoire de liens n'est créée que si la marque du champ condition (c'est à dire le deuxième argument d'une

expression qui contient "IF" comme opérateur) a la valeur "a" et que l'une des 2 autres marques (correspondant à l'une des 2 alternatives) a sa marque égale à "a" aussi. Si par contre, la marque du champ condition n'est pas égale à "a", ou bien que les deux autres marques sont toutes les 2 à "c" ou, à "v", il n'y pas de création de la structure dans la mémoire des liens.

Dans cette mémoire, l'espace occupé par une structure peut être immédiatement récupéré dès que le paquet requête-copie pour l'une des 2 alternatives a été envoyé.

VI.7 La répartition des copies de fonctions

Dans l'émulateur actuel, les paquets initialisation sont utilisés pour charger le programme dans les différentes mémoire du réseau. Une fois le programme chargé, il n'y a plus de paquet initialisation qui circule dans le réseau. Ceci n'est plus valable lorsque le programme contient des fonctions qui peuvent être appliquées plusieurs fois.

Voyant tout d'abord comment peut être implémenté le mécanisme de recherche des expressions.

La recherche d'une expression est réalisée par le biais des 2 informations :

-Le nom de l'expression (partie statique)

-Le numéro de contexte (partie dynamique)

Selon que les fonctions de hachage sont appliquées sur uniquement la partie statique ou bien sur les 2 parties statique et dynamique, nous pouvons avoir 2 méthodes de répartition des copies d'une fonction.

1) Application des fonctions de hachage uniquement sur la partie statique

Dans cette première solution (fig 6.10), chaque fois qu'il est nécessaire d'appliquer les fonctions de hachage, il y a extraction de la partie statique. Lorsqu'une recherche est réalisée en mémoire par contre ce sont les 2 parties qui sont utilisées.

De ce fait, lorsqu'une copie d'une expression est demandée, si suffisamment d'espace mémoire existe dans la mémoire des programmes où est définie l'original, la copie aura lieu localement. Par conséquent, toutes les copies d'une même fonction vont se trouver mémorisées dans un seul noeud du réseau. Ceci jusqu'à ce qu'il se crée

une situation de collision. A partir de ce moment, les prochaines copies de la fonction doivent être stockées dans la mémoire de l'un des voisins.

Lorsqu'un paquet requête-copie arrive à un noeud pour demander une copie et une évaluation d'une fonction et que la mémoire associative des programmes ne contient plus d'espace pour stocker la copie, celle-ci est mise dans un paquet initialisation en collision ensuite envoyée vers l'un des voisins. La valeur du numéro de contexte qui accompagne le nom de l'expression dans ce paquet n'est pas nulle, mais égale au contenu du compteur numéro de contexte de l'originale.

Après cette étape un paquet requête en collision est envoyé vers le noeud où sera mémorisée la copie.

De ce fait les copies de l'expression se trouveront dans un chemin hamiltonien dont la racine est le noeud qui contient l'originale de l'expression.

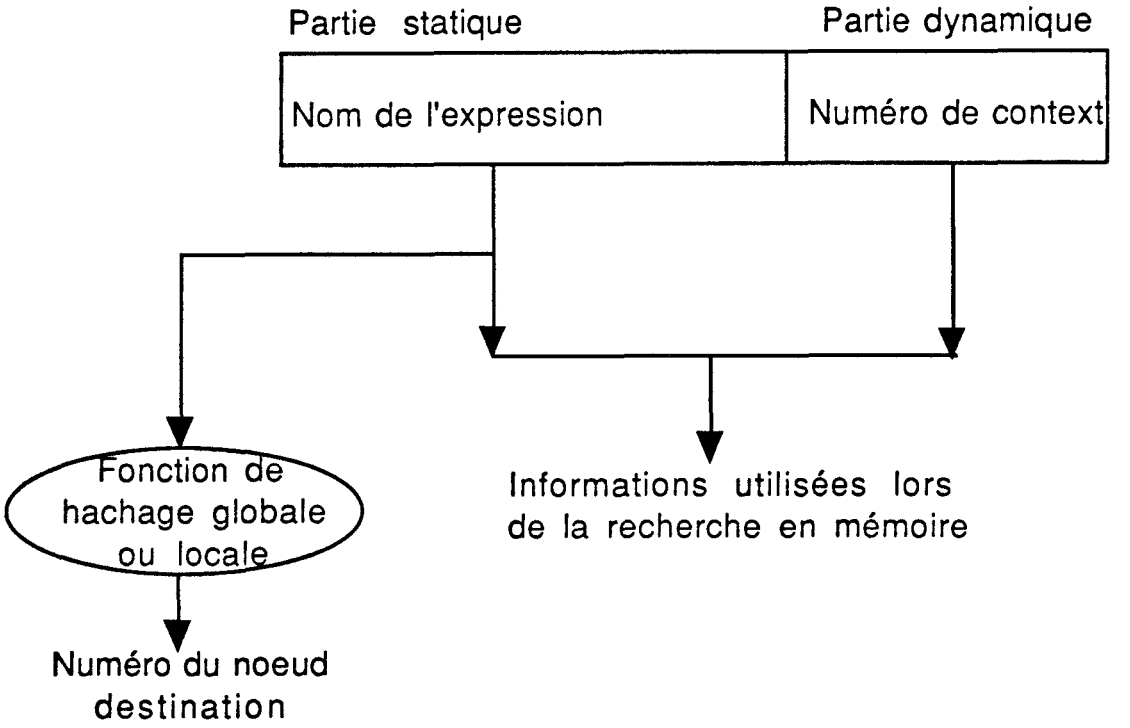


fig 6.10 : Implémentation du numéro de contexte dans les expressions

2) Application des fonctions de hachage sur les 2 parties

Dans cette méthode lorsqu'une copie d'une expression est demandée, on applique la fonction de hachage globale sur le couple (nom

expression, numéro de contexte) pour déterminer dans quel noeud sera mémorisée la nouvelle copie. Un paquet initialisation en routage, contenant la copie de l'expression, est alors construit puis envoyé vers le noeud destination précédemment calculé.

Une fois la copie envoyée, un paquet requête en routage est aussi construit et envoyé vers le noeud qui contient la copie, ceci afin de commencer l'évaluation de la copie.

Comme on peut le voir, cette méthode permet d'exploiter le parallélisme dans le traitement des différentes copies de la fonction.

VI.8 Conclusion

Dans ce chapitre nous avons montré que le noyau de l'émulateur de la machine N-ARCH peut être étendu afin de permettre l'évaluation des fonctions. Comme ceci a été précisé dans ce chapitre, d'autres extensions sont à l'étude. A ce propos, une implémentation sur un réseau de Transputer est en cours de mise au point afin de permettre la manipulation des listes. Dans cette implémentation, l'application d'une fonction sur les éléments d'une liste est faite de manière parallèle.

Nous pensons que toutes ces possibilités permettront à l'émulateur d'exécuter des programmes fonctionnels standards, et qu'il sera alors possible de tester les différentes stratégies d'exécution dans la machine N-ARCH (méthode de répartition des appels de fonction, duplication ou non des structures de données..etc) et aussi de comparer les performances de la machine N-ARCH par rapport aux autres machines parallèles.

CONCLUSION GENERALE

Le travail que nous venons de présenter nous a permis d'étudier le domaine des architectures parallèles. Dans cette étude, nous avons proposé un certain nombre de solutions afin de résoudre une partie des problèmes posés lorsque plusieurs processeurs coopèrent pour exécuter une tâche. Nous ne prétendons nullement avoir résolu tous les problèmes. Bien au contraire, l'une des conclusions aux quelles nous sommes arrivée est que beaucoup de travail reste à faire afin de concevoir des machines parallèles qui répondent aux exigences des nouveaux domaines d'application de l'informatique.

Nous avons vu dans le chapitre II que les modèles d'évaluation non von-Neumann (par réduction ou contrôlés par les données) étaient attractifs mais que leur utilisation posait des problèmes (charge système, évaluation de ce qui n'est pas toujours nécessaire, manipulation des données structurées, ..etc).

Concernant les langages déclaratifs, nous pouvons dire que ces derniers disposent de très bonnes caractéristiques en vue d'une exécution parallèle (indépendance des tâches parallèles, transparence au référent, extraction implicite du parallélisme).

Néanmoins, afin que ces derniers soient l'outil de programmation des futurs ordinateurs parallèles, il faut que le programmeur qui les utilise change sa manière d'exprimer les problèmes. Ceci permettra en particulier à la machine d'exploiter pleinement le parallélisme existant dans les programmes.

L'analyse de "stricticité" et la transformation de programme sont des opérations réalisées avant compilation du code, qui permettront peut être d'avoir des programmes déclaratifs clairs, faciles à maintenir et à comprendre, et (enfin et surtout) efficaces, c'est à dire s'exécutant en un minimum de temps.

Plus particulièrement, concernant l'émulateur de la machine N-ARCH, nous avons montré que les caractéristiques de la machine sont très intéressantes et qu'elles permettent de résoudre un certain nombre de problèmes. Parmi les points intéressants, nous pouvons citer :

- La structure de la machine en mémoire privée, qui (utilisant une bonne méthode de répartition) minimise les temps d'attente durant les opérations de lecture et d'écriture de données.

- La topologie du réseau en hypercube permettant d'avoir un réseau qui offre de bonnes caractéristiques (un nombre de liens réduit par processeurs et des temps d'accès relativement courts).

- L'existence de mémoire associative, permettant de réaliser des opérations de correspondance de forme (pattern matching) dans des temps réduits. Ces opérations de correspondance de forme sont très fréquemment utilisées dans l'implantation des langages déclaratifs (sélection des règles pour une unification dans un programme logique par exemple).

- L'existence dans un noeud N-ARCH d'unités autonomes permettant de réaliser les opérations de communication indépendamment de l'activité de l'unité de contrôle du noeud. Ceci permet de réaliser un gain de temps, car il y a parallélisme dans le traitement des opérations d'émission/réception et l'exécution de la tâche par l'unité de contrôle. Pour cette raison l'utilisation du Transputer (et de son langage de programmation OCCAM) dans la conception de l'émulateur a énormément contribué aux bons résultats d'exécution obtenus.

Les tests réalisés sur l'émulateur ont aussi permis de montrer certains choix du projet qu'il faut revoir, ou tout au moins qu'il faut approfondir.

Le point qui nous semble le plus critiquable est la politique de répartition du programme. En effet, l'application directe de la fonction de hachage globale sur les noms des objets du programmes risque de générer une très mauvaise répartition du programme.

La méthode de répartition par taux de remplissage que nous avons proposée dans le chapitre IV, peut palier un mauvais choix de cette fonction de hachage globale. Pour cela une implantation et des tests sont nécessaires.

Néanmoins, étant donné que les programmes déclaratifs sont souvent récursifs et que le nombre ou une fonction est appliquée peut être très différent par rapport au nombre ou une autre fonction du programme est appelée, toute répartition statique risque de provoquer de mauvaises performances du point de vue temps d'exécution. Pour cela il est nécessaire d'utiliser une méthode de répartition dynamique.

En tout état de cause, les résultats d'exécution de programmes de test sur l'émulateur ont permis de valider le modèle et ont montré qu'il est possible (en général) de réduire le temps d'exécution en multipliant le nombre de processeurs. Ces résultats ont montrés aussi qu'une très grande partie du temps CPU est utilisée pour simuler les accès aux mémoires associatives et pour appliquer les fonctions de hachage locale ou globale sur le nom de l'expression recherchée. De ce fait, il est nécessaire d'intégrer ces deux opérations dans le matériel.

Ces tests ont aussi montré que l'utilisation du phénomène de collision pour répartir le programme n'est intéressante que dans certains cas (mauvaise fonction de hachage globale par exemple) et qu'en général les temps d'exécution dans ce cas sont relativement importants.

Dans le chapitre VI, nous avons montré que l'émulateur réalisé peut facilement être étendu pour supporter l'exécution des langages de programmation fonctionnels réels. L'extension que nous avons proposée concerne l'évaluation des fonctions. Celle-ci peut être étendue et complétée afin de supporter des programmes fonctionnels.

Nous espérons enfin, que cette étude sera profitable pour la suite du projet, et qu'elle a fait apparaître les points importants à prendre en compte lors de la réalisation matérielle du noeud de la machine N-ARCH.

REFERENCES

[Amam87] M.Amamiya :
Data flow computing and parallel reduction machine.
Frontiers in computing. Amsterdam December 1987.

[Atki87] P.Atkin :
Performance maximisation.
Technical note, N17 Inmos 1987.

[Back78] J.Backus :
Can programming be liberated from the von Neuman style of programming?.
CACM Vol 21 August 1978, P613-641.

[Bako87] P.Bakowski:
Manuel d'utilisation du langage de description de matèriel LIDO.
USTL LIFL, 1987.

[Bell86] M.Bellia, G.Levi :
The relation between logic and functionnal languages: A survey.
The journal of logic programming. March 1986.

[Berk75] K.J.Berkling :
Reduction languages for reduction machines.
Proceedings of the international symposium on computer architecture. Janv 1975.

[Bhuy82] L.M.Bhuyan, P.Agrawal:
A general class of processor interconnection strategies.
Proc of the 9 th annual symposium on computer architecture 1982.

[Bour88] A.Bourzoufi:
Etude des fonctions de hachage dans un réseau hypercube pour la machine N-ARCH.
Mémoire de DEA, USTL, LIFL, 1988.

[Burn86] D.Mc.Burn, M.R.Sleep:
Transputer based expirements with ZAPP architecture.
University of East-anglia. Norwich.
Internal report sys-c86-16, Nov 1986.

[Burn88] A.Burn:
Programming in OCCAM2.
Addison-Weslay 1988.

[Burt87] F.W.Burton :
Functionnal programming for concurrent and distributed computing.
The Computer Journal Vol 30 n° 5 1987.

[Cast86] M.Castan :
Mécanismes de base pour la réduction parallèle.
Bigre+Globule n°50, Septembre 1986.

[Cone85] J.S.Conery, D.F.Kibler :
And-parallelisme and non determinism in logic programs.
New generation computing, March 1985.

[Cous88] E.Cousin :
Les machines à réduction et l'intélligence artificielle.
La recherche n° 204, Novembre 1988.

[Cousi86] G.Cousineau, P.L.Currien :
La machine abstraite catégorique CAM.
BIGRE+GLOBULE No 50 , Septembre 1986.

[Crip86] M.D.Cripps, A.J.Field, M.J.Reeve:
The design and implementation of ALICE.
In functional programming: Languages, tools and architectures.
Ed Eisenbach, E.Herwood publishers 1986.

[Darl81] J.Darlington, M.J.Reeve:
ALICE : A multiprocessor reduction machine for the parallel evaluation of applicative languages.
ACM/MIT Conf on functionnal programming languages and computer architecture. 1981.

[Darl87] J.Darlington :
Software developement using functional programming languages.
Internal report, Imperial college London, Jan 1987.

[Denn74] J.B.Dennis :
First version of a data flow procedure language.
Lecture notes on computer science 1974.

[Deve88] N.Devesa, G.Goncalves, M.P.Lecouffe, B.Toursel:
A controlled reduction model of functionnal programs on a distributed associative network.
Euromicro Zurich 1988.

[Eage89] D.L.Eager, J.Zahorjan, E.D.Lazowska:
Speedup versus efficiency in parallel systems.
IEEE transactions on Computers vol38, n3, March 1989, p408.

[Germ89] C.Germain, J.P.Sansonnet :
L'architecture des machines massivement parallèles pour l'intelligence artificielle.
Dans Architectures avancées pour l'intelligence artificielle.
Onzième journées francophones sur l'informatique.
Nancy 1989, EC2 Editeur.

[Gonc87] G.Goncalves, M.P.Lecouffe, B.Toursel, S.Niar:
A network of Transputers to emulate a parallel symbolic processor.
Microprocessing and Microprogramming Nbr 23, 1988, p 149.

[Gonc89] G.Goncalves, S.Niar , M.P.Lecouffe, B.Toursel:
Performance evaluation of a parallel machine emulator.
6th symposium on microcomputer and microprocessor applications. Budapest Oct 1989.

[Hann89] I.Hannequin, G.Goncalves, P.Lecouffe, B.Toursel
PROLOG : A new parallel evaluating scheme.
EUROMICRO Cologne, Septembre 1989.

[Heid87] P.Heidelberg, S.Lavenberg:
Computer performance evaluation methodology.
IEEE Transactions on Computers, Dec 1984.

[Hend80] P.Henderson :
Functional programming, application and implementation.
Prentice-Hall 1980.

[Hwan87a] K.Hwang, J.Ghosh, R.Chowkwanyun:
Computer architectures for artificial intelligence processing.
COMPUTER, January 1987.

[Hwan87b] K.Hwang, F.Briggs :
Computer architecture and parallel processing,
MC Graw-Hill 1987.

[Inmo87] INMOS 1987:
IMS T414, notice de présentatation.

[Iqba86] M.A.Iqbal, J.H.Salts, S.H.Bokhari:
A comparative analysis of static and dynamic load balancing strategies.
Proceedings of the 1986 International conference on parallel processing, 1986 p1040.

[Kenn83] J.R.Kennaway, M.R.Sleep:
Novel architectures for declarative languages.
Software and Microsytems, June 1983.

[Lewi88] T.Lewis, C.Curtis :
Hashing for dynamic and static internal tables.
Computer, Oct 1988, p45.

[Magò80] G.Magò :
A cellular computer architecture for functional programming.
Proceedings of IEEE compcon 1980.

[Munt88] T.Muntean:
Les supers calculateurs à Transputers.
La Recherche Novembre 1988.

[Mura84] K.Murakami, T.Takuta, R.Onai :
Architecture and hardware: Parallel inference machine and knowledge base machine.
Proc of the inter conf on fifth generation computer systems 1984. p18-35.

[Niar86] S.Niar :
Architectures parallèles et programmation logique.
Mémoire de DEA, USTL, LIFL, Octobre 1986.

[Niar87] S.Niar, G.Goncalves, B.Toursel, M.P.Lecouffe:
The OCCAM process of the N-ARCH kernel.
Proceedings of the 7th conference of the OCCAM user group.
Grenoble 1987.

[Niar89] S.Niar , G.Goncalves, M.P.Lecouffe , B.Toursel:
The evaluation of the N-ARCH emulator on a transputer network.
EUROMICRO Cologne, Septembre 1989, (paraitra dans le journal EUROMICRO).

[Onai85] R.Onai, M.Aso, K.Masuda :
Architecture of a reduction-based parallel inference machine: PIM-R.
New Generation Computing, Mars 1985.

[Peyt86] S.Peyton Jones :
The implementation of functional programming languages.
Prentice-Hall 1986.

[Peyt87] S.Peyton Jones:
GRIP: A high performance architecture for parallel graph reduction
Functional programming and computer architecture 1987.
LNCS 274 Springer Verlag

[Peyt89] S.Peyton Jones :
Parallel Implementations of functional programming languages.
The computer journal Vol 32, n° 2 1989.

[Poun87] D. Pountain:
A tutorial introduction to OCCAM programming.
INMOS 1987.

[Silb88] A.Silbey, V.Milutonivic:
Advanced microprocessors and high level languages.
Dans V.Milutonivic: Computer Architecture.
North-Holland 1988.

[Tour86] B.Toursel, M.P.Lecouffe, G.Goncalves:
Le projet N-ARCH, conception d'une architecture parallèle pour les programmes déclaratifs,.
USTL LIFL, note interne N 48, 1986.

[Tour87] B.Toursel, G.Goncalves, M.P.Lecouffe :
N-ARCH, une architecture parallèle pour les programmes déclaratifs.
BIGRE+GLOBULE n56, Novembre 1987.

[Trel82] P.Treleaven, D.Brownbridge, R.Hopking :
Data driven and demand driven computer architecture.
Computer Surveys, vol 14, N° 1, Mars 1982.

[Turn79] D.A.Turner :
A new implementation technique for applicative languages.
Software-practice and experience, vol 9, 1979.

[Vegd84] S.R.Vegdahl :
A survey of proposed architectures for execution of functional languages.
IEEE Transactions on Computers Vol 33 n° 12 December 1984.

[Wats87] P.Watson, I.Watson:
Evaluating functional programs on the FLAGSHIP machine.
Lectures notes on Computer Sciences n 274.
Springer Verlag 1987.

[Wei88] Y.Wei, J.L.Gaudiot:
Demand driven interpretation of FP programs on a data flow multiprocessors.
IEEE Transactions on Computers Vol 37 n° 8 August 1988.

Title :
A Contribution to a Study of Parallel Computer Architecture :
Structure of a N-ARCH Node and Emulation on a Transputer Network

This work presents a contribution to the N-ARCH project. The aim is the design of a parallel computer architecture oriented to symbolic processing.

In the first part of this report, we present the main ideas retained in the design of the architecture. These ideas consist in using declarative languages and a non_von Neumann parallel architecture.

After this step, a functional description of a node is detailed. In this description, we show the important role of content adressable memories at the node level.

Subsequently, the load balancing technique implemented in the N-ARCH machine is presented. This technique is based on the using of hashing functions.

From this description, an emulator of the N-ARCH architecture has been realised. The kernel of the emulator has been written in the parallel programming language OCCAM and developed on a 16-Transputer network. The emulator has allowed us to study the dynamic behaviour of the model.

Afterwards, tests programs have been executed on the emulator. These tests have allowed us to measure the performances of the emulator and to valid the N-ARCH model of execution. The results obtained have shown also, some important points of the N-ARCH machine which must be taken into account during the VLSI realisation of the node.

These points concern the integration of the associative memories, the implementation of dynamic load balancing, and the benefits of autonomous communication controllers.

This work is concluded with a proposition concerning the integration of function evaluation in the emulator. This extension is the first step towards the execution of functional programs on the N-ARCH emulator.

Key-words :
Parallel computer architecture - Non von-Neumann control mode
Declarative languages - Emulation
Transputer network - Speedup

Résumé :

Le présent travail constitue une contribution au projet N-ARCH qui vise la conception d'une architecture d'ordinateur parallèle orientée vers le traitement symbolique.

Dans la première partie de ce rapport, nous présentons les 2 points principaux retenus pour la conception de la machine N-ARCH. Il s'agit d'une part de l'utilisation de langages de nature déclarative et d'autre part de l'adéquation d'un schéma d'évaluation de type non-Von Neumann à ces langages.

Une description fonctionnelle d'un noeud de la machine est, par la suite, présentée. Dans cette description nous montrons le rôle des mémoires associatives dans le fonctionnement d'un noeud, ainsi que la méthode de répartition des programmes utilisée. Cette méthode repose sur l'utilisation de fonctions de hachage.

A partir de cette description, un émulateur de la machine N-ARCH a été réalisé afin d'étudier le comportement dynamique du modèle. Le noyau de l'émulateur a été développé en OCCAM et mis au point sur un réseau de 16 Transputers.

Les programmes de test réalisés sur l'émulateur ont permis de mesurer les performances de celui-ci et de valider le modèle fonctionnel de N-ARCH.

Les résultats de ces tests mettent en évidence certains points clés qui devront être pris en compte lors de la réalisation physique du noeud (comme l'intégration des mémoires associatives, la répartition dynamique du programme, l'existence d'unités de communication autonomes).

Le travail se termine par une proposition d'extension du noyau N-ARCH pour permettre l'exécution de programmes écrits en langages fonctionnels.

Mots-Clés :
Architectures d'ordinateurs parallèles - Mode de contrôle non_von Neumann - Langages déclaratifs - Emulation
Réseau de Transputers - Accélération