Implémentation et évaluation d'un système logique parallèle

Jacques Chassin de Kergommeaux

▶ **To cite this version:**

Jacques Chassin de Kergommeaux. Implémentation et évaluation d'un système logique parallèle. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1989. Français. NNT : . tel-00122736

HAL Id: tel-00122736
https://theses.hal.science/tel-00122736

Submitted on 4 Jan 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TU 8 5

THÈSE

présentée par

Jacques CHASSIN de KERGOMMEAUX

pour obtenir le titre de **DOCTEUR**

de l'UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I

(arrêté ministériel du 5 juillet 1984)

Spécialité : **INFORMATIQUE**

IMPLÉMENTATION ET ÉVALUATION D'UN SYSTÈME LOGIQUE PARALLÈLE

Date de soutenance : *23 Novembre 1989*

Composition du Jury :	*Président*	**L. Trilling**
	Rapporteur	**D.H.D. Warren**
	Examinateurs	**J. Briat**
		J. Mossière
		J-C. Syre

Thèse préparée au sein du Laboratoire de Génie Informatique

Remerciements

Cette thèse n'a été possible que grâce au concours d'un grand nombre de personnes que je tiens à remercier très sincèrement.

Jacques Mossière tout d'abord, dont l'amitié et le soutien ne m'ont jamais fait défaut.

Jean-Claude Syre a dirigé le projet PEPSys et m'a offert la chance d'y contribuer.

D.H.D. Warren dont la collaboration a été précieuse tout au long du projet PEPSys a en outre accepté de juger un travail écrit en Français!

Jacques Briat m'a lancé dans ce domaine de recherche. Ses remarques judicieuses m'ont ouvert de nouveaux horizons.

Laurent Trilling a accepté de plonger dans ce manuscrit pour le juger et de présider le jury.

Hervé Gallaire a su créer à l'ECRC un environnement de travail dont pourraient rêver de nombreux thèsards. Je le remercie d'avoir accepté l'exploitation de mon travail à ECRC par cette thèse.

Le travail présenté dans cette thèse est avant tout un travail d'équipe d'où il m'a été bien difficile d'isoler ma contribution. Les membres "historiques" de l'équipe PEPSys : Max Hailperin, Michael Ratcliffe et Harald Westphal ont jeté les bases qui ont permis la suite du projet. Mon travail n'aurait pas été possible sans la coopération efficace et amicale de Philippe Robert, dont l'esprit inventif a surmonté une grande partie des difficultés que posait la mise en œuvre du modèle de calcul PEPSys. Wolfgang Rapp a apporté une importante contribution au travail d'implémentation, au cours d'un stage de fin d'études exceptionnel. Uri Baron a fait bénéficier l'ensemble de l'équipe de son humour ravageur et le simulateur de sa compétence et de sa rigueur. Les deux sympathiques VSNA, Bounthara Ing et Pierre Heuzé, ont largement dépassé les horaires d'un "service militaire" afin de développer des applications parallèles et les systèmes permettant de les mettre au point.

Jacques Noyé, Bruno Poterie et Hans Benker ont assuré un support technique efficace et amical durant les phases de définition de la machine abstraite et d'implémentation.

Pascal Van Hentenryck a prouvé que le parallélisme était compatible avec les contraintes en réalisant très efficacement le mariage de CHIP avec PEPSys. Ses remarques pertinentes ont contribué à l'amélioration du manuscrit.

Philippe Codognet a contribué à l'étude bibliographique et accepté de relire la première version de cette thèse.

Miguel Santana a chassé impitoyablement les imperfections laissées par les lecteurs précédents.

Bernard Cassagne et Jacques Eudes m'ont apporté une assistance technique précieuse durant la rédaction.

Last but not least Farid Ouabdesselam m'a donné l'idée d'écrire cette thèse et prodigué les conseils permettant de la mener à bien.

Résumé

Cette thèse est consacrée à l'implémentation de PEPSys (Parallel ECRC Prolog System) sur un multiprocesseur à mémoire partagée et à l'évaluation de cette implémentation. Le projet PEPSys vise à exploiter le parallélisme en programmation logique pour obtenir, sur les multiprocesseurs existants actuellement, des gains de performances relativement aux systèmes Prolog les plus efficaces. Un langage, extension de Prolog, un modèle de calcul et une machine abstraite basée sur la *WAM* ont été définis et validés par une implémentation sur multiprocesseur et une simulation d'architecture parallèle extensible. Le modèle de calcul supporte les parallélismes OU et ET indépendant ainsi que leur combinaison avec l'exécution séquentielle et le retour-arrière. L'implémentation de PEPSys qui fait l'objet de cette thèse constitue l'un des premiers systèmes logiques OU-parallèles à procurer des gains de performances, relativement aux systèmes Prolog séquentiels efficaces. Les nombreuses mesures présentées dans la thèse permettent de valider cette implémentation ainsi que les principaux mécanismes du modèle de calcul, tout en suggérant des optimisations.
Mots clés : PEPSys : Parallel ECRC Prolog System, Parallélismes OU et ET, Implémentation sur multiprocesseur, Prolog Parallèle basé sur la WAM, Mesures d'exécution, Gains de performances en Parallèle.

Abstract

This thesis is dedicated to the implementation of PEPSys (Parallel ECRC Prolog System) on a shared memory multiprocessor and to the evaluation of this implementation. The PEPSys project aims at exploiting the parallelism in logic programming to obtain, on existing multiprocessors, performance improvements compared to efficient sequential Prolog implementations. A language, extension of Prolog, a computational model and an abstract machine based on the WAM have been defined and validated by a multiprocessor implementation and a simulator of extensible architectures. The computational model supports OR and independent AND parallelisms and the combination of both with sequential execution and backtracking. The implementation of PEPSys which is the main topic of this thesis, is one of the first OR-parallel logic systems to provide efficiency improvements, compared to state of the art Prolog systems. The numerous measures provided in this thesis allow to validate this implementation and the main design decisions of the computational model, while suggesting improvements.
Keywords: PEPSys: Parallel ECRC Prolog System, OR-AND parallelism, Multiprocessor Implementation, WAM-based Parallel Prolog, Execution Measures, Parallel Speedups.

Table des Matières

I Contexte **17**

1 Introduction **19**
1.1 Intérêt de la programmation logique 19
1.2 Le parallélisme dans la programmation logique 20
1.3 Le projet PEPSys 21
1.4 Contribution de l'auteur au projet PEPSys 22
1.5 Plan de la thèse 23

2 La programmation logique, Prolog et la WAM **27**
2.1 Programmation logique 28
2.1.1 Quelques définitions de la logique des predicats du premier ordre 28
2.1.2 Interprétation et preuve par réfutation 29
2.1.3 Résolution SLD 30
2.1.4 Programmation logique 31
2.2 Prolog 32
2.2.1 Prolog pur 32
2.2.2 Contrôle et *cut* 33
2.2.3 Prédicats prédéfinis 34
2.2.4 Efficacité 34
2.3 Interprétation de Prolog 35
2.3.1 Représentation de l'état courant 35
2.3.2 Pile de restauration ou trace 37
2.3.3 Pile globale 39
2.3.4 Déréférencement, unification et liaison 39
2.3.5 Optimisation d'appel terminal 41
2.4 Présentation de la *WAM* 41
2.4.1 Intérêt de la notion de machine abstraite 41
2.4.2 Principes de la compilation de Prolog 42
2.4.3 Objets manipulés. Opérations élémentaires 42
2.4.4 Zones de données 43
2.4.5 Registres 43
2.4.6 Instructions 44
2.4.7 Exemple de compilation 45

2.4.8 Ramasse-miettes pour Prolog 47
2.5 Résumé du chapitre 47

3 Parallélisme en Programmation Logique 49
3.1 Sources de parallélisme en programmation logique 49
3.1.1 Parallélisme OU 50
3.1.2 Parallélisme ET 51
3.1.3 Autres sources de parallélisme en programmation logique . 54
3.2 Systèmes parallèles logiques gardés 54
3.2.1 Introduction aux systèmes gardés 54
3.2.2 Syntaxe et sémantique des langages gardés 55
3.2.3 Sémantique informelle 55
3.2.4 Problèmes posés par ces langages 56
3.2.5 Utilisation des langages gardés 57
3.2.6 Implémentations des langages gardés 58
3.3 Modèles "théoriques" 58
3.3.1 COALA 59
3.3.2 Le modèle ET-OU 59
3.3.3 MIPAP 60
3.3.4 REDUCE-OR Process Model 60
3.4 Systèmes multiséquentiels 61
3.4.1 Caractéristiques communes aux systèmes multiséquentiels . 61
3.4.2 Parallélisme OU : gestion des résolvantes multiples 62
3.4.3 Parallélisme OU : contrôle de l'exécution 66
3.4.4 Systèmes OU-parallèles 67
3.4.5 Parallélisme ET-restreint 73
3.4.6 Systèmes ET-parallèles 76
3.4.7 Modèles globaux 76
3.5 Conclusion : Quel parallélisme utiliser? 78

4 Présentation du projet PEPSys 81
4.1 Généralités sur le projet PEPSys 81
4.2 Analyses de programmes Prolog existants 82
4.2.1 Analyses statiques 82
4.2.2 Analyses dynamiques 82
4.3 Le langage PEPSys 82
4.3.1 Modularité 83
4.3.2 Déclarations de propriétés 84
4.3.3 Parallélisme ET 86
4.3.4 Validation du langage PEPSys 86
4.4 Le modèle de calcul de PEPSys 89
4.4.1 Combinaison des parallélismes OU et ET avec l'exécution séquentielle et le retour arrière 89

4.4.2 Partage des environnements: gestion des liaisons pour le parallélisme OU 93
4.4.3 Comparaison du parallélisme OU de PEPSys avec les modèles SRI et Kabu-Wake 97
4.4.4 Gestion des liaisons : combinaison des parallélisme OU et ET 98
4.5 Machine Abstraite, Implémentation parallèle 100
4.6 Simulateur de PEPSys 100
4.7 Ecriture d'applications parallèles. Analyse du parallélisme 102
4.8 Résumé du chapitre 104

II Techniques d'implémentation 105

5 Machine Abstraite PEPSys 107
5.1 Généralités 107
5.2 Contrôle 108
5.2.1 Extensions au contrôle de la *WAM* 108
5.2.2 Contrôle du parallélisme OU 109
5.2.3 Contrôle du parallélisme ET 114
5.2.4 Comportement d'un processus ET-parallèle 116
5.2.5 Prédicats une-solution 117
5.3 Gestion des liaisons 118
5.3.1 Objets manipulés 119
5.3.2 Algorithme de liaison 119
5.3.3 Algorithme de déréférencement 119
5.3.4 Extension de l'unification séquentielle 120
5.3.5 Création d'objets non-locaux 122
5.4 Compilateur PEPSys 123
5.4.1 Présentation générale 123
5.4.2 Langage reconnu par le compilateur 125
5.4.3 Analyses statiques 126
5.4.4 Assemblage, édition de liens 127
5.5 Résumé du chapitre 127

6 Implémentation de PEPSys sur MX500 129
6.1 Motivations 129
6.2 Choix d'un multiprocesseur 130
6.2.1 Classification des multiprocesseurs MIMD 130
6.2.2 Choix du MX500 132
6.2.3 Configuration du MX500 utilisé 133
6.3 Architecture du système parallèle 133
6.3.1 Processeurs virtuels 133
6.3.2 Implémentation de la machine abstraite PEPSys 133
6.3.3 Noyau d'ordonnancement 135

6.4 Implémentation de la Machine Abstraite PEPSys 135
6.4.1 Données 135
6.4.2 Emulateur 140
6.4.3 Déréferencement 140
6.4.4 Unification 141
6.4.5 Instructions de contrôle 141
6.4.6 Prédicats une-solution 141
6.4.7 Implémentation du parallélisme ET déterministe 142
6.5 Ordonnancement 142
6.5.1 Ordonnanceur originel 143
6.5.2 Autres stratégies d'ordonnancement 144
6.6 Mise au point 146
6.6.1 pdbx 146
6.6.2 Debogueur PEPSys 146
6.6.3 Instant Replay 147
6.6.4 Mise au point systématique 148
6.7 Prédicats prédéfinis 149
6.7.1 Implémentation du *not* 149
6.7.2 Implémentation du *oneof* 150
6.7.3 Bagof 150
6.8 Extensions 151
6.8.1 Connection à un système Prolog séquentiel 151
6.8.2 Implémentation de CHIP 152
6.9 Résumé du chapitre 153

III Evaluation 155

7 Stratégie d'évaluation de l'implémentation PEPSys 157
7.1 Principaux paramètres mesurés 157
7.1.1 Performances 157
7.1.2 Mesures du modèle de calcul: déréférencement non-local et recherche dans les *hash-windows* 158
7.1.3 Activité des processeurs 163
7.1.4 Autres mesures liées au parallélisme 164
7.1.5 Consommation mémoire 166
7.2 Systèmes utilisés 166
7.2.1 Enchaînement des mesures, exploitation des résultats 167
7.2.2 Discussion: utilisation de l'*Instant replay* pour les mesures . 167
7.3 Conditions expérimentales 168
7.3.1 Influence du système d'exploitation 168
7.3.2 Exécutions répétées 168
7.3.3 Influence du compilateur 169
7.3.4 Paramètres du système 169

7.4 Programmes de test 169
7.4.1 Programmes OU-parallèles toutes solutions 170
7.4.2 Programmes OU-parallèles une solution 172
7.4.3 Programmes ET parallèles déterministes 173
7.5 Résumé du chapitre 174

8 Résultats expérimentaux **175**
8.1 Introduction 175
8.2 Performances 175
8.2.1 Exécution séquentielle 175
8.2.2 Programmes OU-parallèles toutes solutions 176
8.2.3 Programmes une-solution 182
8.2.4 Programmes ET-parallèles 184
8.3 Evaluation du modèle de calcul 186
8.3.1 Déréférencements non-locaux 187
8.3.2 Recherche dans les *hash-windows* 188
8.3.3 Evaluation du modèle de calcul 191
8.4 Activité des processeurs 192
8.4.1 Nombre de processus 192
8.4.2 Granularité des processus 192
8.4.3 Pourcentage du calcul effectué par les processus courts . . . 193
8.4.4 Oisiveté des processeurs 194
8.5 Autres mesures liées au parallélisme 195
8.5.1 Nœuds parallèles utilisés séquentiellement 195
8.5.2 Surcoût provoqué par l'utilisation séquentielle des nœuds parallèles 195
8.5.3 Valeur maximale de l'*OBL* 196
8.5.4 Compétition 196
8.6 Consommation mémoire 197
8.7 Conclusion du chapitre 199

9 Conclusion **201**
9.1 Bilan 201
9.2 Comparaison avec d'autres approches 202
9.3 Perspectives 203
9.4 Le parallélisme en programmation logique est-il rentable? 204

A Lexique et abbréviations **207**

B Tableaux de mesures détaillés **209**

Bibliographie **215**

Liste des Figures

2.1 Etat de l'interprète après deux unifications 38
2.2 Problème des pointeurs fantômes 40
2.3 Différentes techniques d'exécution de la *WAM* 42
2.4 Exemple de compilation Prolog 46

3.1 Exemple de trou noir dans la pile, en parallélisme OU 63
3.2 Le programme des N reines 65
3.3 Liaisons multiples dans le modèle SRI 70
3.4 Graphes d'exécutions statiques 74
3.5 Compilation du parallélisme ET en parallélisme OU 77

4.1 Le programme des N reines écrit en PEPSys 87
4.2 Le programme quicksort écrit en PEPSys 88
4.3 Liaisons multiples dans le modèle PEPSys 94
4.4 Liaisons superficielles et liaisons profondes dans PEPSys 96
4.5 Utilisation des *join-cells* dans les produits croisés 99
4.6 Architecture multi-grappe simulée dans PEPSys 101
4.7 Analyse abstraite du parallélisme du programme *TInAS* 103

5.1 Compilation d'un prédicat OU-parallèle 111
5.2 Structures de données utilisées pour le parallélisme OU 113
5.3 Compilation d'un prédicat ET-parallèle 115
5.4 Compilation d'un prédicat une-solution 117
5.5 Déréférencement non-local 120
5.6 Unification d'une liste non-locale et d'une liste 121
5.7 Création d'objets non-locaux 123
5.8 Exemple de typage du registre pointeur d'environnement **E** 124

6.1 Piles locales et globales de l'implémentation 134
6.2 Arborescence des structures de contrôle utilisées pour la recherche de travail dans le cas du parallélisme OU 136
6.3 Format d'un objet local de pile (locale ou globale) 137
6.4 Format d'un objet non-local de pile (locale ou globale) 138
6.5 Format d'un élément de *hash-window* 139

7.1 Mesure du surcoût provenant de la recherche dans les *hash-windows* 162

8.1 Facteurs d'accélération des programmes OU-parallèles toutes solutions 178
8.2 Facteurs d'accélération des programmes OU-parallèles toutes solutions relativement à un système Prolog efficace 179
8.3 Comparaison entre les temps d'exécution des programmes ET-parallèles déterministes 186
8.4 Pourcentages de déréférencements non locaux et dans les *hash-windows* 188
8.5 Pourcentage des longueurs des chaines de *hash-windows* 190
8.6 Rapports entre l'espace de pile consommé en parallèle et séquentiellement. Influence des trous noirs 198
8.7 Influence de la taille des *hash-windows* sur la consommation mémoire 199

Liste des Tables

3.1 Performances du système Aurora 0.0-Manchester Scheduler, sur Sequent Balance 8000 72

8.1 Performances des programmes OU-parallèles toutes solutions . . . 177
8.2 Comparaison entre SICStus et PEPSys exécuté en parallèle 180
8.3 Résultats des programmes OU-parallèles une-solution 183
8.4 Temps d'exécution des programmes ET-parallèles 185
8.5 Pourcentage du calcul effectué par les processus courts 193
8.6 Pourcentage d'oisiveté durant l'exécution 194

B.1 Résultats détaillés du programme *farmer* 212
B.2 Résultats détaillés du programme *hamilton* 213

Partie I

Contexte

- les systèmes experts se programment relativement facilement en Prolog dont le système est déjà un moteur d'inférence, le programme jouant le rôle de base de règles.

- la ressemblance entre les clauses de Horn et les relations fait de Prolog un bon langage d'accès aux bases de données pour peu qu'une "connection" soit établie avec un système de gestion de bases de données [Bocca et al. 86].

- l'exécution d'un programme logique se traduisant par le parcours d'un arbre de réfutation, ce type de langage est bien adapté à l'expression de problèmes d'exploration d'arbre. L'extension du système logique par des **contraintes** [Dincbas et al. 88] [Colmerauer 87] [Jaffar 87] accroit de façon spectaculaire l'efficacité d'un système logique pour la résolution de ce type de problème.

On peut trouver dans la littérature de nombreux exemples d'utilisation de Prolog pour la programmation d'applications. [Reintjes 88] décrit une expérience de réalisation de système d'aide à la conception de circuits imprimés. Le système réalisé remplace un système écrit en C qui était devenu non maintenable. Il comporte 10.000 lignes de Prolog contre 100.000 lignes à la version précédente. L'efficacité du nouveau système est comparable à celle de l'ancien pour certaines opérations et jusqu'à dix fois plus lente pour d'autres. Le rapport d'efficacité de 1 à 10 entre un programme en C et en Prolog est d'ailleurs classique. [Paaki 88] compare Prolog et Pascal pour la réalisation d'un compilateur de taille modeste. Le rapport de performances entre l'implémentation du compilateur en Pascal et en Prolog n'est ici que de cinq, correspondant à un rapport de deux entre les tailles des implémentations. Il est à noter que le système Prolog utilisé dans cette expérience est un interprète. L'utilisation d'un système Prolog compilé réduirait le facteur de performances entre les deux implémentations à une valeur inférieure à deux.

Les systèmes Prolog actuels permettent donc de réaliser rapidement des logiciels n'ayant pas de gros problèmes de performances. Lorsque le système Prolog considéré comporte des extensions telles que les contraintes, le gain de productivité peut devenir spectaculaire sans qu'il se fasse au détriment de l'efficacité du programme. De plus, les progrès rapides enregistrés dans les performances des machines ouvrent à la programmation logique des domaines d'application de plus en plus larges.

1.2 Le parallélisme dans la programmation logique

L'exécution d'un programme logique se traduit par l'exploration d'un arbre ET-OU, appelé arbre de réfutation. Les deux principales sources de parallélisme en programmation logique sont le parallélisme OU et le parallélisme ET, consistant à explorer en parallèle des sous arbres connectés par un nœud OU (parallélisme OU) ou par un nœud ET (parallélisme ET). Une variante du parallélisme ET, appelé

parallélisme de flot, est également exploitable. Les systèmes logiques parallèles existants se divisent en deux grandes classes appelées respectivement systèmes parallèles logiques gardés et systèmes multiséquentiels [Chassin, Codognet et al. 89]. Les premiers modifient de façon importante la sémantique de la programmation en logique ; ils exploitent essentiellement le parallélisme de flot et sont destinés à être les langages de programmation système des multiprocesseurs de demain. Les systèmes multiséquentiels restent aussi proches que possible des langages de programmation logique comme Prolog. Ces systèmes visent à ne pas créer plus de parallélisme que de ressources disponibles. Dans le pire des cas leur efficacité doit être proche de celle des systèmes séquentiels les plus efficaces, dont ils adoptent, autant que possible, les mécanismes d'exécution. Ils exploitent le parallélisme ET ou le parallélisme OU, parfois une combinaison des deux.

1.3 Le projet PEPSys

Le contexte dans lequel cette recherche a pris place est le projet PEPSys [Syre et al. 88] [Baron, Chassin et al. 88], Parallel ECRC Prolog System, mené depuis 1985 à ECRC (European Computer-Industry Research Centre). Le projet PEPSys vise à l'exploration de l'ensemble des problèmes que pose la parallélisation de la programmation logique, en vue d'améliorer son efficacité. Le projet PEPSys a retenu l'approche multiséquentielle afin d'obtenir des gains de performances sur les multiprocesseurs existants et de demeurer dans les domaines d'applications classiques de la programmation logique.

Le projet PEPSys a successivement défini un langage [Ratcliffe et Syre 87], sur-ensemble de Prolog, pour la programmation parallèle, ainsi qu'un modèle de calcul adapté à ce langage [Westphal et al. 87]. Le langage et le modèle de calcul exploitent le parallélisme OU, le parallélisme ET indépendant, ainsi que leur combinaison avec l'exécution séquentielle et le retour-arrière. Le modèle de calcul définit une technique originale de partage des cellules de variables des piles par les processus parallèles. Cette technique, utilisant des *hash-windows* et des compteurs appelés *OBL* pour dater les liaisons de variables, suppose une bonne localité de référence pour éviter le parcours de longues chaînes de références, dont la longueur n'est en principe pas bornée.

La *WAM (Warren Abstract Machine)*, standard pour l'implémentation efficace de Prolog, a ensuite été adaptée à l'implantation du langage et du modèle de calcul [Robert 88] [Chassin et Robert 88]. Le projet a ensuite été validé par une implémentation sur un multiprocesseur à mémoire partagée [Chassin et al. 88] et par un simulateur d'une architecture extensible [Baron et al 88], tous deux basés sur la machine abstraite définie à partir de la *WAM*.

Seuls les parallélisme OU et ET déterministe[1] ont été implémentés et testés à l'heure actuelle ; la combinaison des parallélismes OU et ET définie par le modèle

[1] Le programmeur ou le compilateur indiquent au système d'exécution que chaque branche ET-parallèle ne produit qu'une seule solution.

de calcul, très complexe, reste à valider. L'implémentation parallèle sur multiprocesseur constitue l'un des premiers systèmes logique OU-parallèle permettant d'obtenir des gains de performances importants relativement à des systèmes Prolog efficaces [Chassin 89]. Les résultats de la simulation confirment et étendent ceux de l'implémentation [Chassin, Baron et al. 89]. Ils indiquent que seuls des problèmes de taille importante sont susceptibles de tirer parti de machines comportant un grand nombre de processeurs. Enfin, l'implémentation parallèle de PEPSys a été étendue pour combiner le parallélisme OU avec une partie des fonctionnalités du système de programmation logique avec contraintes *CHIP*, le maintien de cohérence sur les domaines finis [Van Hentenryck 89]. Cette combinaison obtient des gains de performances importants, - parfois superlinéaires -, relativement au système *CHIP* séquentiel, pour la résolution de problèmes de très grande taille, montrant ainsi que le parallélisme OU est compatible avec la programmation logique avec contraintes.

1.4 Contribution de l'auteur au projet PEPSys

La thèse défendue dans ce document est qu'il est possible de réaliser, sur les multiprocesseurs existants actuellement, des systèmes logiques parallèles plus efficaces que les systèmes Prolog les plus performants, à condition d'utiliser les mêmes techniques d'implémentation. Ce postulat est également à la base des travaux regroupés sous le nom de "systèmes multiséquentiels". Le corollaire de la proposition précédente est que le modèle de calcul de PEPSys permet de coupler l'utilisation des techniques performantes d'implémentation séquentielle avec la mise en œuvre du parallélisme.

L'auteur a rejoint le projet PEPSys alors que le langage et le modèle de calcul venaient d'être définis. Son premier travail fut d'étudier les multiprocesseurs disponibles pour une implémentation parallèle du modèle de calcul. Cette étude a abouti au choix du multiprocesseur à mémoire partagée MX500. Partageant la conviction que l'implémentation et la simulation devaient être basées sur la *WAM*, l'auteur a ensuite collaboré à la définition de la machine abstraite PEPSys en compagnie de Philippe Robert. Il a ensuite transformé le compilateur ICM3 en compilateur PEPSys. Il a alors implémenté une grande partie du système parallèle PEPSys sur MX500, se chargeant plus particulièrement de l'émulateur de la machine abstraite. Postérieurement il a pris en charge l'ensemble de l'implémentation pour expérimenter plusieurs ordonnanceurs. L'auteur a alors réalisé le système de mesures et effectué de nombreuses campagnes de mesures couplées à des phases d'optimisation. En plus de ces activités techniques, l'auteur a pris une part active à la rédaction des articles publiés sur le projet.

L'implémentation parallèle de PEPSys est la principale réalisation qui a permis de valider de l'approche multiséquentielle adoptée par le projet PEPSys. Le système parallèle PEPSys s'exécute séquentiellement avec une efficacité proche des meilleurs systèmes Prolog. Son exécution parallèle procure de réels gains de

performances pour les programmes comportant du parallélisme. Les nombreuses mesures effectuées permettent de valider certaines hypothèses du modèle de calcul et en particulier l'hypothèse que l'utilisation de *hash-windows* ne crée pas de longues chaines de références susceptibles de compromettre l'efficacité du système.

Les résultats des mesures analysent en détail le comportement dynamique du système parallèle, fournissant ainsi une importante source d'informations pour l'élaboration des versions ultérieures du système PEPSys.

1.5 Plan de la thèse

Pour terminer cette introduction nous donnons le plan de la thèse ainsi qu'un résumé des chapitres qui la composent. La thèse est divisée en trois grandes parties. La première partie introduit le contexte dans lequel le travail de la thèse a pris place. La deuxième partie de la thèse décrit les techniques d'implémentation employées pour PEPSys et leur mise en œuvre sur un multiprocesseur à mémoire partagée. La troisième partie est consacrée à l'évaluation de l'implémentation de PEPSys et plus généralement à la validation des principaux concepts de ce projet. Le résumé des chapitres de la thèse est le suivant :

Chapitre 2. Rappels sur la programmation logique et Prolog

Ce chapitre définit les notions utilisées en programmation logique et donne les principaux résultats qui la fondent. Le langage Prolog, principal langage utilisé pour la programmation logique, est ensuite brièvement présenté. On aborde alors les techniques d'implémentation de Prolog, en partant de la description d'un interprète simple du langage qui permet de définir l'ensemble des structures de données utilisées habituellement. Il devient possible de montrer comment la *WAM* optimise les techniques d'implémentation de Prolog et d'en donner les principales caractéristiques.

Chapitre 3. Parallélisme en programmation logique

Au cours de ce chapitre, on montre pourquoi la programmation logique se prête particulièrement bien à l'exécution parallèle. Les différentes types de parallélisme exploitables sont ensuite examinés, en examinant les problèmes que pose l'exploitation de chacun d'entre eux. Enfin, une étude bibliographique passe en revue les principaux modèles de programmation en logique parallèle, en insistant particulièrement sur les systèmes multiséquentiels auxquels se rattache le projet PEPSys.

Chapitre 4. Présentation du projet PEPSys

Ce chapitre présente l'ensemble des aspects du projet PEPSys : langage, modèle de calcul, machine abstraite, simulation et implémentation. Ce chapitre conclut la première partie introductive de la thèse.

Chapitre 5. Machine Abstraite PEPSys

Comme tout système multiséquentiel, PEPSys est basé sur les mécanismes d'exécution séquentielle les plus efficaces de la programmation logique. La *WAM*, qui joue le rôle de standard dans ce domaine, a donc été étendue afin de permettre la compilation du langage PEPSys et l'implémentation de son modèle de calcul. Un compilateur du langage PEPSys dans cette machine abstraite a également été réalisé.

Chapitre 6. Implémentation de PEPSys sur un multiprocesseur à mémoire partagée

Pour valider les concepts du projet et obtenir des résultats expérimentaux, une implémentation a été réalisée sur un multiprocesseur Siemens MX500 à mémoire partagée, comportant huit processeurs. Ce chapitre met en relief les problèmes que pose la mise en œuvre efficace de la machine abstraite PEPSys et les solutions qui leur ont été apportées. Les recherches faites pour définir une stratégie optimale d'allocation de travail aux processeurs sont également présentées. Enfin les extensions faites à l'implémentation pour prendre en compte la modularité du langage PEPSys et la programmation avec contraintes sur les domaines finis sont brièvement décrites. Ce chapitre conclut la deuxième partie de la thèse consacrée à l'implémentation de PEPSys sur multiprocesseur.

Chapitre 7. Stratégie d'évaluation de l'implémentation de PEPSys

De nombreuses mesures ont été effectuées sur le système parallèle, afin d'une part d'en améliorer l'efficacité et d'autre part de valider les concepts développés dans le projet PEPSys. Dans ce chapitre, les paramètres significatifs du système PEPSys sont donnés. Les techniques de mesure, conditions expérimentales ainsi que les programmes de test utilisés sont également décrits.

Chapitre 8. Résultats expérimentaux

Le principal résultat expérimental est bien sûr le facteur d'accélération (*speed-up*), rapport entre le temps passé à exécuter séquentiellement un programme et le temps passé à l'exécuter en parallèle. D'une très grande importance également est l'évaluation du modèle de calcul de PEPSys. L'un des points originaux de ce modèle est la gestion des variables logiques dans un environnement OU-parallèle. Celle-ci favorise la création efficace des nouvelles branches parallèles, au détriment de l'opération de déréférencement des variables logiques. Un grand nombre de mesures permettent l'évaluation et la validation de cette solution. Un nombre important d'autres mesures évaluent la granularité du parallélisme effectivement utilisé ainsi que les différentes sources de surcoût (*overhead*) en temps et en mémoire.

Chapitre 9. Conclusion

De façon très classique, la conclusion de cette thèse en résume les principaux résultats et dresse les perspectives de la recherche dans le domaine.

Chapitre 2

La programmation logique, Prolog et la WAM

Le but de ce chapitre est de fournir les rappels nécessaires à la compréhension du reste de la thèse. Ces rappels couvrent les principaux concepts de programmation logique, introduisent le langage Prolog et donnent les bases de son implémentation.

A la base de la programmation logique se trouve la découverte que la logique a une interprétation procédurale qui en fait un langage de programmation [Kowalski 79]. Les démonstrateurs automatiques de théorèmes de la logique des prédicats du premier ordre, basés sur les travaux de [Robinson 65], peuvent être donc considérés comme des systèmes permettant d'exécuter des programmes logiques. Le cœur du langage Prolog peut être considéré comme une implémentation efficace d'un démonstrateur de théorèmes de la logique du premier ordre restreinte aux clauses de Horn. Un programme écrit en "pur" Prolog est donc un ensemble de formules de la logique des prédicats du premier ordre. Cette base logique étendue par un certain nombre d'opérateurs "non logiques" - entrées-sorties, contrôle, etc. - constitue la base du langage de programmation Prolog.

Les premières implémentations de démonstrateurs de théorèmes de la logique du premier ordre datent des années 60. D'une grande inefficacité, ils consommaient en outre beaucoup de mémoire. Les choix fait pour Prolog ont permis d'améliorer très nettement les performances de ces systèmes. Malgré tout et pour autant qu'il soit possible de les comparer aux implémentations des langages impératifs, les premiers interprètes Prolog étaient toujours inefficaces. Comme cela s'était produit avec le langage Lisp, la réalisation des premiers compilateurs Prolog marqua une avancée significative dans le domaine des performances [Warren 77]. La consommation de ressources mémoire diminua ensuite notablement avec l'optimisation de la récursion terminale [Warren 80]. Plus tard, la *Warren Abstract Machine* ou *WAM* [Warren 83] fit la synthèse des techniques développées précédement et depuis, elle joue le rôle de standard pour l'implémentation des systèmes Prolog efficaces.

2.1 Programmation logique

Les principaux résultats de démonstration automatique sur lesquels sont basés la programmation logique et le langage Prolog sont présentés ici sans démonstration. Le lecteur intéressé pourra se référer à [Lloyd 87] pour une information plus complète sur le domaine.

2.1.1 Quelques définitions de la logique des predicats du premier ordre

Dans les définitions qui suivent, nous utilisons les notations habituelles de Prolog.

Terme

Un terme est soit :

- une variable
- si f est un symbole de fonction et $t_1, ...t_n$ sont des termes, alors $f(t_1, ...t_n)$ est un terme. Une **constante** est un symbole fonctionnel d'arité zéro.

Dans tout ce qui suit on dénotera les variables par des chaines alphanumériques commençant par une majuscule : A, X_1, *Queen* et les symboles fonctionnels par des chaines alphanumériques commençant par une minuscule : a, x_1, *queen*.

Atomes et Formules (bien formées)

- Si p est un symbole de prédicat n-aire et $t_1, ...t_n$ sont des termes, alors $p(t_1, ...t_n)$est une formule, appelée formule atomique ou plus simplement **atome.**
- Si F et G sont des formules, alors :
  - non F : ($\neg$ F),
  - F ou G : (F $\vee$ G),
  - F et G : (F $\wedge$ G),
  - F implique G : (F $\rightarrow$ G),
  - F équivalent à G : (F $\leftrightarrow$ G)

  en sont également.
- Si F est une formule, alors :
  - il existe F : ($\exists$ X)F
  - pour tout F : ($\forall$ X)F

  en sont aussi.

Littéral

Un littéral est un atome (littéral positif) ou sa négation (littéral négatif).

Clauses, clauses de Horn

Une clause est une formule de la forme :

$$\forall X_1 ... \forall X_s (L_1 \vee ... \vee L_m)$$

dans laquelle chaque L_i est un littéral et $X_1 ... X_s$ sont les variables apparaissant dans $L_1 \vee ... \vee L_m$.

On simplifie la notation des clauses de la façon suivante :

$$\forall X_1 ... \forall X_s (A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_n)$$

sera notée :

$$A_1, ..., A_k \leftarrow B_1, ..., B_n$$

Une clause définie possède au plus un littéral positif :

$$A \leftarrow B_1, ..., B_n$$

A est appelée la tête et $B_1 ... B_n$ le corps de la clause. La clause précédente peut se lire : "si $B_1 \wedge ... \wedge B_n$ est vraie, alors A l'est".

Une clause unitaire est une clause définie dont le corps est vide : $A \leftarrow$

Un but est une clause définie dont la tête est vide : $\leftarrow B_1, ..., B_n$. Chacun des B_i est appelé un sous-but du but.

Une clause de Horn est soit une clause définie soit un but.

La clause vide, notée $\square$, est la clause dont la tête et le corps sont vides. Elle marque une contradiction.

Les travaux sur la démonstration automatique ayant donné naissance à la programmation logique portent tous sur des formules sous forme clausale. Il existe un algorithme permettant de mettre sous forme clausale toute formule de la logique des prédicats du premier ordre. Toute clause peut à son tour être transformée en une ou plusieurs clauses de Horn. Aucune des deux transformations ne préserve l'équivalence entre les formules de départ et d'arrivée.

2.1.2 Interprétation et preuve par réfutation

Une interprétation donne un sens à chaque atome d'une formule et permet de l'évaluer à *vrai* ou *faux*. Une formule F est valide si toute interprétation de F s'évalue à vrai. Elle est inconsistante s'il n'existe pas d'interprétation de F qui s'évalue à vrai.

La logique du premier ordre comprend un langage, qui est celui des formules définies précédement, des axiomes, ensemble de formules valides et des règles d'inférences, permettant de dériver une formule valide à partir d'autres formules

valides. La preuve d'une formule est la suite d'étapes établissant sa validité. Il n'existe pas de procédure générale permettant d'établir qu'une formule quelconque est valide.

Une interprétation I est un **modèle** d'une formule F si F s'interprète à vrai relativement à I. Une formule G est une **conséquence logique** d'un ensemble de formules P si G est vraie dans tous les modèles de P. Trouver tous les modèles d'un ensemble de formules est un problème énorme aboutissant à générer un nombre très important de clauses redondantes. L'idée de la preuve par **réfutation** est de ramener ce problème au problème plus simple consistant à montrer que $\neg G$ n'est pas vraie dans aucun modèle de P et donc G est vraie dans tous les modèles de P. Pour prouver que $\neg G$ n'est pas vraie dans aucun modèle, on prouve qu'il est possible d'en dériver la clause vide. Cette stratégie peut être qualifiée de descendante (*top-down*).

Par exemple, pour prouver que :

$$\exists X_1 ... X_p (B_1 ... B_n)$$

est une conséquence d'un ensemble de formules P, on dérive de sa négation :

$$\leftarrow B_1, ..., B_n$$

la clause vide, ce qui assure que la formule initiale est bien une conséquence de P.

2.1.3 Résolution SLD

Dans tout ce qui suit, les formules considérées seront des clauses de Horn.

Unification

Une substitution θ est un **unificateur** de deux littéraux E et F si $E\theta = F\theta$. Un unificateur σ de E et F est le **plus général unificateur** de E et F si et seulement si pour tout unificateur θ de E et F il existe une substitution λ telle que : $\theta = \sigma \circ \lambda$. Le plus général unificateur est unique au renommage près des variables apparaissant dans E et F.

Résolution SLD

Soient C_1 la clause $\leftarrow A_1, ..., A_m, ... A_k$ et C_2 la clause $A \leftarrow B_1, ..., B_q$. G est une **résolvante** de C_1 et C_2 par résolution SLD si :

- A_m est l'atome *sélectionné* de C_1.
- θ est l'unificateur le plus général de A_m et A.
- G est le but $\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1} ..., A_k)\theta$.

On dit alors que G dérive de C_1 et C_2.

La *résolution SLD*, résolution Linéaire avec fonction de Sélection pour les clauses Définies, est la méthode de preuve par réfutation utilisée en programmation logique. Etant donnés un programme, ensemble de clauses de têtes non vides considérées comme axiomes et un but unique, négation d'une formule à prouver, la résolution SLD consiste à dériver du but la clause vide :

- on prend le but à prouver comme résolvante initiale G_0.
- on dérive G_{i+1} de G_i en construisant la résolvante de G_i et d'une clause du programme.

La résolution SLD est *linéaire* car la résolvante construite à une étape de la résolution est l'une des deux clauses utilisées à l'étape suivante, l'autre clause étant choisie dans le programme. La résolution SLD possède quelques propriétés très importantes :

- elle est **saine** (*sound*), c'est à dire qu'elle ne produit que des résultats corrects.
- elle est **complète** pour les clauses de Horn, c'est à dire qu'elle produit toutes les solutions possibles, et ce, quelle que soit la règle de sélection utilisée.

La plupart des implémentations de Prolog ne sont pas saines car elles implémentent de façon incorrecte l'algorithme d'unification pour des raisons d'efficacité. Avant d'unifier une variable et un terme, il est en effet nécessaire de vérifier que la variable n'apparaît pas dans le terme. Cette vérification, connue sous le nom d'*occur-check*, est le plus souvent omise dans les systèmes Prolog.

Par ailleurs, étant donnée une règle de sélection, on peut représenter une réfutation SLD par un arbre ET-OU. La propriété de complétude de la résolution SLD signifie que tous les nœuds solutions font partie de cet arbre. Cependant, l'arbre de réfutation comporte éventuellement des branches infinies. Une implémentation de la résolution SLD, dans laquelle la **stratégie d'exploration** de cet arbre est **non équitable** (*unfair*), peut alors remettre en cause la complétude de la résolution SLD. C'est le cas de Prolog qui adopte une stratégie en **profondeur d'abord** et peut ne pas trouver une solution, en cas de branche infinie.

2.1.4 Programmation logique

Un programme comporte un nombre quelconque de clauses définies de la forme :

$$A \leftarrow B_1, ..., B_n$$

ou

$$A \leftarrow$$

Comme nous l'avons vu, la réfutation permet de prouver qu'un but :

$$\exists X_1 ... X_p (B_1 ... B_n)$$

est une conséquence logique d'un programme P. Le résultat de la réfutation est **oui** ou **non** suivant que la formule est une conséquence logique du programme ou non. Le résultat attendu d'un programme logique est plutôt la substitution des variables du but résultant de la réfutation.

On peut interpréter les clauses de Horn du programme logique de façon procédurale [Kowalski 79]. Une clause $P \leftarrow Q_1, ..., Q_n$ est considérée comme une procédure qui appelle les procédures $Q_1, ..., Q_n$. Ceci restreint le type de fonction de sélection utilisable par la résolution SLD pour le choix d'un atome de la résolvante. Considérons en effet la résolvante :

$$L_1, ..., L_{i-1}, P, L_{i+1}, ..., L_n$$

et la clause $P \leftarrow Q_1, ..., Q_n$. En unifiant l'atome P et la tête de clause, on obtient une nouvelle résolvante :

$$L_1, ..., L_{i-1}, Q_1, ..., Q_n, L_{i+1}, ..., L_n$$

L'interprétation procédurale des clauses du programme implique que l'on résolve l'ensemble des atomes $Q_1, ..., Q_n$ avant l'un quelconque des L_j. C'est le cas en Prolog où le premier atome, par ordre lexicographique, apparaissant dans la résolvante, est sélectionné.

Dans la suite de la thèse, on appellera **procédure** un paquet de clauses définies ayant le même symbole de prédicat en tête. On désignera également un paquet de clauses par **prédicat**.

Remarque : il existe des formes de programmation logique n'impliquant pas d'interprétation procédurale. C'est le cas des langages logiques qui supportent les coroutines.

2.2 Prolog

De même que pour la section précédente, il n'est pas dans notre intention de donner une présentation exhaustive du langage Prolog, mais plutôt de rappeler les points essentiels utiles à la compréhension du reste de la thèse. Il est possible de trouver une description du langage dans [Clocksin et Mellish 81]. On peut également trouver une bonne introduction à la programmation en Prolog dans [Sterling et Shapiro 86].

2.2.1 Prolog pur

Comme nous l'avons vu dans la section précédente, Prolog est une implémentation efficace de la résolution SLD dans laquelle :

- La *fonction de sélection* consiste à choisir le premier atome (lexicographiquement, c'est-à-dire le plus à gauche) de la résolvante.

- La *stratégie d'exploration* de l'arbre de réfutation procède en profondeur d'abord. Les nœuds OU de l'arbre de réfutation correspondent à des prédicats définis par plusieurs clauses. A la rencontre d'un nœud OU de l'arbre de réfutation, la nouvelle résolvante est construite en utilisant la première clause, lexicographiquement, dont la tête est unifiable à l'atome sélectionné. Les autres choix possibles seront explorés ultérieurement par **retour arrière**.

La stratégie d'exploration en profondeur d'abord présente l'avantage énorme, par rapport à d'autres stratégies plus équitables, de nécessiter relativement peu de mémoire. En effet si l'on utilise une pile pour implémenter l'exploration de l'arbre de réfutation, chaque retour arrière permet de libérer de l'espace en sommet de pile. Par contre, ainsi que nous l'avons remarqué précédemment, cette stratégie n'étant pas équitable, elle fait **perdre à Prolog la complétude de la résolution SLD**.

2.2.2 Contrôle et *cut*

L'interprétation procédurale de Prolog assimile la résolution du sous-but (atome) courant à un appel de procédure. La première clause de cette "procédure" est alors sélectionnée, ce mécanisme de contrôle pouvant être déterminé statiquement. Par contre le retour arrière, qui consiste à rendre le contrôle à l'aternative la plus récente, n'a pas d'interprétation procédurale et donne à Prolog sa propriété de langage **non-déterministe**. Le non-déterminisme de Prolog signifie qu'il n'est pas possible de déterminer statiquement quelle clause conduit à une solution.

Le *cut*, noté habituellement "!" (il est parfois appelé *slash* et alors noté "/") a d'abord été introduit pour des raisons d'efficacité. Rappelons qu'il est utilisé comme un sous-but dans une clause, qu'il réussit toujours, et qu'il a pour effet d'écarter toutes les alternatives rencontrées depuis l'invocation de la procédure le contenant. Il est utilisé lorsque le programmeur peut déterminer à l'écriture de son programme que le "succès" d'une clause ou d'une partie de ses sous-buts rend inutile l'exploration des autres clauses de la procédure. Le rôle du *cut* est d'élaguer l'arbre de réfutation, ce qui permet des gains d'efficacité ainsi que d'importantes récupérations mémoire dans la pile d'évaluation lors du prochain retour-arrière.

L'utilisation du *cut* en programmation logique a donné et donne encore lieu à de nombreux débats, souvent passionnés. Le *cut* ne change pas la sémantique déclarative du programme où il est employé. Par contre, lorsqu'il est utilisé de façon *non sûre* - c'est-à-dire pour couper des branches de l'arbre de réfutation menant à une solution -, il conduit à une réponse incorrecte. Une utilisation particulièrement fréquente du *cut* est pour programmer la négation (*negation as failure*) ou des structures de contrôle telles que *si .... alors .. sinon*. L'utilisation du *cut* pour améliorer l'efficacité des programmes conduit à des programmes **déclarativement**

incorrects (voir [Lloyd 87]) comme :

$$max(X, Y, Y) \leftarrow X <= Y, !$$

$$max(X, Y, X)$$

Ce prédicat ne retourne des résultats corrects qu'en raison de la stratégie en profondeur d'abord de Prolog qui exécute les deux clauses dans l'ordre lexicographique. Certaines implémentations Prolog [Thom 86] proposent des constructions permettant de limiter l'usage du *cut* afin d'améliorer la sémantique déclarative des programmes.

2.2.3 Prédicats prédéfinis

De nombreux *prédicats systèmes* ou *prédéfinis* ont été rajoutés à Prolog pour en faire un langage de programmation pratiquement utilisable. Le nombre et la nature de ces prédicats prédéfinis dépendent du système Prolog considéré. On trouve habituellement :

- les prédicats d'entrées-sorties.
- les prédicats "fonctionnels" implémentant les opérations arithmétiques.
- les prédicats permettant l'ajout et le retrait de clauses de la base (*assert et retract*). Ces prédicats permettent de simuler l'affectation en Prolog. Leur abus conduit à des programmes très inefficaces.
- les prédicats "d'ordre supérieur" - c'est-à-dire ayant d'autres prédicats comme arguments - tels que *metacall, findall, bagof et setof* ayant respectivement pour effet d'appeler un prédicat ou de calculer, sous diverses formes, l'ensemble de ses solutions.

2.2.4 Efficacité

Nous avons dit que les choix faits dans Prolog en faisaient une implémentation particulièrement efficace de la résolution SLD au détriment de sa complétude. Par contre, comparé aux langages algorithmiques, Prolog apparaît plutôt inefficace. Les travaux visant à améliorer l'efficacité des systèmes Prolog se sont développés dans différentes directions, qui ne sont d'ailleurs nullement incompatibles :

Compilation : les premiers systèmes Prolog étaient interprétés. Le premier compilateur réalisé en 1977 par David Warren a marqué une étape décisive dans la recherche de meilleures performances pour les systèmes Prolog. Actuellement, la machine abstraite pour la compilation Prolog définie en 1983 par le même Warren [Warren 83] est le standard utilisé par les systèmes Prolog commerciaux et les progrès dans ce domaine semblent se stabiliser.

Matériel spécialisé : l'implémentation de la machine de Warren par du matériel spécialisé permet des gains de performances spectaculaires par rapport aux implémentations sur processeurs généraux [Benker et al. 89].

Contraintes : la programmation logique avec contraintes [Dincbas et al. 88], [Colmerauer 87], [Jaffar 87], permet de réduire considérablement la taille de l'arbre de recherche pour nombre de problèmes combinatoires.

Parallélisme : il s'agit ici de tirer parti de la puissance de calcul offerte par les multiprocesseurs qui sont de plus en plus présents sur le marché. Ce thème sera largement développé dans les chapitres suivants.

Unité de mesure

L'unité de mesure habituellement utilisée pour comparer les performances des systèmes Prolog est le Klip (prononcer "Clip") ou Kilo-inférence par seconde, une inférence étant l'unification d'un sous but d'une résolvante avec une tête de clause. La mesure des performances d'un système dépend du programme utilisé et le programme de test employé classiquement est le *naive-reverse* d'une liste de 30 éléments. L'intérêt commercial de ce programme de test est qu'il se prête à une compilation optimisée et permet d'obtenir une mesure élevée. Cependant, étant complètement déterministe, il ne teste qu'imparfaitement les systèmes Prolog. La compilation de Prolog en utilisant la *WAM* permet d'obtenir des vitesses de l'ordre de 20 Klips/Mips. Ce dernier chiffre reposant sur une unité de mesure, le Mips (Million d'instructions par seconde), dont la définition est sujette à caution, il n'est fourni ici que pour donner un ordre de grandeur.

Syntaxe

En l'absence de normalisation, plusieurs syntaxes ont été développées pour Prolog. Dans la suite du rapport, nous emploierons la plus communément utilisée, celle dite "d'Edimbourg" telle qu'elle est définie dans [Clocksin et Mellish 81]. En particulier, le symbole ← sera fréquemment remplacé par ": –".

2.3 Interprétation de Prolog

Cette section décrit le fonctionnement d'un interprète Prolog standard sur lequel est fondée la *WAM*, objet de la section suivante. Cette description utilise le formalisme de [van Emden 84]. Pour plus de détails on peut consulter également [Warren 77] [Chassin 86] et [Boizumault 88].

2.3.1 Représentation de l'état courant

L'état courant d'un interprète Prolog est le sous-arbre de l'arbre de réfutation contenant l'ensemble des nœuds parcourus jusqu'au nœud courant. En raison de la

stratégie d'exploration en profondeur d'abord utilisée par Prolog, on le représente par une pile.[1] On ne recopie pas chaque résolvante dans la pile mais plutôt les substitutions qu'il faut appliquer aux variables des sous-buts qui la composent. Selon la théorie de la résolution, avant de tenter une unification entre le sous-but courant et une tête de clause, on renomme les variables de telle sorte que le sous-but et la tête de clause n'aient aucune variable en commun. Dans les implémentations Prolog les variables logiques sont identifiées par leurs adresses et le renommage d'une variable est simulé par la création d'une nouvelle cellule de variable sur la pile Prolog. Cette caractéristique se révèle particulièrement importante pour la compréhension des mécanismes de gestion de variables logiques mis en œuvre par les systèmes logiques OU-parallèles multiséquentiels tel que PEPSys.

Un interprète Prolog utilise quatre variables d'état :

1. *APPEL_COURANT* : désigne le code source du sous-but en cours de résolution ; elle permet d'en déduire le sous-but suivant de la même clause.
2. *CLAUSE_COURANTE* : indique la clause dont la tête a été unifiée avec le sous-but courant.
3. *ENVIRON_COURANT* : identifie l'environnement d'appel de *APPEL_COURANT* dans la pile ; il s'agit de l'environnement père du sous-but courant.
4. *RETOUR_COURANT* : désigne l'environnement correspondant au dernier point de retour arrière ; permet de déterminer la clause alternative à celle utilisée au point de retour-arrière.

L'unification du sous-but courant avec une tête de clause est enregistrée dans un environnement de pile qui contient les liaisons (substitutions) effectuées par l'unification ainsi qu'une sauvegarde des quatre variables d'état. Pour expliciter la gestion de la pile et la gestion des variables logiques, nous pouvons prendre en exemple un programme simple :

```
(C1) f( X, g( b ), Z ) :- t( X, Z ), v( X, Z ).
(C2) t( a, h( c ))     :- ...
(C3) t( a, h( d ))     :- ...
(C4)                   :- f( a, g( X ), h( Y )),
                          u( X, Y ).
```

Soient E0, E1 et E2 les environnements d'appel de clauses construits successivement à l'exécution du programme ci-dessus. La construction de ces environnements est schématisée par la figure 2.1. Ses différentes étapes sont :

[1] Toute modification de la stratégie en profondeur d'abord de Prolog introduisant une composante en largeur d'abord, ce qui est le cas avec le parallélisme, perturbe la gestion mémoire en pile par l'introduction de trous noirs (*garbage slots*) et de sous-buts inclus (*trapped goals*) (voir paragraphe 3.4.1 et figure 3.1).

1. Création d'un environnement initial E0 contenant deux liaisons indéfinies pour les variables *X* et *Y* de (C4). La résolvante courante devient *f(a, g(X0), h(Y0)), u(X0, Y0)*

2. Unification de (C4) et de la tête de (C1). On crée un nouvel environnement E1. Dans E0, liaison de *X0* à la constante *b*. Dans E1, liaison de *X1* à *a* et de *Z1* à *h(Y0)*. Sauvegarde dans la pile des trois premières variables d'état qui désignent respectivement les clauses (C4) et (C1) ainsi que l'environnement E0. La résolvante courante devient *t(a, h(Y0)), v(a, h(Y0)), u(b, Y0)*.

3. Unification de *t(a, h(Y0))* et de *t(a, h(c))* ; liaison de *Y0* à la constante *c* ; pas de liaison dans E1 ni dans E2. On sauvegarde dans l'environnement E2 de la pile les variables d'état *APPEL_COURANT*, *CLAUSE_COURANTE* et *ENVIRONNEMENT_COURANT* qui contiennent un pointeur sur respectivement le sous-but *t(X, Z)* de (C1), la clause (C4) et E1. Comme une autre clause est unifiable au sous-but courant, l'environnement E2 est un point de retour arrière. On y sauvegarde donc aussi la valeur précédente de *RETOUR_COURANT*, à savoir le pointeur vide puisqu'il s'agit du premier point de retour arrière.

A ce stade de l'exécution, représenté sur la figure 2.1, les variables d'état de l'interprète ont les valeurs suivantes :

- *APPEL_COURANT* : pointeur sur le sous-but *t(X, Z)*.
- *CLAUSE_COURANTE* : pointeur sur la clause (C2).
- *ENVIRON_COURANT* : pointeur sur E1.
- *RETOUR_COURANT* : pointeur sur E2.

2.3.2 Pile de restauration ou trace

Lors d'un retour arrière, le désempilement de la pile locale est insuffisant pour restituer l'état de l'interprète lors de l'exécution initiale. Des environnements existants à cet instant et donc ne sont pas désempilés par le retour arrière, contiennent en effet des liaisons qui ont été établies ultérieurement et doivent donc être "défaites". C'est le cas dans l'exemple précédent pour la liaison de la variable *Y0* de l'environnement E0 avec la constante *c*, lorsque l'on fait un retour arrière sur le sous-but *t* d'arité deux. Les interprètes Prolog enregistrent les adresses de ces liaisons dans une pile appelée **pile de restauration ou trace**. L'adresse du sommet de la pile de restauration est sauvegardé dans les points de retour arrière. Au cours d'un retour arrière, on défait toutes les liaisons empilées dans la pile de restauration depuis le dernier point de retour arrière avant de désempiler la pile de restauration et la pile Prolog. La figure 2.1 montre l'utilisation de la pile de restauration.

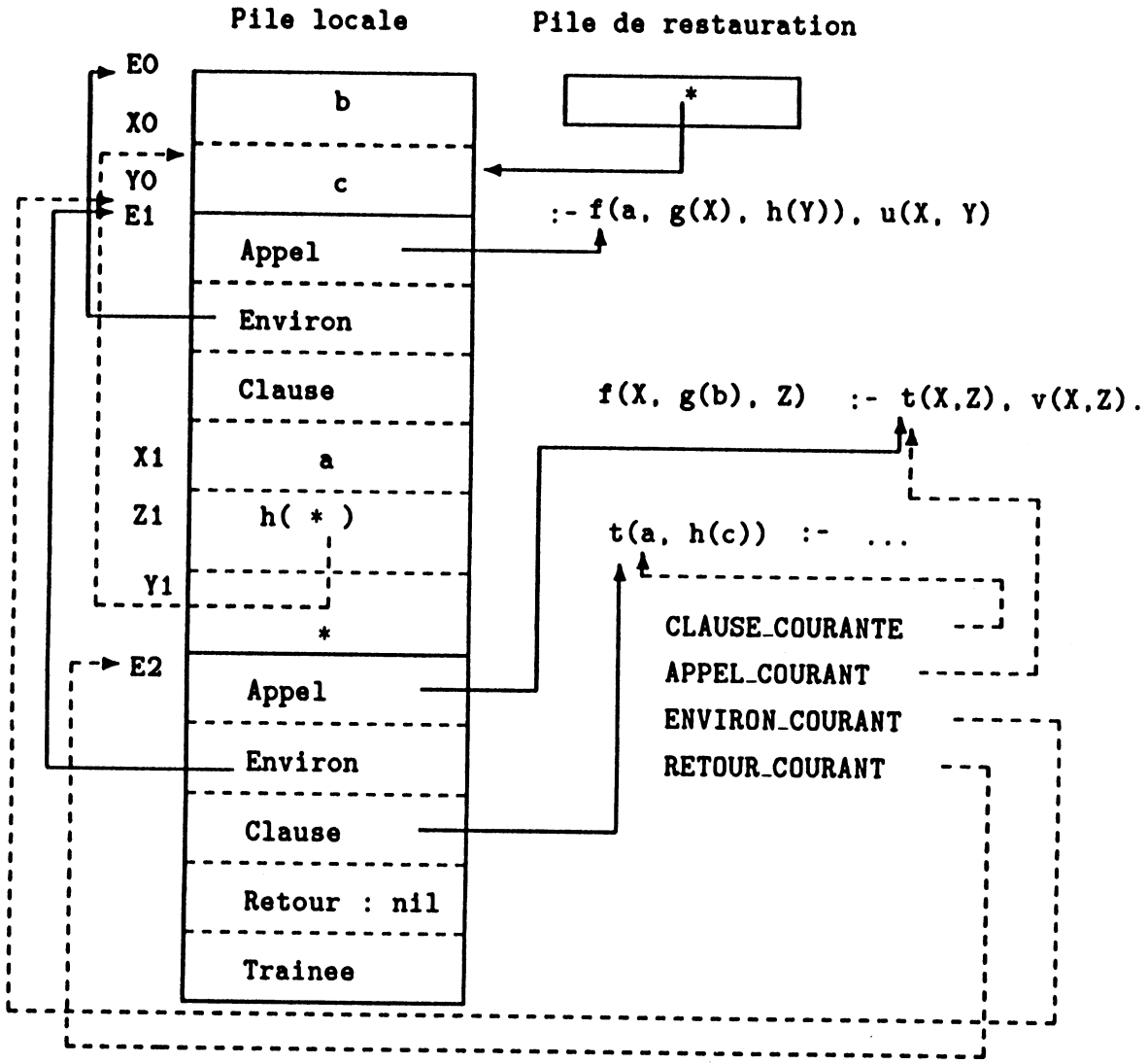

Figure 2.1: Etat de l'interprète après deux unifications

On peut remarquer qu'il n'y a aucune variable liée dans l'environnement d'appel E2. La liaison de *Y0* à *c*, faite à la création de E2, est enregistrée dans la pile de restauration afin de pouvoir être réinitialisée lors du retour arrière.

2.3.3 Pile globale

La gestion mémoire qui a été présentée est très coûteuse puisque l'espace occupé par la pile n'est libéré que lors des retours arrières. Si l'on fait une analogie entre Prolog et un langage de programmation déterministe, il est possible de libérer les environnements de sommet de pile correspondant à la résolution d'un sous-but, lorsque celui-ci est complètement résolu ; c'est le cas lorsque toutes les alternatives possibles pour résoudre ce but ont été essayées. Cette libération n'est possible que si toutes les liaisons entre variables sont orientées de la plus récente vers la plus ancienne. Ce n'est pas toujours le cas, ainsi qu'on peut s'en convaincre avec un exemple simple :

```
(1) f( g( X ), Y) :- h( X, Y ).
(2) h( a, b ).
(3) t( g( a )).
(4)                 :- f( X, Y), t( X ).
```

Après deux étapes de résolution, la pile locale comporte trois environnements E0, E1 et E2 ; seuls les deux premiers E0 et E1 contiennent des liaisons de variables (voir figure 2.2). Le but déterministe *f(X, Y)* étant complètement résolu, on cherche à désempiler les environnements d'appel correspondants E1 et E2 ; c'est possible pour E2 ; ce n'est pas le cas pour E1 puisque *X1* sert à l'évaluation de *X0*, nécessaire à la résolution du sous-but courant *t(X)*.

Ce problème se présente dès qu'une variable apparaît dans une structure, *X* de *g(X)* dans l'exemple ci-dessus, servant à **construire** un résultat. Pour rendre possible la récupération mémoire sur la pile Prolog, il est donc nécessaire d'utiliser une autre pile, appelée **pile globale**. La pile Prolog prend donc le nom de **pile locale**. La pile globale n'est désempilée que lors des retours arrières ; l'adresse du sommet de pile globale doit donc être sauvegardé à chaque point de retour arrière.

2.3.4 Déréférencement, unification et liaison

A la base de l'implémentation de l'unification se trouvent deux opérations basiques qui ont été définies implicitement dans ce qui précède : il s'agit du déréférencement et de la liaison. Par **déréférencement** nous entendons l'opération qui consiste à déterminer si une variable est liée et, si oui, quelle est la valeur de la liaison. L'**unification** est l'opération décrite au paragraphe 2.1.3. Elle a pour résultat de lier certaines variables libres à d'autres objets pouvant également être des variables libres. Dans la plupart des systèmes séquentiels, l'opération de **liaison** est effectuée en écrivant la valeur de la liaison directement dans la cellule de variable qui était libre auparavant. La liaison de deux variables libres entre elles est effectuée en faisant pointer la plus ancienne sur la plus récente[2]. C'est le cas sur la

[2]Cet ordre permet de recupérer l'espace en sommet de pile en évitant les pointeurs fantômes (voir paragraphe 2.3.3 et figure 2.2).

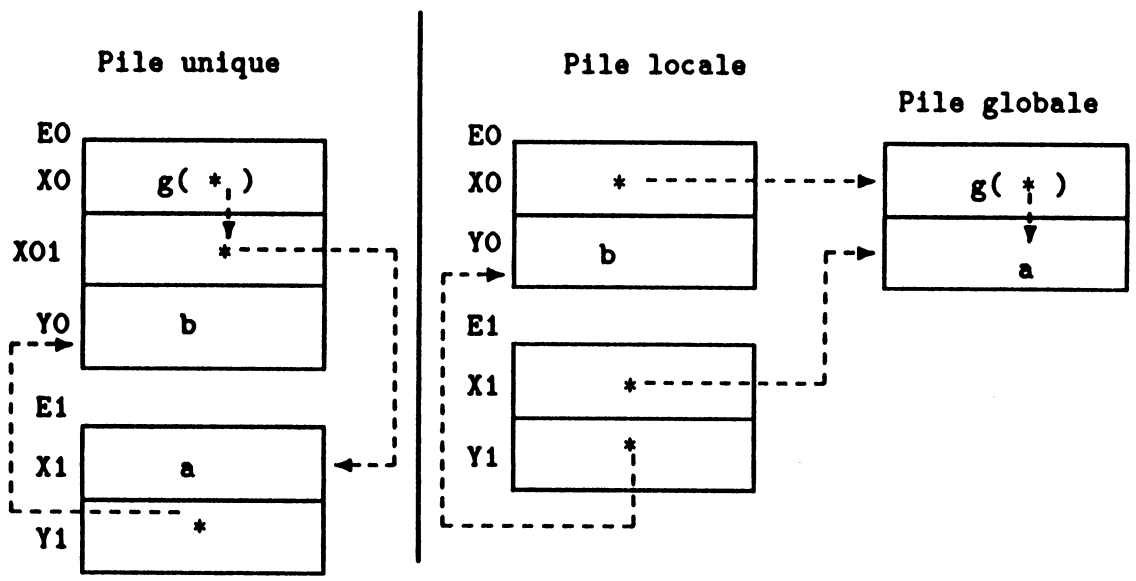

Figure 2.2: Problème des pointeurs fantômes

Par souci de simplicité seules les liaisons des variables sont représentées dans cette figure. Dans la partie gauche de la figure, la récupération de l'environnement E1 à la fin de l'exécution de la clause (2) transforme le pointeur vers *X1* en pointeur fantôme. L'environnement d'appel E2 n'est pas représenté car il ne contient aucune liaison de variable. La figure de droite montre comment l'utilisation de la pile globale résoud ce problème.

figure 2.2 entre *Y1* et *Y0*. L'opération de déréférencement consiste donc à suivre une chaîne d'indirections pour obtenir la valeur d'une liaison. Dans la figure 2.2, le déréférencement de *Y1* retourne la constante *c*, celui de *X1 : a* et celui de *X0 : g(a)*.

2.3.5 Optimisation d'appel terminal

Il est possible de récupérer l'environnement situé en sommet de pile locale si on se trouve dans la situation suivante :

- Les environnements d'appel des sous-buts déterministes sur la pile locale ont été libérés.
- Il s'agit de l'appel du dernier sous-but d'une clause, toutes les alternatives de cette clause ayant déjà été explorées.

Exemple :

```
a :- b, c.
b :- d, e.
b :- f, g.
f.
g :- p, q.
  :- a.
```

L'optimisation d'appel terminal [Warren 80] est réalisable à l'appel du sous-but *g*. En sommet de pile locale, on peut substituer à l'environnement d'appel du sous-but *b*, l'environnement d'appel du sous-but *g*. Des précautions doivent cependant être prises lors de l'unification du sous-but courant avec la tête de clause sélectionnée par la stratégie Prolog, afin d'éviter les pointeurs fantômes vers des variables qui seront désempilées. Il est parfois nécessaire de sauvegarder certaines variables dans la pile globale.

2.4 Présentation de la *WAM*

Nous ne donnons ici que les principales caractéristiques de la *WAM*. Le lecteur désirant plus d'informations est invité à consulter [Warren 83],[Chassin 86],[Boizumault 88].

2.4.1 Intérêt de la notion de machine abstraite

Les techniques de compilation Prolog utilisent des **machines abstraites** comme machines cibles des compilateurs. Le code objet constitué d'instructions de la machine abstraite est ensuite, soit interprété par un émulateur de la machine abstraite écrit en langage évolué (SICStus Prolog par exemple) ou en langage d'assemblage (Quintus Prolog), soit traduit en instructions de la machine cible (ALS Prolog),

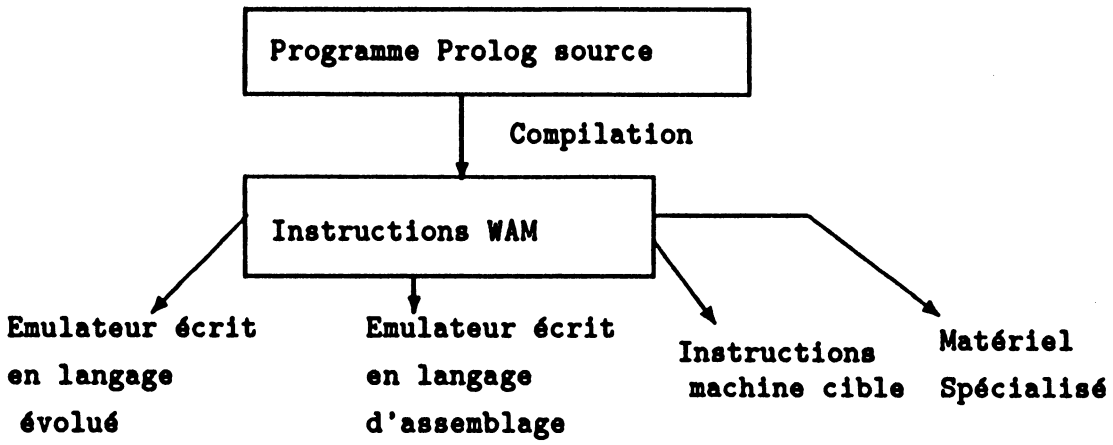

Figure 2.3: Différentes techniques d'exécution de la *WAM*

Les techniques d'exécution de la *WAM* sont classées par ordre d'efficacité croissante, de gauche à droite.

soit exécuté par du matériel spécialisé [Benker et al. 89] (voir figure 2.3). L'intérêt de cette technique est de faciliter la portabilité du compilateur. Celle-ci est très rapide dans les systèmes Prolog utilisant un émulateur de *WAM* écrit en langage évolué. Dans les autres cas le portage d'un système Prolog est limité à la réécriture des "couches basses" du système.

2.4.2 Principes de la compilation de Prolog

La compilation de Prolog repose essentiellement sur deux idées :

- accélération des unifications par la traduction des têtes de clauses en instructions spécialisant les unifications en fonction du type des arguments ;
- accélération du choix de la clause candidate à la résolution du sous-but courant. Des instructions d'aiguillage, permettant un choix rapide de la première clause unifiable en fonction des arguments du sous-but courant, sont générées par le compilateur au début de chaque paquet de clauses.

La *WAM* incorpore également toutes les optimisations classiques de gestion mémoire telle que l'optimisation d'appel terminal ; aucun environnement n'est créé pour les clauses comportant moins de deux sous-buts en partie droite.

2.4.3 Objets manipulés. Opérations élémentaires

Les objets manipulés par la *WAM* sont des mots étiquetés. L'étiquette contient le type de l'objet et éventuellement des informations pour le ramasse miettes ; elle est

identifiable rapidement, surtout par un processeur spécialisé. Une taille de mots importante est souhaitable ; un mot de 40 bits par exemple permet de disposer de 32 bits de données ou d'adresse, tout en gardant 8 bits d'étiquette. Les types d'objets possibles comprennent au moins les variables Prolog non liées, les entiers, réels, atomes, structures, pointeurs sur liste et référence.

Les trois opérations élémentaires de Prolog (voir paragraphe 2.3.4) sont définies sur ces objets, à savoir le **déréférencement**, l'**unification** de deux arguments et la **liaison** d'une variable logique avec un autre objet. Ces opérations sont appelées par les instructions qui compilent l'unification d'une tête de clause avec le sous-but courant (voir paragraphe 2.4.6).

2.4.4 Zones de données

La *WAM* manipule les trois piles définies dans la section 2.3 : **locale, globale et trace** (*trail*).

La pile locale contient deux types de structures de données : d'une part les **environnements** d'appels des clauses, d'autre part les **points de choix** (*choice-point*), qui sont créés lors de l'appel d'un prédicat comportant plusieurs clauses candidates, et qui contiennent toutes les informations nécessaires à la restitution de l'état du système à l'appel du prédicat. Les points de choix sont utilisés lors des retour-arrières. Un point de choix contient les arguments du prédicat courant, une référence à l'environnement de la clause appelante ainsi qu'un moyen d'accéder au code de la clause qui sera choisie lors du retour-arrière. La distinction entre point de choix et environnements est une optimisation de la *WAM*. Si on reprend l'exemple de la figure 2.1, la *WAM* crée un point de choix à l'appel du prédicat *t* d'arité deux, suivi éventuellement d'un environnement, si la clause (C2) comporte plusieurs sous-buts. Comme précédement, l'espace de la pile locale est récupéré lors des retour-arrières et des appels terminaux.

La pile globale contient les structures et les listes construites par unification. L'espace y est libéré lors des retours-arrières seulement. La pile globale contient parfois également les clauses introduites dynamiquement dans la base de clauses (*assert*). Cette pile est également appelée **tas** dans la mesure où, la récupération mémoire sur retour-arrière étant insuffisante, on y implante fréquement un ramasse miettes.

La pile trace ou encore **pile de restauration** sert à restaurer un état précédent du calcul lors d'un retour arrière. Elle contient des pointeurs sur les variables à réinitialiser lors d'un retour arrière. Lorsqu'une variable logique antérieure au niveau de retour arrière est liée, son adresse est enregistrée dans la pile trace.

2.4.5 Registres

La *WAM* dispose de nombreux registres qui se divisent en :

registres d'état : les registres d'état de la *WAM* gèrent la même information que les variables d'état définies au paragraphe 2.3.1. On trouve les poin-

teurs d'instruction courante **P** (*CLAUSE_COURANTE*), d'environnement courant **E** (équivalent à *ENVIRON_COURANT*), de point de choix **B** (*RETOUR_COURANT*), d'instruction de continuation **CP** (équivalent à *APPEL_COURANT*). En font également partie les pointeurs de sommets de piles locale **L**, globale **G** et trace **T** ; la redondance de certains registres d'état a pour but d'accélérer l'exécution des programmes.

registres arguments : ces registres sont en nombre important et contiennent les arguments des sous-buts. Ils sont chargés par les instructions de queue de clause et utilisés par les instructions de tête de clause (voir section suivante). Ils sont habituellement désignés par A_i ou X_i, le registre X_i contenant le $i^{ème}$ argument du sous-but appelé. En plus de ces registres, la *WAM* utilise un pointeur vers les éléments de listes et structures, **S**. **S** est initialisé par les instructions manipulant les listes et les structures. Il sert à en accéder efficacement les éléments.

2.4.6 Instructions

La *WAM* utilise une soixantaine d'instructions que l'on peut diviser en plusieurs classes.

Instructions d'indexation

L'indexation permet de sélectionner rapidement une clause dont la tête est susceptible de s'unifier au sous-but courant, parmi les clauses d'un paquet. Dans la *WAM*, l'indexation se fait suivant le type ou la valeur du premier argument. Il existe plusieurs instructions d'indexation et plusieurs niveaux d'indexation sont possibles, l'indexation au plus haut niveau se faisant alors suivant le type du premier argument et les indexations de niveaux inférieurs se faisant suivant sa valeur[3]. L'indexation suivant le premier argument se traduit par un style de programmation particulier où les programmeurs placent en première position l'argument le plus instancié, parfois au détriment de la lisibilité du programme. Pour y remédier et permettre une meilleure indexation, certains compilateurs Prolog indexent en utilisant l'argument le plus instancié. En raison de leur nom, les instructions d'indexation sont aussi appelées instructions *switch*.

Instructions de contrôle

Deux types de contrôle cœxistent durant l'exécution des programmes Prolog. Le premier, que nous appellerons la **continuation** est proche des langages impératifs et enchaine les appels de prédicats pour résoudre les sous-buts de la résolvante.

[3]L'indexation suivant la valeur du premier argument, en utilisant un mécanisme d'adressage dispersé, permet d'accélérer considérablement le choix de la clause candidate dans un prédicat type "base de données". Elle n'en demeure pas moins insuffisante pour les prédicats de forte arité pour lesquels certains systèmes Prolog implantent une indexation portant sur plusieurs arguments.

Il est implémenté dans la *WAM* par diverses instructions *call*, *execute* et *proceed*. A l'exécution d'une instruction *call* le registre continuation **CP** est chargé avec l'adresse de l'instruction suivante. L'instruction *execute* implémente l'optimisation de l'appel terminal dans le cas où la queue de clause n'a qu'un seul sous-but. Celui-ci est alors "appelé" sans création d'environnement (voir figure 2.4). Lors du succès dans la résolution d'un sous-but (instruction *proceed*), le contrôle est transféré à l'instruction désignée par le registre **CP**. Le pointeur d'environnement **E** contient alors l'adresse de l'environnement d'appel de la clause.

Le second mécanisme de contrôle, sans équivalent dans les langages impératifs est le **retour-arrière** et se produit lors d'un échec d'unification entre un sous-but et une tête de clause. Il peut aussi être provoqué artificiellement par le système si l'utilisateur désire une autre solution ou encore pour collecter l'ensemble des solutions à un prédicat (*bagof*, *setof*). Le retour-arrière est implémenté par les instructions *try*, *retry* et *trust*. La première sauvegarde l'état courant du système en créant un **point de choix**. L'adresse du point de choix est chargée dans le registre **B**. Les instructions *retry* et *trust* restaurent l'état du système en utilisant le point de choix pointé par **B**. En outre, *trust* libère le point de choix et fait pointer **B** sur le point de choix précédent.

Compilation de l'unification

L'unification est compilée en deux séries d'instructions : d'une part les instructions de queue de clause qui préparent les arguments des sous-buts dans les registres arguments avant l'appel des clauses unifiables au sous-but ; d'autre part les instructions de tête de clauses qui réalisent l'appariement des objets de tête de clause avec les arguments du but courant. La spécialisation en fonction du type d'éléments de la tête de clause permet réduire le plus souvent l'unification à quelques tests simples. L'exemple le plus significatif est la première apparition d'une variable en tête de clause, qui est compilée en une instruction d'affectation. De plus, l'utilisation des étiquettes des données permet d'accélérer les tests. L'opération de déréférencement est fréquement appelée par les instructions de tête de clause. En raison de leurs noms les instructions de tête et de queue de clause sont appelées respectivement instructions *get* et *put*.

Certains arguments de tête de clause ne permettent pas la génération d'instructions spécialisées à la compilation. L'instruction *WAM* générée par le compilateur pour unifier un argument de ce type avec l'argument correspondant du sous-but courant se réduit alors à un appel à l'unification générale.

2.4.7 Exemple de compilation

La compilation de Prolog en instructions de la *WAM* est illustrée dans la figure 2.4 par l'exemple simple du programme de concaténation de deux listes.

On peut remarquer que lorsqu'il est appelé avec des arguments instanciés le programme de concaténation ne crée ni point de choix ni environnement sur la

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1, L2, L3).
```

CODE SOURCE	CODE COMPILE en WAM
	conc: switch_on_term(C1a,C1,C2a,fail)
	try C1
	trust C2
conc(	C1:
[],	get_nil X1
L, L	C1a: get_value X2, X3
).	proceed
conc(	C2:
[	get_list X1
X |	C2a: unify_variable X4
L1], L2,	unify_variable X1
[	get_list X3
X |	unify_value X4
L3]) :-	unify_variable X3
conc(L1, L2, L3).	execute conc

Figure 2.4: Exemple de compilation Prolog

L'indexation permet de sélectionner directement la "bonne" clause si le premier argument est instancié. Dans le cas contraire un point de choix est créé (instruction *try*) puis enlevé (instruction *trust*). L'instruction *execute* implémente l'optimisation de récursion terminale, qui évite la création d'environnement sur la pile pour ce programme. Les instructions *get_nil* et *get_list* ne sont que de simples tests sur les étiquettes des arguments. Les instructions *unify* utilisent implicitement le registre pointeur d'argument S. L'instruction *unify_variable* se réduit à une simple affectation tandis que les instructions *get_value* et *unify_value* sont des appels à l'unification générale. L'allocation des registres est optimisée dans la compilation de la seconde clause, ce qui permet d'éviter la génération d'instructions de queue de clause (*put*).

pile. C'est pour cette raison qu'il est fréquement utilisé pour mesurer les systèmes Prolog à des fins "publicitaires", puisqu'il permet d'exhiber des performances très élevées. Sa valeur de test n'en demeure pas moins réduite puisqu'il n'utilise pas certaines caractéristiques importantes du langage, tel le non-déterminisme.

2.4.8 Ramasse-miettes pour Prolog

Les techniques de récupération mémoire développées pour Prolog se révèlent insuffisantes pour de grosses applications. En particulier la pile globale n'est récupérée que lors des retours arrières ce qui peut être problématique dans le cas des grosses applications déterministes. Des techniques de ramasse miettes (*garbage collection*) adaptées à la *WAM* [Appleby 88] rendent les systèmes Prolog compilés utilisables pour ces applications. Signalons également l'existence d'une technique d'implémentation de Prolog [Bekkers 86] basée sur une récupération mémoire complète. La partition entre les piles locale et globale disparaît et toute la mémoire est allouée dans un tas qui est retassé parallèlement à l'exécution du programme. Cette implémentation obtient des résultats remarquables pour la résolution de très gros problèmes.

2.5 Résumé du chapitre

Ce chapitre rappelle les notions nécessaires à la compréhension du reste de la thèse. Les résultats de démonstration automatique servant de fondations au langage Prolog sont présentés et ce dernier est rapidement décrit. La suite du chapitre est consacrée aux techniques d'implémentation de Prolog qui sont introduites par la description du fonctionnement d'un interprète simplifié du langage. Les techniques de compilation "standard" de Prolog en utilisant la *Warren Abstract Machine* ou *WAM* font l'objet de la dernière partie de ce chapitre.

Chapitre 3

Parallélisme en Programmation Logique

Comme nous l'avons vu au chapitre précédent, la sémantique des langages de programmation logique n'est en rien séquentielle. La SLD-résolution n'impose pas de stratégie pour l'exploration de l'arbre de réfutation associé à un programme, même si la plupart des implémentations utilisent une stratégie en profondeur d'abord, pour des raisons d'efficacité. La résolution parallèle de programmes logiques ne change donc pas leur sémantique déclarative mais *seulement* leur sémantique opérationnelle. Pour cette raison, les langages logiques sont souvent considérés comme les meilleurs candidats pour la programmation des multiprocesseurs [Baldwin 87].

Dans la suite de ce chapitre, nous présentons les sources de parallélisme en programmation logique ainsi que les caractéristiques des projets visant à les exploiter. Ces projets peuvent être regroupés en deux grandes familles qui sont les **systèmes parallèles logiques gardés** et les **systèmes parallèles logiques non déterministes**. La présentation des systèmes de la première famille sera brève, puisque le projet PEPSys, décrit plus en détail au chapitre suivant, appartient à la seconde famille. Dans les projets de la seconde famille, on distinguera entre les modèles "théoriques", dont l'auteur ne pense pas qu'ils puissent être mis en œuvre efficacement dans un avenir proche, et les modèles **multiséquentiels**, dont le modèle PEPSys fait partie, qui visent à procurer des gains d'efficacité sur les multiprocesseurs existant actuellement.

Le lecteur intéressé trouvera dans [Chassin, Codognet et al. 89] une étude bibliographique plus complète des systèmes de programmation logique parallèle.

3.1 Sources de parallélisme en programmation logique

Les principales sources de parallélisme en programmation logique correspondent à l'exploration en parallèle des différentes branches des nœuds ET-OU de l'arbre de réfutation, et sont pour cette raison appelés parallélisme ET et parallélisme

OU. Les autres types de parallélisme sont le plus souvent des variantes des deux premiers : le parallélisme de flot est une variante du parallélisme ET alors que le parallélisme de base de données est une variante du parallélisme OU. Le parallélisme d'unification semble quant à lui plus difficilement exploitable.

3.1.1 Parallélisme OU

Définition du parallélisme OU

Le parallélisme OU apparaît au cours de l'exécution d'un programme logique lorsque le sous-but courant de la résolvante, en Prolog le premier de la résolvante, est unifiable à plusieurs têtes de clauses du programme. Dans un modèle exploitant le parallélisme OU, plusieurs de ces clauses sont essayées en parallèle. Pour illustrer cette définition, considérons la résolvante : $p(X,Y), q(Z,X)$, le prédicat p d'arité deux étant défini dans le programme par :

$$p(X,Y) \leftarrow ..., X = a, s_1(X,Y)$$

$$p(X,Y) \leftarrow ..., X = b, s_2(X,Y)$$

$$p(X,Y) \leftarrow ..., X = c, s_3(X,Y)$$

Les trois clauses de p peuvent être résolues en parallèle. Parmis ces résolutions quelques unes peuvent réussir et donc plusieurs liaisons de la variable X peuvent être passées aux s_i et à q. Même si la "taille" de p est modeste, la granularité du parallélisme peut être importante si la résolution de q nécessite beaucoup de calculs ; en effet, toutes les branches calculerons le prédicat q en parallèle. Dans un programme logique, les prédicats définis par plusieurs clauses sont des sources potentielles de parallélisme OU. Il va sans dire que toutes les branches potentielles ne conduisent pas à une solution du problème, ni même à celle du prédicat (p dans l'exemple précédent).

Le parallélisme OU dans les programmes logiques

Le parallélisme OU n'apparaît pas uniquement dans les programmes délivrant beaucoup de solutions. Ce type de parallélisme est utilisable pour tous les programmes non-déterministes, même s'il ne produisent que peu de solutions. Si l'objectif du programme est de ne trouver qu'une seule solution à un problème, l'exploration de l'arbre de réfutation tire parti de l'utilisation du parallélisme, pourvu que les branches OU de l'arbre soient longues et nombreuses. Souvent le parallélisme OU ne permet pas d'accélérer l'exécution des programmes existant car ceux-ci utilisent fréquement, pour des raisons d'efficacité, des algorithmes complètement déterministes. Le parallélisme OU est néanmoins exploitable pour de grandes classes de problèmes qui font naturellement appel au non-déterminisme. On peut citer par exemple des algorithmes de recherche dans les graphes qui sont essentiels en intelligence artificielle, l'analyse du langage naturel ou des traitements de base de données. L'activation de règles dans un système expert peut se faire de

façon non-déterministe. Enfin de nombreux traitements itératifs peuvent tirer parti du parallélisme OU, invoqué alors par un prédicat "système" d'ordre supérieur tel que *bagof*. On peut considérer, par exemple, la compilation en parallèle de toutes les procédures d'un fichier.

Gestion des résolvantes multiples

L'un des principaux problèmes que pose l'implémentation d'un système Prolog OU-parallèle est la gestion des résolvantes multiples. Une résolvante différente est attribuée à chacune des activités créées après un nœud OU-parallèle. Dans l'exemple précédent, trois activités OU-parallèles peuvent être créées pour résoudre en parallèle les résolvantes :

$$\leftarrow s_1(a,Y), q(Z,a) \quad , \quad \leftarrow s_2(b,Y), q(Z,b) \quad , \quad \leftarrow s_3(c,Y), q(Z,c)$$

Dans la plupart des implémentations Prolog, une résolvante est représentée par une pile d'environnements chainés, contenant des variables logiques liées entre elles lors des unifications intervenues depuis le début de l'exécution du programme. A l'initialisation d'une nouvelle activité, il existe plusieurs possibilités pour construire la résolvante associée à l'activité. Certaines solutions consistent à créer une copie de la résolvante, exclusive à chacun des processus OU-parallèles. D'autres solutions essaient de limiter le surcoût de création de chaque nouvelle activité, en partageant les parties communes de la pile des environnements de variables entre les processus OU-parallèles. Le problème est alors de limiter le surcoût lors de l'accès aux variables des environnements partagés. Comme le projet PEPSys appartient à cette dernière catégorie, ce problème sera largement développé par la suite.

Complétude de la résolution OU-parallèle

La stratégie de résolution OU-parallèle est "plus complète" que la stratégie en profondeur d'abord utilisée par des systèmes comme Prolog. Le fait qu'un processeur explore une branche infinie n'empêche pas les autres de calculer une solution. Cela ne suffit pas à faire des modèles exploitant le OU-parallélisme des mises en œuvre complètes de la résolution puisque, comme l'explique [Kalé 87b], la complétude ne peut exister que si le retour-arrière n'est jamais utilisé. Ceci n'est possible que si l'on dispose de ressources infinies, ce qui n'est pas très réaliste.

3.1.2 Parallélisme ET

Définition du parallélisme ET

Le parallélisme ET consiste en la résolution simultanée de plusieurs littéraux d'une résolvante. Soit par exemple la résolvante :

$$l_1, ...l_i, ..., l_n$$

Le parallélisme ET sera obtenu par résolution simultanée de plusieurs des littéraux l_i. Le grain du parallélisme ET semble plus fin que celui du parallélisme OU ; en effet, le grain du parallélisme ET correspond à la résolution d'un sous-but, alors que dans le cas du parallélisme OU, le grain correspond à la résolution complète d'une résolvante.

Le parallélisme ET dans les programmes logiques

A la différence du parallélisme OU, le parallélisme ET apparaît dans tous les programmes logiques, même complètement déterministes. Cependant, ainsi que nous le verrons dans les sections suivantes, sa faible granularité et la difficulté de prévention des liaisons conflictuelles aux variables partagées rendent son exploitation efficace difficile.

Préventions des liaisons conflictuelles

Il semble peu réaliste de proposer de développer indépendamment les sous-buts d'une clause pour s'assurer ensuite de la cohérence des variables partagées par une opération du type *join* (cf. bases de données) ; il est donc nécessaire de synchroniser les différents sous-buts qui ont des variables communes.

Une première approche est de considérer qu'une telle synchronisation au niveau des variables est suffisante. Elle doit être donnée par le programmeur, qui exprimera un certain contrôle sur le parallélisme en indiquant de quelles variables un littéral est producteur ou consommateur. C'est l'approche utilisée dans les *langages gardés*, qui seront présentés dans la section suivante.

On peut également résoudre le problème des variables communes en considérant que les littéraux partageant une variable doivent être exécutés de manière séquentielle comme cela est proposé par [Conery 83] (voir aussi le paragraphe 3.3.2). On obtient ainsi une synchronisation brutale entre sous-buts, assurant ainsi qu'aucun conflit ne peut apparaitre dans les liaisons de variables puisqu'on exécute en parallèle uniquement les littéraux sans variables communes. De cette façon on n'a plus alors qu'un seul environnement d'unification à gérer, comme en Prolog séquentiel. Notons que le parallélisme ainsi obtenu est plus faible que celui utilisé dans les langages gardés puisque les sous-buts partageant des variables ne peuvent se communiquer les termes auxquels celles-ci sont instanciées au fur et à mesure de leurs constructions. Considérons par exemple un but:

$$\leftarrow p(X), q(X,Y)$$

et les clauses suivantes pour p et q:

$$p([a|Y]) \leftarrow p'(Y)$$

$$q([a|Y], Z) \leftarrow q'(Z)$$

$$q([b|Y], Z) \leftarrow q''(Z)$$

q peut à priori s'exécuter dès que X est liée à $[a|Y]$, sans attendre que p' soit terminé. Ce type de parallélisme, appelé **parallélisme de flot** est exploité par les systèmes parallèles logiques gardés qui sont décrit dans la section suivante. Cependant, si l'on considère que les littéraux partageant une variable doivent s'exécuter séquentiellement, q doit attendre que p (donc p') soit terminé pour commencer. Ceci explique donc le terme de parallélisme ET **restreint** employé par [DeGroot 84] pour désigner une telle méthode où seuls les littéraux indépendants (sans variables communes) sont exécutés en parallèle.

Cependant détecter l'indépendance entre littéraux n'est pas chose facile. [Conery 83] propose un modèle où les dépendances entre littéraux sont calculées dynamiquement et représentées par des graphes indiquant les relations de production/consommation des variables entre les différents sous-buts. Bien que ce modèle soit attrayant du point de vue théorique, le recalcul dynamique des dépendances entre littéraux s'avère peu efficace en pratique. Plus récement, un autre modèle permettant une analyse dynamique efficace de l'indépendance entre littéraux a été proposé [Lin 88]. Des jetons (*tokens*), représentés par des vecteurs de bits, sont associés aux variables partagées. La détection dynamique des dépendances se réduit donc à quelques opérations sur vecteurs de bits.

A l'opposé, on peut chercher à calculer statiquement ces dépendances par une analyse faite lors de la compilation du programme. Dans le cas général, une telle analyse statique ne peut découvrir le parallélisme maximal possible (car on doit choisir le "pire des cas" pour les possiblités d'instanciation des variables), mais ceci est compensé par l'abandon d'algorithmes coûteux de détection de dépendance lors de l'exécution. Plusieurs méthodes d'analyse statique ont été proposés [DeGroot 84,Mannila 87,Codognet 86], principalement basées sur la notion d'**interprétation abstraite** (voir [Mellish 86]). L'interprétation abstraite d'un programme consiste en une exécution symbolique où l'on ne s'intéresse pas à la valeur des termes calculés (interprétation **réelle** du programme) mais uniquement à des propriétés de ceux-ci, par exemple déterminer si certains termes sont interdépendants (partagent des variables) ou si certains termes sont clos.

Ce domaine de recherche en programmation logique est cependant très récent, et il n'est pas sûr qu'actuellement la recherche soit suffisament avancée pour qu'une analyse statique faite à la compilation puisse à elle seule découvrir un parallélisme. Dans l'attente de telles analyses sophistiquées, certains proposent de laisser au programmeur le soin d'indiquer les littéraux indépendants, comme [Ratcliffe et Syre 87] pour le langage PEPSys.

Une méthode mi statique et mi dynamique, appelée **parallélisme ET restreint** a été proposée par [DeGroot 84]. Une analyse statique est effectuée dans le but de transformer chaque clause en une expression conditionnelle exhibant le parallélisme possible ; celle-ci comporte des tests simples sur les variables afin de représenter plusieurs schémas possibles d'exécution. Cette méthode, qui semble représenter un compromis intéressant, est décrite ultérieurement.

3.1.3 Autres sources de parallélisme en programmation logique

Nous avons déjà mentionné le **parallélisme de flot**, variante du parallélisme ET et qui est exploité par les langages gardés.

Le **parallélisme de base de données** consiste à découper les "gros" prédicats contenant des faits (appelés habituellement prédicats bases de données) en parties exécutées en parallèle. Il s'agit donc d'une variante du parallélisme OU, qui semble moins efficace pour accélérer ce type de traitement qu'une "connection" entre le langage de programmation logique et un système de bases de données (comme par exemple [Bocca et al. 86]).

Le **parallélisme d'unification** consiste à unifier en parallèle les arguments d'un prédicat. Ce parallélisme ne semble pas exploitable efficacement. Il semble tout d'abord que l'unification soit intrinsèquement séquentielle [Dwork 84]. Des mesures faites sur des programmes existant [Ratcliffe et Robert 85] ont montré que le nombre d'arguments des prédicats est en moyenne inférieur à trois. Enfin, il s'agit d'un parallélisme de grain extrêmement fin.

3.2 Systèmes parallèles logiques gardés

3.2.1 Introduction aux systèmes gardés

Les notions de base de la plupart des langages de Programmation Logique Parallèle gardés ont été introduites par le Relational Language [Clark 81b], dont les origines remontent aux travaux sur IC-Prolog [Clark 81a]. On peut les résumer brièvement par trois caractéristiques :

Garde , concept introduit par Dijkstra pour les langages algorithmiques parallèles, qui consiste à spécifier pour chaque clause du programme une condition qui doit être satisfaite afin que le corps de la clause puisse être exécuté ;

Don't care non-determinism , qui restreint le non-déterminisme au choix d'une clause parmi un paquet de clauses définissant un prédicat, ce choix ne pouvant plus être remis en cause par la suite en cas d'échec, éliminant ainsi tout retour-arrière ;

Synchronisation entre processus par indication du mode d'accès (lecture ou écriture) des arguments. Ces annotations contrôlent les canaux de communication entre les sous-buts d'une clause.

Ces caractéristiques sont communes aux langages gardés, dont les trois principaux représentants sont :

Parlog [Clark 86] (Imperial College, G.B.),

Concurrent Prolog ou CP [Shapiro 83a] (Weizmann Institute, Israel), et

Guarded Horn Clauses, ou GHC [Ueda 85] (ICOT, Japon).

3.2.2 Syntaxe et sémantique des langages gardés

Syntaxe

En CP, GHC ou Parlog, un programme est essentiellement un ensemble fini de **clauses définies gardées** du type:

$$P \leftarrow G_1, \ldots, G_n | B_1, \ldots, B_p \; n, p \geq 0$$

| est appelé opérateur d'engagement (*commit*), $G_1, \ldots, G_n$ la garde de la clause, $B_1, \ldots, B_p$ son corps et P sa tête. L'ensemble des clauses comportant le même prédicat de tête (avec la même arité) constitue la **définition** de ce prédicat, et chaque clause est appelée **clause alternative** pour celui-ci. Un but reste comme en Prolog une conjonction de littéraux : $\leftarrow L_1, \ldots, L_n$

3.2.3 Sémantique informelle

La signification déclarative de la clause précédente est :

$$P \textit{ si } G_1 \textit{ et } \ldots \textit{ et } G_n \textit{ et } B_1 \textit{ et } \ldots \textit{ et } B_p$$

Sa sémantique procédurale est cependant plus complexe. Lors de la réfutation d'un littéral P on lance en parallèle autant de processus qu'il y a de clauses alternatives dans la définition de P, chacun de ceux-ci étant chargé d'exécuter l'unification avec la tête d'une alternative et la garde de celle-ci. L'une des clauses ayant évalué sa garde avec succès est choisie aléatoirement, la clause contenant cette garde est engagée (*commit*) et les processus évaluant les autres gardes de P sont arrêtés. L'opération d'engagement est atomique. Ce non-déterminisme est appelé *Don't care non-determinism* : on n'attache pas d'importance au choix de la clause qui sera retenue (i.e. dont le corps sera exécuté) du moment que la garde est évaluée avec succès. On perd donc le non-déterminisme fondamental de Prolog (*Dont know non determinism*), la garde étant en quelque sorte une systématisation de l'opérateur *cut*.

L'utilisation de clauses gardées limite le parallélisme OU à l'exécution simultanée des gardes des clauses alternatives définissant un prédicat. La gestion du parallélisme ET nécessite quant à elle d'introduire des mécanismes supplémentaires de *synchronisation* entre les différents littéraux du but courant. L'idée de base du **parallélisme de flot** (*stream-parallelism*), sur lequel sont basés les langages précédents, est de définir des relations du type Producteur/Consommateur entre les différents littéraux du but courant (i. e. entre les différents processus actifs). Ces relations sont réalisées en précisant, pour chaque littéral, un mode d'accès pour les variables partagées par plusieurs littéraux : une occurrence de variable en *entrée* dans un littéral indique que celui-ci est **consommateur** de ce terme (accès uniquement en lecture) et sera donc mis en attente tant que le terme n'aura

pas été produit ; une occurence de variable en *sortie* dans un littéral indique que celui-ci sera **producteur** du terme correspondant (accès en lecture ou écriture). Par exemple, considérons le but : $\leftarrow p(X), q(X), r(X)$. En indiquant que X est en sortie dans p et en entrée dans q et r, on force une exécution où q et r sont suspendus tant que p n'a pas produit X, q et r s'exécutant en parallèle dès que X est produit.

Ce mécanisme de synchronisation, sur les variables partagées et non directement entre littéraux, est très puissant. En effet, les termes communiqués sont transmis au fur et mesure de leurs constructions, permettant ainsi un parallélisme important.

3.2.4 Problèmes posés par ces langages

Restriction des gardes

Sûreté des gardes

L'exécution parallèle des gardes des clauses alternatives lors de l'appel à un littéral pose, comme toute exécution parallèle OU, un problème de gestion des environnements multiples nécessaires. En effet chaque processus pouvant à priori instancier des variables du littéral appelant, il est nécessaire que ceux-ci possèdent chacun un environnement distinct, formé des variables locales à la clause et des variables de l'appelant. La sélection d'une clause unique parmi les alternatives ne simplifie rien puisqu'il faut non seulement gérer des environnements multiples jusqu'à l'engagement mais en plus assurer l'atomicité de cette opération ainsi que l'unicité du processus choisi.

Ce problème a conduit les concepteurs des langages gardés à définir la notion de garde **sûre** . Une garde est dite sûre (*safe*) si elle n'instancie aucune variable de l'appelant. Cela revient à n'utiliser les gardes que comme des tests sur les termes transmis par l'appelant, toute modification de ceux-ci devant se faire dans le corps de la clause. L'utilisation de gardes sûres permet de n'avoir à gérer qu'un environnement unique.

Platitude des gardes

Dans une garde, les littéraux peuvent être des prédicats définis par le programmeur, donc un appel à un tel littéral lancera en parallèle des processus correspondant aux gardes des clauses alternatives de sa définition. Ainsi une garde peut engendrer un autre système de gardes, chacune des gardes pouvant elle-même engendrer un autre système de garde, et ainsi de suite. Cependant dans certaines applications, comme par exemple la programmation système, il n'est pas nécessaire d'engendrer une telle hiérarchie de gardes car celles-ci sont réduites à des appels de prédicats prédéfinis. Dans un tel cas le système de gardes est **plat**, puisqu'une garde n'en appelle jamais une autre, et l'implémentation est alors simplifiée.

Des restrictions plates des langages gardés ont été définies, dans lesquelles seuls les prédicats systèmes prédéfinis sont autorisés dans les gardes. Ces prédicats prédéfinis opérant en général sur des arguments totalement instanciés, les gardes de FCP et de FGHC sont en général sûres. Ces langages peuvent donc bénéficier d'implémentations plus efficaces.

Sémantique

L'opérateur d'engagement et le *Don't care non-determinism* introduisent des **incomplétudes** dans la procédure de résolution et les mécanismes de synchronisation (mode des arguments et règles de suspension des processus) peuvent mener à des phénomènes de **blocage** (*dead-locks*). De plus ces langages n'ont pas toujours été spécifiés avec toute la rigueur nécessaire pour lever les ambiguités sémantiques, en particulier la définition originelle de CP [Shapiro 83a] pose de nombreuses questions sur la sémantique du langage (voir [Saraswat 86]).

Efficacité

Durant l'exécution d'un programme d'un langage gardé, un processus est créé pour chaque littéral du but, ceux-ci engendrant eux-même des processus fils au cours de leurs exécutions et ainsi de suite. Ces processus sont mis en attente si leurs termes en entrée n'ont pas encore été produits ou si tous les processeurs sont déjà occupés à résoudre d'autre sous-buts. Ce mécanisme de synchronisation risque d'engendrer un nombre important de processus, et en particulier de processus suspendus, ainsi que de fréquents changements de contextes. Un autre problème réside dans la faible granularité du parallélisme exploité par ces langages, qui les rend difficiles à implémenter efficacement sur les architectures multiprocesseurs existantes.

3.2.5 Utilisation des langages gardés

L'un des objectifs principaux des langages gardés est de devenir les langages de base de (futures) machines parallèles utilisant la Programmation Logique, pour être en quelque sorte à ces machines ce que le langage C est aux machines UNIX [1]. Les versions simplifiées des langages gardés, *Flat GHC, Flat CP* ou *Flat Parlog*, sont destinées à permettre d'écrire la plus grande partie d'un système d'exploitation. Pour plus d'information sur le sujet voir [Silvermann 86] et [Sato 87].

Il semble également que ce type de langages soit adapté à la **programmation orientée objet** [Shapiro 83b] : il existe une analogie naturelle entre objet et littéral, le symbole de prédicat représentant le type de l'objet et les arguments de celui-ci ses attributs. Dans un langage gardé, un objet peut être représenté par un littéral du but courant, ou plus exactement par une suite de littéraux précisant les divers changements d'états de l'objet, c'est-à-dire un processus perpétuel correspondant à cette suite de littéraux récursivement énumérés. La communication

[1] UNIX est un produit de ATT Bell Labs.

entre objets se fait grâce aux variables partagées, simulant ainsi l'envoi de messages.

3.2.6 Implémentations des langages gardés

Si les langages gardés se sont révélés difficiles à implémenter en raison des problèmes qu'ils posent, leurs versions plates ont donné lieu à de nombreuses implémentations dont nous ne pouvons mentionner ici que quelques unes.

Implémentations séquentielles

Des machines abstraites, basées sur la WAM, ont été définies pour FCP [Houri 86], GHC [Levy 86b], FGHC (KL1) [Kimura 87] et Parlog [Foster 86]. Notons que toutes ces implémentations sont optimisées pour une exécution séquentielle, utilisant une stratégie d'exploration en profondeur d'abord limitée de l'arbre ET/OU des processus et évaluant les gardes des clauses alternatives séquentiellement et non en parallèle.

D'autres méthodes sont basées sur la compilation des prédicats en graphes de décision. Un compilateur d'un sous-ensemble de FCP sans unification générale, produisant du code natif, atteint la performance maximale de 75 KLIPS sur Sun-3 [Kliger 88].

Implémentations parallèles

Là encore le nombre des travaux ne permet pas de les citer tous. Citons les implémentations de FCP [Levy 86a], Parlog [Crammond 86] et KL1 [Sato 88]. Sur multiprocesseur à mémoire partagée de type Sequent Symmetry, les performances sont de l'ordre de 5 Klips par processeur, le facteur d'accélération atteignant cinq sur six processeurs. Signalons enfin l'importance des travaux menés sur la machine PIM (*Parallel Inference Machine*), dans le cadre du projet japonais de cinquième génération.

3.3 Modèles "théoriques"

Nous abordons l'étude des modèles de programmation logique parallèle non déterministes en évoquant brièvement quelques modèles que nous qualifions de "théoriques" en raison de la difficulté qu'ils présentent à implémenter efficacement. L'un des buts essentiels de ce type de modèles est d'extraire *tout* le parallélisme potentiel des programmes Prolog. L'accent n'est pas tant mis sur la faisabilité d'une implémentation efficace sur les matériels existants que sur la correction de l'algorithme de résolution et l'importance du parallélisme extrait. Ces modèles ne semblent cependant pas implémentables efficacement sur les multiprocesseurs existants pour les raisons suivantes :

- ils mettent un jeu un très grand nombre de processus, très supérieur au nombre de processeurs existants sur les multiprocesseurs actuels ;
- ces processus échangent entre eux de nombreux messages, certains de taille non bornée, ce qui entraine un surcoût très important du aux communications ;
- la granularité de ces processus est le plus souvent extrêmement faible et leur contribution à la solution ne compense pas le surcoût due à leur création.

3.3.1 COALA

Le modèle d'exécution et l'architecture COALA (Calculateur Orienté Acteur pour la Logique et ses Applications) [Percebois et al. 86] utilise le langage Prolog pur. Dans ce schéma d'exécution, un programme est *précompilé* en un graphe dont les nœuds sont les clauses du programme ; un arc entre le littéral l d'une clause c_1 et une clause c_2 indique que l est unifiable (statiquement) avec la tête de c_2 et représente donc une possibilité de réduction de l. Cet arc est étiqueté par la substitution correspondante et cette étiquette est appelée environnement d'unification de l'arc. Dans ce modèle, les arcs sont considérés comme les éléments de base (acteurs), et l'exécution d'un programme correspond au "dépliage" du graphe statique précompilé : on crée de nouveaux arcs représentant des chemins parcourus dans le graphe, donc des résolvants. La communication entre acteurs se fait par l'envoi de messages. Un simulateur détaillé de COALA, écrit en T-Prolog, a été réalisé. Sur des programmes simples, la courbe des performances, par ailleurs très faibles, s'aplatit à partir d'une demi-douzaine de processeurs.

3.3.2 Le modèle ET-OU

Le modèle ET-OU [Conery 83] vise à l'implémentation de programmes logiques *purs*. Chaque étape de l'exécution donne lieu à la création de deux types de processus : les processus **ET** et **OU**, un processus ET étant chargé de résoudre le corps d'une clause alors qu'un processus OU est chargé de résoudre un littéral à l'intérieur d'une clause. Chaque processus crée d'autres processus - ses fils - ce qui construit un arbre de processus et échange avec eux et avec le processus père des messages.
Exemple :

```
p( X1, X2 ) :- p1( X1, X2 ), p2( X1, X2 ).
p1(...) :- ...        p2(...) :- ...
p1(...) :- ...
```

Le processus ET créé pour résoudre la clause p crée lui-même deux processus OU pour résoudre p_1 et p_2, qui eux-mêmes créent respectivement deux et un processus ET, etc. Les processus ET assurent l'appariement des solutions trouvées par leurs processus fils de type ET, tandis que les processus OU collectent l'ensemble des solutions produites par leurs processus fils de type ET.

Une des principales caractéristiques du modèle est que certains messages, envoyés du fils au père après le calcul d'une solution, contiennent une copie du littéral résolu. De ce fait les solutions intermédiaires sont copiées de processus en processus vers la racine de l'arbre et la taille des messages n'est pas bornée. Le comportement des processus ET constitue la deuxième caractéristique du modèle. Contrairement aux modèle gardés, deux littéraux susceptibles de lier la même variable ne sont pas exécutés en parallèle. Des algorithmes complexes permettent, à l'exécution, de réordonner les buts d'une clause et de trouver le parallélisme maximal entre buts indépendants.

3.3.3 MIPAP

Mipap (Modèle d'Interprétation Parallèle pour Prolog) est un modèle développé au CNET-Lannion ([Kharoune 88]) et qui s'inspire de [Conery 83] et des techniques de détermination statique du parallélisme ET (voir les sections 3.1.2 et 3.4.5). Un des principaux intérêts de ce modèle réside dans la détection statique du parallélisme ET qui permet de diminuer le coût des tests à l'exécution par rapport à [Conery 83]. Le modèle proposé a été simulé et les résultats concernant la simulation d'une machine à base de processeurs 68000 montrent des gains de performances presque linéaires, en fonction du nombre de processeurs. Cependant la simulation s'appuie sur un système interprété dont la vitesse est relativement faible et il est à craindre que les coûts de transmission de messages deviennent relativement trop importants si l'exécution est optimisée (par exemple en compilant).

3.3.4 REDUCE-OR Process Model

Le REDUCE-OR Process Model [Kalé 87a] est basé sur une représentation originale de l'arbre de réfutation d'un programme Prolog, le *REDUCE-OR tree*. Les principales qualités de ce modèle, selon son auteur, sont d'extraire le parallélisme potentiel maximal des programmes Prolog ainsi que de donner une implémentation complète de Prolog, contrairement à la résolution. On peut cependant remarquer que la granularité du parallélisme extrait semble extrêmement faible et donc difficile à exploiter efficacement, d'autant plus que le modèle de calcul suppose l'échange de nombreux messages de tailles non bornées entre les nombreux processus. Par ailleurs la complétude de la résolution suppose l'existence de ressources illimitées, ce qui est rarement le cas dans la pratique! Des implémentations de ce modèle basées sur la *WAM* ont été faites sur plusieurs multiprocesseurs à mémoire commune ou non [Ramkumar et Kalé 89]. Les premiers résultats montrent de bons facteurs d'accélération en exécution parallèle sur de petits programmes (*fibonacci, queens*). On peut toutefois remarquer que ces bonnes performances ont été obtenues en restreignant sévèrement le parallélisme au moyen de *pragmas* et de duplication des prédicats parallèles[2] dans le programme source.

[2]Version parallèle au delà d'une certaine granularité, séquentielle en deça.

De plus les programmes de test choisis utilisent des données et produisent des résultats de taille réduite, ce qui limite la taille des messages échangés, importante source potentielle d'inefficacité.

3.4 Systèmes multiséquentiels

3.4.1 Caractéristiques communes aux systèmes multiséquentiels

Les systèmes de programmation logique parallèle multiséquentiels ont pour but essentiel de résoudre les problèmes que pose l'utilisation efficace du parallélisme en programmation logique sur les multiprocesseurs MIMD existants. Ces problèmes sont analogues à ceux qui se posent dans d'autres systèmes de programmation parallèle et certains ont déjà été mentionnés dans la section sur les modèles théoriques, pour signaler que ces modèles les prenaient imparfaitement en compte. Parmi ces problèmes, on peut citer :

communication : on cherche à minimiser le volume d'informations échangées entre processus coopérant à l'élaboration d'une solution.

synchronisation : on cherche a minimiser les synchronisations entre processus parallèles, celles-ci étant susceptibles de diminuer le parallélisme effectif à l'exécution.

granularité : la granularité des activités parallèles doit être suffisante pour compenser le surcoût provenant de l'installation de ces activités.

A la différence des systèmes "théoriques" qui cherchent à exploiter "tout" le parallélisme potentiel des programmes Prolog, les systèmes multiséquentiels ont pour but de n'exécuter en parallèle que des processus de longue durée, en nombre comparable au nombre de processeurs de la machine hôte, et aussi indépendants que possible pour limiter les communications.

Principe des systèmes multiséquentiels

Le principe des systèmes multiséquentiels est de ne s'exécuter en parallèle que s'il se présente une possibilité d'exécution parallèle *et* qu'un processeur au moins est *inoccupé*. Les possibilités d'exécution parallèle correspondent à des parties importantes et indépendantes des problèmes. Elles peuvent être indiquées par le programmeur à moins qu'il ne soit possible de les détecter statiquement à la compilation. Un programme ne comportant pas de parallélisme est exécuté séquentiellement avec une efficacité proche de celle des systèmes Prolog séquentiels les plus efficaces. C'est pourquoi les mécanismes d'exécution des systèmes multiséquentiels sont basés sur la *WAM*. La recherche de travail et la création d'activités parallèles sont à la charge des processeurs oisifs, afin de ne pas perturber l'exécution des processeurs

actifs. Ceux-ci ne font "qu'offrir" du travail s'il se présente une possibilité d'exécution parallèle. Les systèmes OU-parallèles existants diffèrent suivant la gestion des résolvantes multiples ainsi que les techniques d'initialisation des processus parallèles. Les systèmes ET-parallèles se distinguent suivant les techniques utilisées pour assurer la cohérence des liaisons des variables.

Gestion en pile et parallélisme

L'introduction du parallélisme dans une implémentation séquentielle introduit une composante largeur d'abord (*breadth first*) dans la stratégie en profondeur d'abord de Prolog. Cette modification de la stratégie d'origine a plusieurs conséquences indésirables sur la gestion des piles de la *WAM*. La première est la rupture de l'unicité de l'association entre l'adresse d'une cellule de variable de la pile et une variable logique. Ce problème provient de l'existence simultanée de plusieurs résolvantes et il est traité en détail dans le paragraphe 3.4.2. Les autres conséquences sont définies par [Hermenegildo 87] comme le *garbage slot problem*, apparition de trous noirs dans la pile, et comme le but inclus (*trapped goal problem*). Le premier problème apparaît lorsqu'une portion de pile inutilisée ne peut être récupérée car elle se trouve en milieu de pile (voir figure 3.1). Le second problème se produit lorsque la résolution du sous-but courant exige une allocation mémoire en milieu de pile. Par la suite nous n'aurons qu'à nous préocupper du premier problème qui a pour conséquence d'augmenter la consommation mémoire des systèmes logiques parallèles (voir section 8.6).

3.4.2 Parallélisme OU : gestion des résolvantes multiples

Introduction

Nous avons signalé que le principal problème du parallélisme OU est la gestion de résolvantes multiples. Dans le cadre des systèmes multiséquentiels, le critère utilisé pour choisir une solution est bien sûr l'efficacité. Les solutions existantes représentent des compromis entre d'une part le temps passé à initialiser une nouvelle activité parallèle et d'autre part l'efficacité des processus parallèles. Le choix d'une solution dépend également du multiprocesseur utilisé, certaines solutions étant destinées aux multiprocesseurs à mémoire partagée, d'autres étant définies pour des multiprocesseurs à mémoire privée et au coût de communication élevé.

Les modèles visant à l'utilisation optimale des multiprocesseurs à mémoire commune partagent entre processus, autant que possible, les piles des environnements de liaisons des variables, conduisant à des piles cactus (*cactus stack*). Ce partage peut être également envisagé sur un multiprocesseur à mémoire distribuée, si le modèle de calcul ne nécessite pas de fréquentes communications interprocessus. Le reste de cette section examine les problèmes que pose la gestion des variables logiques dans une pile cactus. Une solution alternative consistant à créer une nouvelle copie de la résolvante sera considérée ultérieurement lors de la présentation

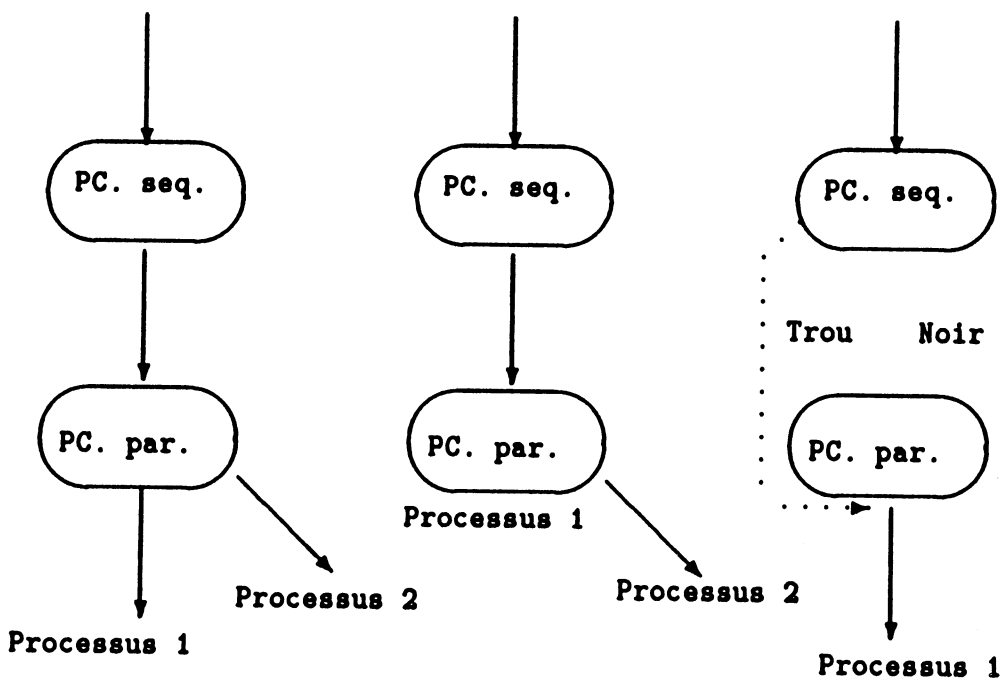

Figure 3.1: Exemple de trou noir dans la pile, en parallélisme OU

Dessin de gauche : le processus 1 a créé un point de choix parallèle qui a été exploité par le processus 2. Dessin du milieu : au retour arrière sur ce point de choix, le processus 1 ne trouve plus d'alternative à exécuter et exécute une alternative du point de choix séquentiel précédent. La portion de pile comprise entre les deux points de choix doit être conservée car elle est utilisée par le processus 2. Dessin de droite : lorsque le processus 2 termine, cette portion de pile constitue un trou noir non récupérable en l'absence de ramasse-miettes. Une tentative ultérieure d'utilisation de l'espace du trou noir avant qu'il n'apparaisse en sommet de pile est susceptible de provoquer un problème de but inclus.

du modèle Kabu-Wake. Enfin, nous mentionnerons brièvement d'autres modèles, qui évitent la recopie ou le partage de résolvante au prix de calculs redondants.

Partage de la pile par plusieurs résolvantes

En Prolog, une variable logique est identifiée par une adresse de cellule de variable dans la pile (voir paragraphe 2.3.1). Elle est initialisée à libre lorsqu'elle est créée, puis éventuellement liée. La gestion de la pile et le retour-arrière garantissent l'unicité de l'association entre cellule de variable et variable logique, cette dernière étant identifiée par l'adresse de la cellule de variable dans la pile. Cette unicité n'est plus garantie si plusieurs branches OU-parallèles sont initialisées entre la création et la liaison de la variable et que ces branches se partagent la pile locale. Le parallélisme OU remplace certains retour arrières par l'exécution simultanée de plusieurs alternatives. Une même cellule de variable sur la pile est alors susceptible de correspondre à plusieurs variables logiques distinctes. Chaque variable logique est gérée et éventuellement liée par l'une des branches OU-parallèles.

Comme on peut le voir dans l'exemple de la figure 3.2, le prédicat *index* a pour but d'instancier successivement *Nle_Ligne* à un entier compris entre 1 et *Taille*. Rappelons tout d'abord comment ces différentes liaisons sont gérées dans une implémentation séquentielle. Lors de l'invocation de la deuxième clause du prédicat *resoudre*, une cellule de variable libre est créée sur la pile pour représenter la variable *Nle_Ligne*. Plus tard, durant la résolution du but *index(Nle_Ligne, Taille)* la variable *Nle_Ligne* est liée à une constante. Cette liaison s'effectue en écrivant la valeur de la liaison (valeur de la variable *Taille*) dans la cellule de variable initiale (voir paragraphe 2.3.4) et en enregistrant l'adresse de cette cellule dans la pile de restauration (voir paragraphe 2.3.2). A l'exécution du prochain retour arrière, les variables dont l'adresse est enregistrée sur la pile de restauration sont réinitialisées à libre afin de détruire toutes les liaisons effectuées entre la création du dernier point de choix et le retour arrière sur ce point de choix. Les cellules de variables correspondantes peuvent ensuite être réutilisées pour stocker de nouvelles liaisons.

Le schéma de liaison utilisé dans l'exemple précédent est le plus efficace pour les implémentations séquentielles. Il est donc intéressant d'utiliser ce type de liaisons autant que possible dans un système OU-parallèle multi-séquentiel. Ce type de liaison est appelé **liaison superficielle** (*shallow binding*) dans un système de programmation logique parallèle.

Liaison profonde

Dans un système OU-parallèle, il est nécessaire de pouvoir lier *simultanément* à des valeurs distinctes, plusieurs variables logiques identifiées par la *même* adresse de cellule de variable. C'est le cas dans l'exemple de la figure 3.2 pour les variables logiques identifiées par l'adresse de la cellule *Nle_Ligne*. Leur nombre est compris entre *un* (exécution séquentielle) et *Taille* s'il y a suffisament de processeurs libres. *Pour ce faire, un autre mécanisme de liaison généralement appelé* **liaison profonde** *(deep binding) est utilisé.* Dans une

```
reines( Taille, Solution ) :-
        resoudre( Taille, [], Solution, 0 ).

resoudre( Taille, L, L, Taille ).
resoudre( Taille, Courant, Final, Colonne ) :-
        Colonne < Taille,
        Nle_Colonne is Colonne + 1,
        index( Nle_Ligne, Taille ),
        sur( Courant, Nle_Colonne, Nle_Ligne ),
        resoudre( Taille, [s(Nle_Colonne,Nle_Ligne)|Courant],
                        Final, Nle_Colonne ).

index( Size, Size ).
index( N, Size ) :-
        S1 is Size - 1,
        S1 > 0,
        index( N, S1 ).

sur( [], _, _ ).
sur( [s(I,J)|L], X, Y ) :-
        not(menace( I, J, X, Y )),
        sur( L, X, Y ).

menace( I, _, I, _ ) :- !.
menace( _, J, _, J ) :- !.
menace( I, J, X, Y ) :-
        ( U is I - J ),
        ( U is X - Y ),
        !.
menace( I, J, X, Y ) :-
        ( U is I + J ),
        ( U is X + Y ).
```

Figure 3.2: Le programme des N reines

Ce programme Prolog résoud, à l'appel de la requête *reines(N,S)*, le problème des N reines. *N* est instancié à la taille de l'échiquier et *S* est la solution qui est rendue sous la forme de la liste des coordonnées des reines (*[s(Colonne, Ligne)...]*). La $2^{ème}$ clause du prédicat *resoudre* est appelée récursivement pour chaque colonne. Pour chaque colonne une ligne est choisie par le prédicat *index* et la validité de ce choix est testée par le prédicat *sur*. Le parallélisme OU apparaît lors de l'exécution du prédicat *index* par lequel la variable *Nle_Ligne* peut être instanciée alternativement à toutes les valeurs comprises entre 1 et *N*.

liaison profonde, le couple (adresse de la variable, valeur de la liaison) est stocké là où est effectuée la liaison , c'est à dire dans la pile gérée par le processeur qui fait la liaison. Ce mécanisme est plus coûteux que la liaison superficielle mais il est indispensable pour associer plusieurs variables logiques à une même cellule de variable. Les figures 3.3 et 4.3 illustrent deux mécanismes différents de liaison profonde utilisés dans les modèles SRI et PEPSys.

Liaisons conditionnelles et universelles

Une autre façon de voir ce problème est de distinguer deux types de liaisons possibles : les **liaisons conditionnelles** et les **liaisons universelles**. Lorsqu'une cellule de variable est créée puis liée avant que ne soit rencontrée une alternative, cette liaison est qualifiée d'universelle. En effet, quelles que soient les branches de calcul à avoir connaissance de cette variable, elles sont toutes issues d'alternatives plus récentes que la liaison de la variable, et donc partagent toutes la valeur de cette liaison. L'unicité de l'association entre cellule de variable et variable logique est alors conservée. *Toute liaison universelle peut donc être faite de façon superficielle.* Au contraire, si une alternative est créée entre la création de la cellule de variable et sa liaison, la liaison est appelée conditionnelle. Chaque branche alternative associe une variable logique différente à la cellule initiale. Les liaisons conditionnelles doivent donc soit être faites de façon profonde, ce qui est le cas pour les modèles SRI (voir paragraphe 3.4.4) et ANL-WAM (voir paragraphe 3.4.4), soit offrir la possiblité de tester leur validité si elles sont faites de facon superficielle, ce qui est le cas du modèle PEPSys (voir paragraphe 4.4.2).

3.4.3 Parallélisme OU : contrôle de l'exécution

Le parallélisme OU en programmation logique consiste à faire exécuter simultanément certaines alternatives du programme par différents processeurs. Le premier problème de contrôle réside dans le moyen utilisé pour choisir les alternatives qui doivent être distribuées entre les processeurs. Comme il a été vu dans l'introduction sur les modèles multi-séquentiels, ce contrôle est à la charge du processeur libre. La seule contrainte pour le processeur qui crée une alternative est de faire en sorte que la structure de données qui représente cette alternative puisse être accédée par un éventuel processeur libre. La stratégie de choix d'une alternative par un processeur libre prend en compte plusieurs facteurs pour sélectionner un travail :

- on recherche en général le travail disponible offrant la plus forte granularité. Une heuristique communément utilisée consiste à sélectionner le travail disponible le plus proche de la racine de l'arbre de réfutation.

- la recherche de travail tente d'éviter la création de trous noirs. Il est par exemple possible de suspendre un processus qui fait un retour arrière sur un point de choix parallèle (le processus 1 dans le dessin central de la figure 3.1).

Cette solution ayant l'inconvénient de limiter l'exécution parallèle on peut aussi limiter la recherche aux processus issus du point de choix parallèle (le processus 2 dans le même dessin). C'est la stratégie retenue dans le scheduler originel de PEPSys (voir paragraphe 6.5).

- si nécessaire, la recherche de travail sélectionne celui qui a le meilleur comportement relativement au modèle de gestion des liaisons. C'est le cas dans le modèle SRI (voir paragraphe 3.4.4) où un travail est cherché "le plus près possible" de la position du processeur qui cherche du travail dans l'arbre de réfutation, pour limiter le nombre de liaisons profondes à recopier (voir ci-dessous).

Le problème de la détermination d'une "bonne" stratégie fait actuellement l'objet d'un certain nombre de recherches ([Brand 88a], [Calderwood 89] et [Butler et al. 88]).

Après la détermination d'une alternative susceptible d'être traitée en parallèle se pose le problème de l'accès aux données de cette alternative. Dans les systèmes séquentiels les informations nécessaires à cet accès sont stockées, lorsque l'alternative est rencontrée, dans un point de choix. Les arguments de l'appel provenant du point de choix sont utilisés pour effectuer l'unification du but courant avec la clause choisie. Lors du retour-arrière, les arguments de l'appel et la référence vers l'environnement courant sont restaurés. Dans un système parallèle des informations analogues à celles se trouvant dans un point de choix doivent être transmises au processeur se chargeant d'une alternative et en particulier les valeurs des registres arguments à l'appel du prédicat. Cependant la gestion simultanée de plusieurs résolvantes par des branches parallèles implique la transmission d'autres informations dont la nature et la quantité diffèrent suivant le modèle utilisé :

- soit l'ensemble de la pile, à l'exception des liaisons conditionnelles, est transmis au nouveau processus. C'est la solution retenue par le modèle Kabu-Wake décrit au paragraphe 3.4.4.

- soit l'ensemble des liaisons profondes est transmis au nouveau processus, dans le cas où les liaisons conditionnelles sont faites de façon profonde. C'est le cas du modèle SRI (voir paragraphe 3.4.4).

- soit seulement l'information permettant de déterminer la validité des liaisons conditionnelles est communiquée au nouveau processus. C'est le cas du modèle PEPSys qui transmet la "date" de création de l'alternative d'où est issue le nouveau processus (appelée *split-OBL* dans le paragraphe 4.4.2).

3.4.4 Systèmes OU-parallèles

Après avoir présenté les problèmes posés par l'implémentation du parallélisme OU dans les sytèmes de programmation logique nous allons montrer comment ces problèmes peuvent être résolus en décrivant les modèles mis au point par plusieurs

équipes. La liste des recherches présentées ici n'est certainement pas exhaustive mais elle met en évidence celles qui proposent les plus intéressants résultats ou promesses. Le modèle Kabu-Wake procède par création d'une nouvelle copie de résolvante dans chaque branche parallèle, tandis que le modèle SRI optimise la gestion d'une pile cactus pour un multiprocesseur à mémoire partagée. Le modèle PEPSys, qui sera présenté dans le chapitre suivant, gère aussi une pile cactus, mais sans faire de supposition sur l'architecture du multiprocesseur cible ; l'initialisation d'une nouvelle activité y est plus simple que dans le SRI-modèle, au prix d'un accroissement de complexité des opérations de déréférencement et unification. Les autres modèles présentés dans ce chapitre évitent les inconvénients des précédents au prix de calculs redondants dans les branches parallèles.

Le modèle Kabu-Wake

Le modèle Kabu-Wake développé entre 1984 et 1986 dans les laboratoires Fujitsu au Japon est l'une des premières implémentations OU-parallèles de Prolog.

Mécanisme de liaison

Chaque liaison est étiquetée avec une *date* qui est le nombre de points de choix créés depuis le début de la branche.

Mécanisme de déréférencement

Le déréférencement des variables dans le modèle Kabu-Wake est identique au déréférencement en Prolog.

Installation d'une tâche

L'opération de transmission d'une tâche est assez coûteuse. Lorsqu'une tâche disponible est trouvée par un processeur libre, le processeur qui a créé cette tâche est interrompu, il suspend son exécution séquentielle et passe en mode d'exécution "Kabu-Wake". Dans ce mode il recopie tout l'historique de l'exécution précédant la création de la tâche demandée. Cet historique est constitué par le contenu des différentes piles entre leur fond et leur sommet telles qu'elles étaient lors de la création de la tâche considérée. Durant cette copie toutes les variables liées dont l'étiquette de date est supérieure à la date du point de choix constituant la tâche sont réinitialisées à *libre*. Cette opération est équivalente aux déroulement de la pile de trace qui se passe normalement au cours du retour-arrière. Elle est, d'après les auteurs du modèle, plus efficace. Cette copie est transmise au processeur libre qui la réinstalle dans sa propre pile. Le processeur qui prend la tâche se comporte alors comme l'aurait fait celui qui la donne après une phase de retour arrière. On peut trouver une description du modèle Kabu-Wake dans [Sohma et al. 85].

Expérimentations et résultats

Un multi-processeur expérimental a été construit. Il s'agit d'un système comportant 16 processeurs élémentaires (Motorola 68010) et deux réseaux de communication : le réseau de contrôle et le réseau de données. Les chiffres, indiqués dans [Masuzawa 86], montrent des résultats intéressants pour des programmes à grands espaces de recherche. Les programmes utilisés sont, d'une part, une résolution du problème des 8-9-10 reines et, d'autre part, un analyseur syntaxique appliqué à

plusieurs phrases complexes. On note un gain de performance de 11 obtenu avec 12 processeurs pour l'analyse de la phrase la plus complexe. En effet, dans ces cas l'arbre de recherche est composé de branches nombreuses et de grande taille. On conçoit donc assez bien que tous les processeurs soient vite occupés à effectuer des calculs totalement indépendants.

Il faut cependant noter que le système est basé sur un interpréteur séquentiel dont la vitesse sur un processeur est modeste (environ 0,5 Klips) par rapport à ce qui est possible avec les techniques de compilation. Il est donc à craindre qu'en augmentant les performances de l'implémentation séquentielle il ne soit pas possible d'obtenir d'aussi bons gains de performance. En effet, il est vraisemblable que le mécanisme de transmission des tâches ne puisse pas être optimisé de la même façon, et donc que le surcoût qui lui est associé devienne trop importante.

Le modèle SRI

Le modèle SRI [Warren 87b] a été mis au point par D. H. D. Warren quand il était à SRI International en 1983. Cette étude est actuellement poursuivie dans le cadre du projet "Gigalips". Il est évidemment basé sur la WAM de telle sorte que les principales zones de données de la WAM ([Warren 83]) se retrouvent ici.

Mécanisme de liaison

Chaque processeur virtuel (*worker*) est une instance de la machine abstraite modifiée. Il possède, en plus des zones de données classiques, un *tableau de liaisons* (*binding array*) qui est utilisé pour enregistrer les liaisons profondes. Toutes les liaisons conditionnelles sont effectuées en utilisant le mécanisme de liaison profonde alors que les liaisons universelles utilisent le mécanisme de liaison superficielle. Les liaisons profondes sont enregistrées sur la pile de trace (voir figure 3.3). Le surcoût en temps d'exécution du à une liaison profonde est 40% par rapport à un système purement séquentiel.

Mécanisme de déréférencement

Une cellule de variable liée de façon conditionnelle contient un index dans le tableau de liaisons. Lorsqu'une variable est liée de façon conditionnelle la valeur de la liaison est stockée dans le tableau de liaisons du processeur virtuel qui effectue la liaison en utilisant l'index trouvé dans la cellule de variable. Lors du déréférencement d'une variable, sa cellule est accédée dans la pile du processus créateur : si elle est liée universellement la valeur de la liaison est enregistrée dans la pile, sinon l'index trouvé dans la cellule est utilisé pour accéder au tableau de liaisons local. Ce mécanisme, très simple comparé à d'autres modèles, provoque un surcoût de 60% pour l'opération de déréférencement par rapport à un système purement séquentiel.

Installation d'une tâche

Les processeurs virtuels peuvent se déplacer dans l'arbre de recherche, en défaisant les liaisons conditionnelles, enregistrées dans leur tableau de liaisons, lorsqu'ils vont vers la racine de l'arbre et en les réinstallant lorsqu'ils vont vers les feuilles. Une réinstallation consiste à recopier le tableau de liaisons et la pile

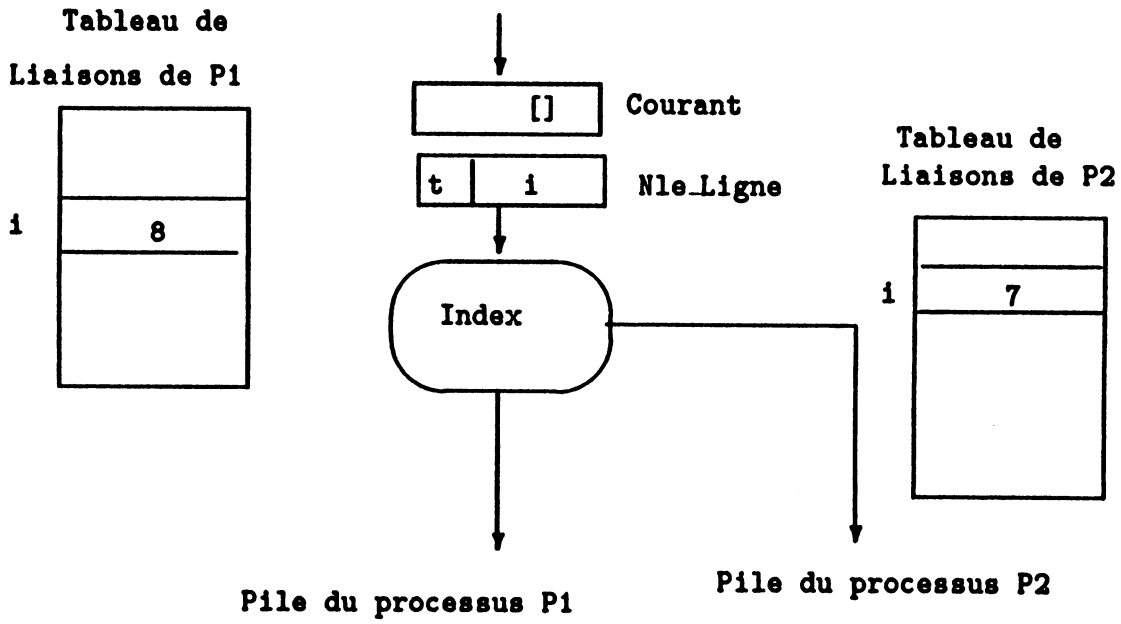

Figure 3.3: Liaisons multiples dans le modèle SRI

Cette figure illustre l'accès aux variables *Courant* et *Nle_Ligne* du programme des 8-reines de la figure 3.2. Le processus P1 a commencé l'execution du programme. Un processus P2 a été créé pour exécuter la seconde clause du prédicat *index*. Lors de l'exécution du prédicat *sur* par P1 et P2, la variable *Courant* est liée inconditionnellement à *[]* pour les deux processus, tandis que la variable *Nle_Ligne* est liée à 8 pour P1 et à 7 pour P2. Les deux liaisons sont enregistrées au même index i des Tableaux de Liaisons de P1 et P2.

de restauration de la branche de l'arbre d'évaluation du programme sur laquelle une alternative a été trouvée. Il s'agit d'une recopie incrémentale c'est-à-dire que seules les différences entre les tableaux de liaisons du processeur qui prend du travail et de celui qui en fournit sont recopiées. L'exécution efficace d'un programme parallèle suppose un ordonnanceur suffisamment intelligent pour sélectionner la tâche qui minimise le nombre de liaisons à défaire et à réinstaller [Brand 88a] [Calderwood 89] [Butler et al. 88].

Expérimentations et résultats

Le modèle SRI a été implémenté dans le cadre du groupe de travail Gigalips en modifiant une implémentation efficace de Prolog [Carlsson 87], donnant naissance au système Prolog parallèle Aurora[Lusk et al. 88]. Les performances de ce système (voir table 3.1), en font l'une des plus efficaces implémentations Prolog OU-parallèle existant à ce jour. Les résultats présentés dans la table 3.1 sont obtenus en calculant l'ensemble des solutions aux programmes considérés. *Hamilton* est la recherche d'un chemin hamiltonien dans un graphe, *mandel* est le calcul d'un ensemble de mandelbrot (300 points), *queens1* est le programme des huit reines, dans une version plus efficace que celle de la figure 3.2, *saltm* est un puzzle de Lewis Carroll et enfin, *TInA3* est une application écrite à ECRC en utilisant le langage PEPSys, générant des circuits touristiques en utilisant la base des données de l'office du tourisme bavarois ; le test mesuré ici, utilise un sous-ensemble de la base de données, comporte plusieurs centaines de clauses PEPSys (voir section 7.4 pour une description des programmes de test).

Autres systèmes OU-parallèles

ANL-WAM

Développé par "Argonne National Laboratory" (USA), ce système de programmation logique parallèle utilise le parallélisme OU [Butler 86,Disz 87]. Il est basé sur une exécution séquentielle efficace utilisant un code compilé étendu à partir de la WAM. Le mécanisme de gestion des liaisons utilise une gestion mémoire conçue pour les multiprocesseurs à mémoire commune. Toutes les liaisons conditionnelles et universelles faites dans les cellules de variables partagées utilisent le mécanisme de liaison profonde dans des *hash-windows* (voir la section 4.4.2 pour une définition de *hash-window*). Celles-ci sont donc largement plus utilisées que dans le modèle PEPSys ce qui se traduit par des performances "pathologiques" pour certains programmes.

L'ANL-WAM a constitué la première implémentation efficace d'un système Prolog parallèle, sur plusieurs multi-processeurs commerciaux tels que le Sequent Balance ou le Encore Multimax et a été largement utilisée à des fins expérimentales. Des résultats sur l'ANL-WAM sont présentés dans [Disz 87]. Dans les programmes testés, le gain de performance peut atteindre plus de 11 en utilisant 16 processeurs. La vitesse des processeurs séquentiels est cependant faible, de l'ordre de 1 Klips, à cause du manque de performances du processeur (NS32032) d'une part et parce que l'implémentation séquentielle n'est pas optimale d'autre part.

Table 3.1: Performances du système Aurora 0.0-Manchester Scheduler, sur Sequent Balance 8000

Programme	Aurora(1)	Aurora(4)	Aurora(8)	Quintus Sun 3/50
Hamilton	266,4 s.	68,4 s.	35,8 s.	21,1 s.
Mandel	39,2 s	12,7 s.	8,3 s.	13,2 s.
Queens1	46,9 s.	11,7 s.	5,9 s.	3,5 s.
Saltm	6,9 s.	1,8 s.	1,0 s	0.5 s,
TInA3	103,2 s.	26,2 s.	13,7 s.	9,5 s.

L'implémentation de Aurora est basée sur le système SICStus qui est un émulateur de WAM codé en C. Le Sequent Balance 8000 est un multiprocesseur à mémoire commune dont le processeur est un NS 32032 d'une efficacité d'environ 0,5 Mip. Quintus Prolog est l'un des plus efficaces systèmes Prolog séquentiel existant sur le marché. L'implémentation de Quintus Prolog est un émulateur de WAM écrit en assembleur. Le processeur du Sun 3/50 est un Motorola 68020, cadencé à 16 MHz et délivrant 1,5 Mips.

BC Machine

La BC Machine [Ali 88] est une architecture combinant des mémoires privées, où sont stockées les piles Prolog, avec une mémoire partagée, utilisée par le contrôle de l'exécution. Le point clé de cette architecture est son réseau d'interconnection qui permet à un processeur actif (maitre) de diffuser efficacement les liaisons qu'il effectue vers les mémoires privées de processeurs inactifs (esclaves) qui lui sont attachés. Lorsqu'une alternative est créée par un maitre, elle est allouée à un esclave. La transmission de tâches est donc extrêmement efficace, de même que les opérations de liaison et de déréférencement. Lorsque plusieurs processeurs deviennent inactifs, l'un des processeurs actifs est interrompu et diffuse l'état de ses piles aux processeurs inactifs qui deviennent ses esclaves. Un prototype de neuf processeurs est en cours de construction. Les résultats préliminaires, portant sur l'exécution de puzzles de grandes tailles sur sept processeurs, montrent d'excellentes performances.

Delphi

Le modèle Delphi [Clocksin 87] réduit les surcoûts habituels des systèmes Prolog OU-parallèles, en effectuant des calculs redondants. Les parties communes des branches parallèles sont recalculées par chaque branche, évitant ainsi les problèmes classiques d'accès aux cellules des variables protégées. L'allocation de travail aux processeurs se fait en utilisant des "oracles" qui sont des chemins à parcourir dans l'arbre d'évaluation du programme. Une implémentation expérimentale de ce modèle a été faite sur un réseau de SUNs [Alshawi 88] et donne des résultats

encourageants en dépit de la lenteur du réseau d'interconnection. Une autre implémentation de ce modèle a été faite par Clocksin sur un multiprocesseur à mémoire partagée, et des résultats encourageants ont été signalés dans le *newsgroup* Prolog du réseau *Usenet*.

3.4.5 Parallélisme ET-restreint

Dans le *Restricted AND-parallelism* [DeGroot 84], la compilation du programme génère des tests simples qui permettront, à l'exécution, de déterminer si des sous-buts sont indépendants ou non. En cas de doute, les sous-buts sont exécutés séquentiellement. Un exemple simple ([DeGroot 84]) est la clause :

$$f(X) \leftarrow p(X), q(X)$$

Il est clair que si, lors de l'exécution de cette clause, la variable X est instanciée à un terme clos (*ground*), les sous-buts p(X) et q(X) peuvent être exécutés en parallèle. Par contre, si X n'est pas liée lors de l'appel de f, l'exécution parallèle de p et q risque de provoquer des instanciations conflictuelles de la variable X. DeGroot suggère donc d'insérer dans cette clause ce test : *si, à l'appel de f, X est clos, exécuter en parallèle p(X) et q(X) ; sinon exécuter ces deux sous-buts séquentiellement.*

La génération statique de tests d'indépendance peut être simple, comme dans l'exemple précédent. Elle est plus complexe dans cet autre exemple de [DeGroot 84] :

$$f(X) \leftarrow p(X), q(X), s(X)$$

Trois graphes d'exécution différents sont alors possibles (figure 3.4).

Différents graphes statiques ont été proposés [DeGroot 84], [DeGroot 85], [Hermenegildo 86]. Ces propositions étant équivalentes, nous présentons l'une d'entre elles [DeGroot 84]. Dans ce modèle, six types d'expressions exécutables sont produites par le compilateur :

- *G* : G est un sous-but unique.
- *(SEQ E1 ... En)* : les sous-buts Ei doivent être exécutés séquentiellement.
- *(PAR E1 ... En)* : les sous-buts Ei peuvent être exécutés en parallèle.
- *(GPAR(X1, ..., Xk) E1 ... En)* : si les variables Xi sont instanciées à des termes clos, les sous-buts Ei peuvent être exécutés en parallèle.
- *(IPAR(X1, ..., Xk) E1 ... En)* : si les variables Xi sont indépendantes deux à deux, les Ei peuvent être exécutés en parallèle.
- *(IF E1 E2 E3)* : si l'expression booléenne E1 est vraie, alors exécuter E2, sinon E3.

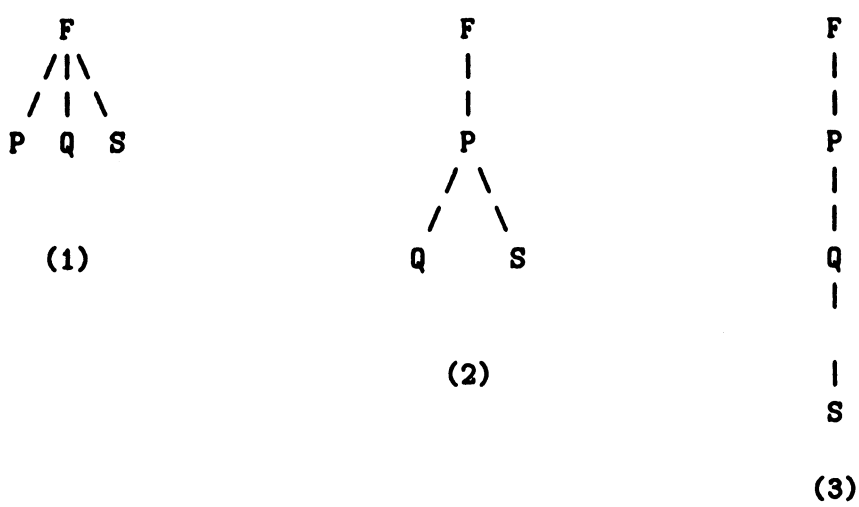

Figure 3.4: Graphes d'exécutions statiques

Dans le cas (1), X est close à l'appel de f, ce qui permet d'exécuter les trois sous-buts en parallèle. Dans le cas (2), la variable X, n'est pas close à l'appel de f mais est liée à un terme clos par l'exécution de p, ce qui permet de lancer l'exécution de q et s en parallèle. Dans le troisième cas enfin, X n'est pas close après l'exécution de p et q et s doivent donc être exécutés séquentiellement.

La clause de l'exemple précédent sera compilée selon le graphe suivant :

```
f(X) = (GPAR(X)
        p(X)
        (GPAR(X) q(X) s(X)))
```

Les tests à l'exécution générés par le *Restricted And Parallelism* devraient, selon ses auteurs, être très rapides. Si l'on prend l'exemple du test *GPAR*, c'est vraisemblablement le cas pour des termes simples, mais si X est un terme complexe, il peut se révéler coûteux de tester s'il est clos.

A supposer que, contrairement aux méthodes dynamiques, le surcoût introduit par les tests à l'exécution des graphes de dépendance statique soit faible, ces méthodes présentent un inconvénient notable. L'analyse statique d'indépendance des sous-buts échoue souvent pour des sous-buts qui pourraient être effectivement exécutés en parallèle. Un exemple cité par [DeGroot 84] est celui du programme *quicksort* pour lequel l'analyse statique ne décèle le parallélisme ET du programme que s'il a été écrit en pensant à cette analyse... Si le programme est :

```
quicksort([X | L], SL, Acc) :- partition(L, X, L1, L2),
                               quicksort(L1, SL, [X | SL2]),
                               quicksort(L2, SL2, Acc).
```

le compilateur ne détecte pas le parallélisme ET entre les deux appels récursifs à quicksort. Ce parallélisme est par contre détecté dans le programme :

```
quicksort([X | L], SL, Acc) :- partition(L, X, L1, L2),
                               quicksort(L1, SL, [X | SL2bis]),
                               quicksort(L2, SL2, Acc)
                               SL2 = SL2bis.
```

Des méthodes d'analyse plus fines, telles que [Winsborough 88] ont été conçues pour résoudre ce type de problème.

Retour arrière semi-intelligent

Dans les systèmes ET-parallèles, l'information d'indépendance entre littéraux, nécessaire au contrôle de l'exécution parallèle, peut également être utile pour contrôler le retour-arrière et permettre un gain d'étapes de calculs. Notons cependant que ce retour-arrière est moins précis que celui des méthodes de retour-arrière intelligent pour Prolog comme [Codognet 88], d'où sa qualification de "semi-intelligent". [Hermenegildo 86b] décrit un tel mécanisme pour la machine abstraite basée sur le parallélisme ET restreint [Hermenegildo 86a], décrite dans la section suivante.

3.4.6 Systèmes ET-parallèles

APEX : AND-Parallel EXecution

Le modèle mentionné au paragraphe 3.1.2 [Lin 88] a été implémenté sur un multiprocesseur à mémoire commune Sequent Balance 21000, version améliorée du Balance 8000, utilisant un bus et un cache plus efficaces et permettant de supporter jusqu'à 21 processeurs de type NS32032. Une extension de la WAM a été définie pour supporter le modèle. Les instructions de cette WAM étendue sont exécutées par un émulateur codé en C. Les performances de ce système montrent des gains de performances importants dûs au parallélisme, pour une implémentation déjà efficace séquentiellement.

An Abstract Machine for Restricted AND-Parallelism

Ce système est une extension de la *WAM* supportant le *Restricted And Parallelism* dans un Prolog multi-solutions efficace [Hermenegildo 86]. Hermenegildo suppose que le programme Prolog a déjà été compilé en un graphe de dépendances statique qu'il appelle CGE (*Conditionnal Graph Expression*). La Machine Abstraite reprend le modèle à trois piles de la *WAM*. Les extensions permettent à un processeur d'offrir du travail aux autres processeurs lors de l'exécution parallèle d'une conjonction de buts indépendants et de se synchroniser avec ces processeurs.

La stratégie de recherche de travail [Hermenegildo 87] tient compte de la gestion mémoire en pile de la WAM et cherche à bénéficier de la même récupération mémoire dans les piles locales et globales de la WAM, lors du retour-arrière. Un algorithme d'étiquetage des sous-buts est défini afin qu'un processeur cherchant du travail ne "vole" pas un sous-but susceptible de perturber la gestion en piles. Ce système implémente également le retour-arrière "semi-intelligent" [Hermenegildo 86b].

Une évaluation du système montre que des gains de performances importants peuvent être attendus de l'exécution en parallèle des programmes sur un multiprocesseur, par rapport à l'exécution séquentielle des mêmes programmes. Il faut cependant noter qu'il ne s'agit que de résultats de simulation portant sur de "petits" programmes, aucune implémentation effective n'ayant été réalisée sur multiprocesseur.

3.4.7 Modèles globaux

Les modèles intégrant parallélisme OU et parallélisme ET visent à exploiter le parallélisme fourni par les programmes tant déterministes qu'indéterministes. L'intégration des deux parallélismes au sein d'un même modèle pose des problèmes de contrôle de l'exécution, afin d'obtenir l'ensemble des solutions. Plusieurs niveaux d'intégration sont envisageables :

- **parallélisme ET déterministe dans un environnement OU-parallèle** : seuls les sous-buts déterministes, c'est-à-dire ne produisant qu'une seule

```
and_par(X, Y) :- findall(I-Z, solve(I, X, Y, Z), List),
                 member(1-X, List),
                 member(2-Y, List).

solve(1, X, _, X) :- call(X).
solve(2, _, Y, Y) :- call(Y).

member( X, [X | Xs] ).
member( X, [Y | Ys] ) :- member( X, Ys).
```

Figure 3.5: Compilation du parallélisme ET en parallélisme OU

Dans cet exemple *and_par* est un prédicat d'ordre supérieur appelé avec des prédicats ET-parallèles comme arguments. Ces prédicats sont en fait exécutés en parallélisme OU par le truchement du prédicat OU-parallèle *solve*. L'ensemble des solutions aux deux prédicats est enregistré dans la liste *List*. Les deux appels à *member*, dans lesquels les arguments des prédicats *X* et *Y* ne sont pas instanciés, permettent d'extraire tous les couples de solutions aux prédicats *X* et *Y*. Chaque appel à *member* est lui-même exécuté en parallélisme OU, ce qui correspond au produit croisé entre parallélisme ET et parallélisme OU (comparer avec la description du produit croisé dans PEPSys au paragraphe 4.4.1).

solution, sont exécutés en ET parallèle. C'est la façon la plus simple de tirer parti du parallélisme des programmes déterministes.

- **transformation du parallélisme ET en parallélisme OU** : il s'agit d'une idée simple permettant la combinaison de solutions multiples produites par deux sous-buts ET-parallèles et dont l'exécution de chacun génère du parallélisme OU. Les solutions de chacun des sous-buts sont collectées dans une liste par un prédicat *findall(X, G, L)* défini par [Clocksin et Mellish 81] comme construisant une liste L de tous les objets satisfaisant le but G. La continuation des sous-buts ET-parallèles constitue alors l'ensemble des combinaisons possibles entre solutions, en extrayant les solutions des listes. Il est ainsi possible de *compiler* le parallélisme ET en parallélisme OU de la façon décrite par [Carlsson 88a] (voir figure 3.5).

  Cette méthode présente plusieurs inconvénients : tout d'abord, l'exécution du prédicat *findall* est en général inefficace ; de plus elle crée un goulot d'étranglement du parallélisme : si les buts ET-parallèles génèrent du parallélisme OU durant leur exécution, il est nécessaire d'attendre la fin de l'exécution de chacune des branches (chacune des clauses de *resoudre*) pour exécuter la continuation (prédicats *member* et sous-buts suivants le sous-but *Y* dans la résolvante). Quelques expériences limitées d'utilisation de ce type de parallélisme sur l'implémentation de PEPSys sont décrites au chapitre 8.

Ce goulot d'étranglement a été confirmé par l'analyse du parallélisme dans une application écrite en Prolog [Ing 87a]. Pour remédier à cet inconvénient, [Carlsson 88a] suggère de recopier les solutions des sous-buts ET parallèles au fur et à mesure de leur production, dans une zone de mémoire permanente (tas Prolog), d'où ces solutions seraient alors consommées par les sous-buts *continuation*. Si cette solution supprime le goulot d'étranglement de la solution précédente, elle conserve cependant l'inconvénient de nécessiter la recopie des solutions, ce qui peut être coûteux pour les termes complexes. [Tick 89] montre qu'un système OU-parallèle implémentant le parallélisme ET de cette façon est moins efficace qu'un système purement ET-parallèle.

- **formation paresseuse du produit des solutions** : pour éviter les limitations des solutions précédentes, le but est ici de combiner les solutions des branches ET parallèles au fur et à mesure de leur production, mais aussi seulement en cas de besoin ; le besoin est ici celui de fournir du travail à un processeur inactif. Le contrôle de l'exécution est très complexe, si l'on veut garantir que l'ensemble des solutions au programme soit produit. C'est la solution retenue par le modèle PEPSys qui sera présenté au chapitre suivant.

- **parallélisme OU combiné au parallélisme ET dépendant** : dans le langage Andorra [Brand 88b], les sous-buts sont exécutés en parallélisme ET dès qu'ils deviennent déterministes, jusqu'à ce que tous les sous-buts soient non-déterministes. Dans ce cas le premier sous-but, - un ordre est maintenu par le système d'exécution entre les sous-buts à résoudre -, est choisi pour donner naissance à un point de choix éventuellement source de parallélisme OU. Selon ses auteurs, Andorra subsume à la fois Prolog et les Langages Gardés. Il faut toutefois noter que le modèle d'exécution de Andorra repose sur un compilateur capable d'analyser le déterminisme des programmes écrits en Andorra. Le langage Andorra est en cours d'implémentation dans le cadre du projet Gigalips. Le modèle SRI a été étendu afin de supporter le parallélisme ET déterministe et un interprète du modèle écrit en C a été développé sur un Sequent Balance 8000 [Yang 88]. Les résultats préliminaires indiquent des gains de performances linéaires en parallèle, sur de petits programmes purement ET-parallèles tels que *naive-reverse* et *hanoi*.

3.5 Conclusion : Quel parallélisme utiliser?

La sémantique déclarative des langages de programmation logique les rend particulièrement bien adaptés aux implémentations parallèles. Les systèmes de programmation logique parallèle se divisent en deux grandes familles. Les langages gardés, où le *Don't care non determinism* remplace le *Don't know non-determinism* de Prolog, visent à être les "assembleurs" des futures machines parallèles. Dans cette thèse nous nous intéressons plutôt aux modèles non-déterministes, et en particulier aux systèmes multi-séquentiels, qui visent à tirer parti dans les systèmes

parallèles des techniques les plus efficaces d'implémentation séquentielle.

Les premiers résultats obtenus par les prototypes d'implémentations OU-parallèles de Prolog indiquent que cette technique permet d'obtenir d'importants gains de performances, par rapport aux meilleurs systèmes Prolog séquentiels, pour plusieurs types de programmes non-déterministes. Ce point sera repris ultérieurement dans les chapitres consacrés aux mesures du système PEPSys et à leur interprétation.

Comme nous l'avons vu, le parallélisme ET est difficile à "extraire" automatiquement des programmes. Cependant, si l'on donne au programmeur la possibilité de l'exprimer, il semble présent dans de nombreuses applications (voir par exemple [Ing 87a]). Comparé au parallélisme OU, il pose néanmoins quelques problèmes :

- il est nécessaire de synchroniser les branches parallèles avant d'exécuter la continuation du programme ; de plus la continuation utilisant en général des variables instanciées par chacune des branches, le parallélisme ET ne sera sans doute efficace que dans les architectures multiprocesseurs à mémoire partagée.

- Le grain du parallélisme ET semble plus fin que celui du parallélisme OU ; en effet, le grain de parallélisme correspond à l'exécution d'un sous-but, alors que dans le parallélisme OU, il correspond à la séquence d'exécution du programme depuis la création du parallélisme jusqu'au retour-arrière sur le prédicat OU-parallèle.

L'intérêt du parallélisme ET est d'être présent dans de nombreux programmes et particulièrement des programmes déterministes. A ce titre, il semble indispensable à tout système logique parallèle désirant offrir des gains de performances pour de larges classes d'applications. Une comparaison expérimentale entre deux systèmes logiques ET- et OU-parallèles [Tick 89], utilisés pour résoudre les mêmes problèmes sur la même machine, confirme la difficulté à tirer parti efficacement du parallélisme ET alors même qu'il semble plus "abondant" dans les programmes logiques que le parallélisme OU.

L'utilisation combinée des parallélismes OU et ET semble nécessaire pour tirer pleinement parti du parallélisme de la programmation logique. La définition et l'implémentation d'un modèle multi-séquentiel incorporant ces deux formes de parallélisme est cependant très complexe ainsi que nous le verrons dans le chapitre suivant consacré au projet PEPSys. Il n'existe à notre connaissance aucune implémentation efficace d'un modèle incorporant les parallélismes OU et ET.

Chapitre 4

Présentation du projet PEPSys

4.1 Généralités sur le projet PEPSys

Le projet PEPSys (Parallel ECRC Prolog System) s'est déroulé à ECRC (European Computer-Industry Research Centre) de 1985 à 1988 [Baron, Chassin et al. 88]. Son but était d'explorer l'ensemble des problèmes liés à l'exploitation du parallélisme en programmation logique et d'y proposer des solutions. Les principales étapes de ce projet ont été les suivantes :

1985 : étude bibliographique de l'état de l'art en programmation logique parallèle, analyses statiques et dynamiques de programmes Prolog existants.

1986 : définitions d'un langage pour la programmation logique parallèle et d'un modèle de calcul supportant ce langage. Définition d'une machine abstraite permettant la compilation du langage PEPSys et supportant le modèle de calcul.

1987 : implémentation d'un compilateur du langage PEPSys. Implémentations d'un système parallèle sur un multiprocesseur à mémoire partagée et d'un simulateur d'une architecture extensible, tous deux basés sur la machine abstraite PEPSys. Réalisation d'un outil d'analyse du parallélisme dans les programmes PEPSys et implémentation d'applications dans le langage PEPSys.

1988 : mise au point du simulateur et du système parallèle. Expérimentations et mesures.

Nous ne reviendrons pas sur l'étude de l'état de l'art en programmation logique parallèle, qui a été faite au chapitre précédent. Une conséquence de cette étude a été, en raison des problèmes que posent les langages gardés, d'adopter l'approche non-déterministe et multi-séquentielle pour le langage et le modèle de calcul de PEPSys. Dans la suite de ce chapitre, nous présentons les différents constituants du projet, en insistant particulièrement sur le langage et le modèle de calcul, dont

les définitions ont conditionné le travail qui fait l'objet de cette thèse. Le lecteur interessé trouvera dans [Syre et al. 88] une présentation plus complète de l'ensemble du projet.

4.2 Analyses de programmes Prolog existants

4.2.1 Analyses statiques

Le but des analyses statiques de programmes Prolog [Ratcliffe et Robert 86] était d'une part d'obtenir des statistiques sur les programmes existants, d'autre part de tenter d'extraire automatiquement le parallélisme des programmes Prolog. Les analyses statiques de programmes Prolog ont donné les enseignements suivants :

- le nombre d'arguments d'un prédicat est en général faible, ce qui confirme le manque d'intérêt du parallélisme d'unification.
- le parallélisme OU est difficile à extraire des programmes. En effet, il n'est pas clair si l'ordre des clauses d'un prédicat est significatif ou non. De plus, les effets de bord et surtout le *cut* - largement utilisé dans les programmes Prolog - ont le rôle d'inhibiteurs de parallélisme.
- le parallélisme ET est également difficile à extraire des programmes, même si on se limite au parallélisme ET restreint. De plus, comme nous l'avons vu, il semble également difficile à exploiter efficacement (voir la section 3.4.5 du chapitre précédent).
- le dernier résultat de ces analyses statiques est plus empirique, puisqu'il s'agit de la constatation que le programmeur détecte "facilement" les sources de parallélisme dans un programme, là où un système d'analyse automatique ne saurait pas conclure. En outre la granularité du parallélisme, paramètre fondamental pour l'obtention de bonnes performances, semble très difficile à apprécier par un système automatique.

4.2.2 Analyses dynamiques

Les analyses dynamiques [Hailperin 85] ont permis d'établir que les programmes Prolog présentaient une bonne localité de référence à l'exécution. Cette propriété a largement influencé la gestion des résolvantes dans le modèle de calcul.

4.3 Le langage PEPSys

La définition du langage PEPSys [Ratcliffe et Robert 86] [Ratcliffe et Syre 87] a bénéficié des enseignements de l'étude des systèmes parallèles existants et des analyses de programmes Prolog. La définition du langage PEPSys a pour but

de fournir aux utilisateurs un langage aussi proche que possible de Prolog, mais leur permettant d'exprimer le parallélisme OU et ET de leurs programmes. Cet objectif place le langage PEPSys dans la catégorie non-déterministe, à la différence des langages gardés. Le langage PEPSys a pour objectif d'offrir l'ensemble des possibilités de Prolog, y compris les possibilités de contrôle et les effets de bord, à la différence de certains systèmes parallèles non-déterministes qui ne supportent que le langage "pur-Prolog", difficilement utilisable pour des applications "réelles".

4.3.1 Modularité

En plus des effets bénéfiques qu'elle apporte à l'ingénierie du logiciel, la modularité dans le langage PEPSys a pour objectif de séparer nettement les **modules séquentiels** des programmes, dans lesquels les effets de bord et le *cut* sont autorisés sans restriction, des **modules parallèles** dans lesquels les possibilités de contrôle et d'effet de bord seront plus limitées. Les liaisons entre modules se font au moyen des déclarations :

```
?- export([predicat-exporte / arite]).
?- import_from(nom-de-module, [predicat-exporte / arite]).
```

Les modules séquentiels supportent l'ensemble du langage Prolog, et sont exécutés en utilisant la même stratégie de résolution. A partir d'un module séquentiel, il n'est possible d'appeler un prédicat défini dans un module parallèle que de façon indirecte, à travers l'un des prédicats prédéfinis *oneof, bagof et setof*. Le premier fournit une solution quelconque au prédicat appelé, habituellement la première trouvée ; les autres fournissent l'ensemble des solutions aux prédicats, les doublons étant éliminés dans le cas du *setof*. Cette restriction à l'appel des prédicats des modules parallèles a pour but de "fermer" les appels aux modules parallèles, en ne laissant aucun point de choix à la fin de l'exécution du prédicat du module parallèle. Une autre conséquence est que les branches OU-parallèles se terminent toutes par un retour arrière.

Les parties des programmes destinées à s'exécuter en parallèle sont regroupées dans les modules parallèles. La stratégie de résolution n'est alors plus nécessairement en profondeur d'abord mais également parfois en largeur d'abord, lorsqu'un prédicat donne naissance à du parallélisme OU. Le contrôle est exprimé au moyen de **déclarations de propriétés** qui servent également au programmeur à exprimer le parallélisme OU de son algorithme. Un **opérateur d'indépendance**, noté #, sert à exprimer le parallélisme ET. Il est possible d'appeler d'autres modules parallèles depuis un module parallèle, par contre l'appel d'un module séquentiel n'est pas autorisé. Cette restriction complète la restriction d'appel des modules parallèles depuis les modules séquentiels, de telle sorte que les appels aux modules parallèles soient entièrement "fermés". Le *cut* et les effets de bords ne sont pas autorisés dans les modules parallèles.

4.3.2 Déclarations de propriétés

A l'intérieur du module parallèle, les clauses des prédicats doivent être regroupées et tout prédicat doit être précédé d'une déclaration de propriétés. Une déclaration de propriétés est un terme Prolog de la forme :

```
-properties( [solutions( {all, one} ), clauses( {ordered, unordered}),
              execution( {lazy, eager} ] ).
```

Propriété *solutions*

Cette propriété peut prendre deux valeurs, *all* qui signifie que le programmeur désire obtenir toutes les solutions du prédicat, ou *one*, qui signifie que seule la première solution du prédicat sera retenue. La propriété *solutions(one)* attachée à un prédicat est équivalente à un *cut* terminal à la fin de chacune des clauses du prédicat.

Propriété *clauses*

Les deux valeurs possibles pour cette propriété sont *ordered*, signifiant que l'ordre des clauses est significatif et *unordered* dans le cas contraire. Dans le premier cas, les solutions au prédicat doivent être produites dans l'ordre des clauses qui ont conduit à leur production, au prix d'un réordonnancement éventuel si le prédicat est exécuté en parallèle.

Propriété *execution*

Cette dernière propriété est une indication donnée par le programmeur au compilateur et au système d'exécution. Si elle prend la valeur *eager*, cela signifie que le programmeur estime que le prédicat peut être une bonne source de parallélisme OU. Cette propriété est utilisée par le compilateur pour générer des instructions permettant au système d'exécution de créer des activités OU-parallèles, pourvu qu'il y ait des ressources disponibles à l'appel du prédicat. La valeur *lazy* signifie que le programmeur ne souhaite pas que le prédicat soit utilisé comme "source " de parallélisme.

Expression de la stratégie de Prolog

La stratégie de résolution de Prolog correspond à la combinaison de propriétés suivantes :

```
-properties( [ solutions( all ), clauses( ordered ) ] ).
```

Expression du parallélisme OU

Pour déclarer qu'un prédicat est source de parallélisme OU, il faut le faire précéder de la combinaison de propriétés suivantes :

```
-properties( [ execution( eager ) ] ).
```

A priori la déclaration *clauses* est libre dans un prédicat OU-parallèle. Cependant, la conservation de l'ordre de solutions produites en parallèle est techniquement difficile à réaliser ; elle pose en outre un problème de performances puisque les solutions doivent être réordonnancées avant d'être retournées. Aussi, le compilateur PEPSys ne considère comme OU-parallèles que les prédicats comportant les déclarations :

```
-properties( [ clauses( unordered ), execution( eager ) ] ).
```

Conversion du *cut*

Les déclarations de propriétés rendent l'utilisation du *cut* inutile dans les programmes écrits en langage PEPSys. Par contre, il peut être utile de convertir des programmes Prolog existants en langage PEPSys. [Rapp 88] définit une méthode générale de conversion des *cut* en PEPSys. Soit un prédicat Prolog comportant un *cut* :

```
p(args) :- clause1.
p(args) :- clause2.
p(args) :- avant, !, apres.
p(args) :- clause3.
p(args) :- clause4.
```

Ce prédicat Prolog se traduit en trois prédicats PEPSys :

```
-properties( [ liste de proprietes de p ] ).
p(args) :- clause1.
p(args) :- clause2.
p(args) :- test_avant(Test), faire_reste(Test, args).

-properties( [ solutions(one), clauses(ordered) ]).
test_avant(true) :- avant.
test_avant(false).

-properties( [ liste de proprietes de p ] ).
faire_reste(true, args)  :- apres.
faire_reste(false, args) :- clause3.
faire_reste(false, args) :- clause4.
```

Exemple de programme OU-parallèle

La traduction en langage PEPSys du programme des n-reines de la figure 3.2 est assez immédiate. Elle est donnée à la figure 4.1.

4.3.3 Parallélisme ET

Opérateur d'indépendance

Le langage PEPSys offre au programmeur la possibilité de déclarer que deux sous-buts d'une clause sont indépendants, c'est-à-dire ne sont pas susceptibles de lier la même variable. Comme nous pouvons le constater dans le programme *quicksort* (voir figure 4.2), l'indépendance de deux sous-buts n'impose pas nécessairement qu'ils n'aient aucune variable en commun. De même que la déclaration *execution(eager)* pour le OU-parallélisme, l'opérateur d'indépendance ne génère pas obligatoirement du parallélisme ET à l'exécution. C'est plutôt une indication au compilateur pour générer des instructions qui permettront, si les ressources à l'exécution sont suffisantes, d'exécuter les deux sous-buts indépendants en parallèle. Les sous-buts exécutés simultanément peuvent eux-mêmes générer du parallélisme OU, ce qui impose alors un produit croisé des solutions de chaque sous-but. Dans les implémentations actuelles de PEPSys, aucune vérification de l'indépendance effective des sous-buts n'est faite à l'exécution. Une telle vérification semble cependant faisable efficacement en utilisant des techniques analogues à [Lin 88]. D'autre part, l'opérateur d'indépendance peut être considéré comme un moyen de suppléer à l'absence de technique de détection d'indépendance entre sous-buts à la compilation. Lorsque de telles techniques seront opérationnelles, elles rendront cet opérateur inutile.

Exemple de programme ET-parallèle

Le programme *quicksort* est un exemple de programme ET-parallèle. Comme nous le verrons dans le chapitre consacré aux mesures, sa parallélisation n'apporte qu'un gain de performances limité. Une version de ce programme en langage PEPSys est donnée à la figure 4.2.

4.3.4 Validation du langage PEPSys

Le langage PEPSys a été utilisé pour programmer des applications "réelles" [Ing 87a] et convertir des programmes Prolog existants comme CHAT80. Il semble qu'il remplisse actuellement les objectifs qui lui ont été assignés, à savoir permettre au programmeur d'exprimer, dans un langage proche de Prolog, le parallélisme de ses algorithmes. Les principes qui ont présidé à la définition du langage PEPSys semblent d'ailleurs avoir été "redécouverts" par [Butler et al. 88] qui recommande au programmeur d'applications parallèles :

```
/* Module sequentiel : fichier nreines.seq */

?- export( [reines/1] ).                    /* Point d'entree */
?- import_from( 'nreines.par', [reines/2] ).

reines( Taille ) :-
      bagof( Sol, reines( Taille, Sol ), Solutions ),
      writeln( Solutions ).

/* Module parallele : fichier nreines.par */

?- export( [reines/2] ).

-properties([solutions(all), clauses(unordered), execution(lazy)]).
reines( Taille, Solution ) :- resoudre( Taille, [], Solution, 0 ).

-properties([solutions(all), clauses(unordered), execution(lazy)]).
resoudre( Bs, L, L, Bs ).
resoudre( Taille, Courant, Final, Colonne ) :-
        Colonne \== Taille,
        Nle_Colonne is Colonne + 1,
        index( Nle_Ligne, Taille ),
        sur( Courant, Nle_Colonne, Nle_Ligne ),
        resoudre( Taille, [s(Nle_Colonne,Nle_Ligne)|Courant],
                        Final, Nle_Colonne ).

-properties([solutions(all), clauses(unordered), execution(eager)]).
index( Size, Size ).
index( N, Size ) :- S1 is Size - 1, S1 > 0, index( N, S1 ).

-properties([solutions(all), clauses(unordered), execution(lazy)]).
sur( [], _, _ ).
sur( [s(I,J)|L], X, Y ) :- not(menace( I, J, X, Y )), sur( L, X, Y ).

-properties([solutions(one), clauses(unordered), execution(lazy)]).
menace( I, _, I, _ ).
menace( _, J, _, J ).
menace( I, J, X, Y ) :- ( U is I - J ), ( U is X - Y ).
menace( I, J, X, Y ) :- ( U is I + J ), ( U is X + Y ).
```

Figure 4.1: Le programme des N reines écrit en PEPSys

Un seul prédicat génère du parallélisme OU : le prédicat *index*. Nous verrons dans le chapitre 8 que ce parallélisme est très important pour une taille supérieure ou égale à 8. Le *cut* terminal de la clause *menace* est remplacé par la propriété *solutions(one)*. Le *cut* du prédicat *resoudre* est remplacé par un test.

```
/* Module sequentiel : fichier quicksort.seq */

?- export( [quicksort/2] ).                    /* Point d'entree
*/
?- import_from( 'quicksort.par', [quicksort/3] ).

quicksort( Liste, Liste_triee ) :-
      oneof( quicksort( Liste, Liste_triee, [] ) ),
      write( 'La liste ' ), write( Liste ),
      write(' se trie en '), write( Liste_triee ).

/* Module parallele : fichier quicksort.par */

?- export( [quicksort/3] ).

-properties( [clauses(unordered), execution(lazy)] ).

quicksort([X | L], SL, Acc) :- partition(L, X, L1, L2),
                               quicksort(L1, SL, [X | SL2]) #
                               quicksort(L2, SL2, Acc).
quicksort( [], [] ).

-properties([solutions(one), clauses(unordered), execution(lazy)]).
partition( ... ) :- ...
partition( ... ) :- ...
partition( ... ) :- ...
```

Figure 4.2: Le programme quicksort écrit en PEPSys

Les deux appels à *quicksort* peuvent être exécutés en parallélisme ET bien qu'ils partagent la variable *SL2*. Ceci est possible parce que seul le second appel à *quicksort* instancie cette variable. Le programmeur sait que les sous-buts sont indépendants alors qu'un système d'analyse ne pourrait détecter l'indépendance statiquement.

- de "circonscrire " les "parties OU-parallèles" de l'algorithme en utilisant les prédicats *get_all* et *get_one* équivalents aux *bagof* et *oneof* de PEPSys. L'ordre des solutions ne doit pas être considéré comme important.
- de ne pas utiliser dans les "parties OU-parallèles" de *cut* ou d'effet de bords, car leurs implémentations dans un système parallèle ont pour effet de sérialiser l'exécution.

Le langage PEPSys doit être considéré comme expérimental. Des extensions au langage ont été considérées (voir par exemple [Ratcliffe et Syre 87]). Les déclarations de propriétés pourraient également être simplifiées si certaines combinaisons se révèlent inutiles.

4.4 Le modèle de calcul de PEPSys

Le modèle de calcul de PEPSys [Hailperin 86,Westphal et al. 87] a été défini simultanément au langage PEPSys dont il a pour but de permettre une implémentation efficace. Ce modèle définit la façon dont le parallélisme est contrôlé et les activités parallèles créées. Un des choix fondamentaux de ce modèle a été de retarder les calculs supplémentaires dus au parallélisme, source de surcoût, jusqu'au point où ils deviennent absolument nécessaires. Les principales caractéristiques du modèle sont les suivantes :

- intégration des parallélismes OU et ET et de la combinaison des deux avec la stratégie de résolution de Prolog séquentiel
- partage des résolvantes entre processus OU-parallèles, le mécanisme permettant le partage utilisant une datation explicite des liaisons superficielles.

4.4.1 Combinaison des parallélismes OU et ET avec l'exécution séquentielle et le retour arrière

Une des originalités du modèle PEPSys est la façon dont les parallélismes OU et ET sont combinés avec l'exécution séquentielle et le retour-arrière. Comme il a été mentionné dans les généralités sur les systèmes multiséquentiels, la recherche de travail et la création de processus sont à la charge des processeurs oisifs. Les processeurs actifs génèrent des activités potentielles ; s'il se trouve des processeurs oisifs, ces potentialités servent à générer du parallélisme ; dans le cas contraire, le travail correspondant sera exécuté par le processeur l'ayant proposé.

Bien que le langage ne donne pas de limite au nombre de sous-buts d'une clause pouvant être connectés par des opérateurs d'indépendance, le modèle de calcul ne considère que les combinaisons d'un seul côté gauche avec un seul côté droit, ce qui ne correspond qu'à la connection des sous-buts deux par deux. Nous supposerons donc dans tout ce qui suit que l'opérateur d'indépendance # est associatif à droite et que la notation : a # b # c est une simplification de : a # (b # c).

Parallélisme OU seul

Un processeur actif exécutant un prédicat générateur de parallélisme OU - qu'on appelera improprement **prédicat OU-parallèle** dans le reste de la thèse - créé une structure de données appelée *branch-point* qui :

- indique aux processeurs oisifs qu'il existe un travail possible à exécuter.
- contient les informations permettant aux processeurs oisifs de démarrer de nouveaux processus pour exécuter ces activités.

Si les activités potentielles ne trouvent pas "preneur", elles sont exécutées par le processeur "séquentiel" de la même façon qu'en Prolog séquentiel, lors d'un retour arrière. Le contrôle se complique lorsque le parallélisme se mélange au retour-arrière. En effet le modèle PEPSys assure le partage des environnements entre processus pères et fils. Si un processus père effectue un retour-arrière sur un *branch-point* servant de racine à un ou plusieurs processus, deux cas sont possibles :

- il reste au moins une alternative inexplorée dans le *branch-point* ; dans ce cas, le processus père exécute cette alternative.
- il ne reste aucune alternative inexplorée mais certaines alternatives sont encore en cours d'exploration par d'autres processeurs ; le processus père doit alors stopper son retour arrière car il ne lui est pas possible de libérer une section de la pile d'environnements partagée par le ou les processus fils. Plusieurs stratégies sont alors possibles pour le processus père afin de conserver le processeur actif et d'éviter la formation de trous noirs dans sa pile (voir paragraphe 3.4.1 et figure 3.1).
  - "attendre" que l'ensemble de ses fils ait fini de s'exécuter. Cette stratégie, la plus simple possible, a été mise en œuvre dans la première version du simulateur PEPSys puis abandonnée en raison de l'insuffisance de ses performances.
  - "aider" l'un de ses fils en lui "prenant" du travail. Le processus père recherche du travail dans ses fils tant que ceux-ci n'ont pas terminé. Si aucun de ses fils "n'offre" de travail, il s'agira d'une forme "d'attente active". Cette stratégie est celle qui a été définie avec le modèle [Hailperin 86] et mise en œuvre dans le simulateur et dans l'implémentation de PEPSys.
  - rechercher du travail dans l'ensemble des processus exécutant le programme. Comme l'explique [Hermenegildo 87], cette stratégie résulte en la création de **trous noirs** dans la pile d'environnements (voir section 3.4.1). De nombreuses stratégies sont possibles pour cette recherche "générale" de travail afin de trouver celui qui a la plus grosse granularité possible. Un certain nombre ont été explorées par le simulateur PEPSys (voir [Baron 89]). Les expériences faites en ce domaine par l'implémentation PEPSys sont présentées au chapitre 7.

Parallélisme ET déterministe

Dans tout ce qui suit, nous appellerons parallélisme ET déterministe le parallélisme engendré par des sous-buts qui ne produisent chacun qu'une seule solution. Nous supposerons que cette propriété est connue statiquement à la compilation du programme, soit par analyse, soit parce que l'utilisateur le fait savoir au compilateur[1]. La connaissance du déterminisme de chacun des sous-buts ET-parallèles permet de simplifier de façon importante l'implémentation, puisqu'il n'est pas nécessaire d'effectuer le produit croisé des solutions aux sous-buts. Un sous but déterministe peut néanmoins générer plusieurs branches OU-parallèles. Seule la première branche de chaque partie à produire une solution sera utilisée.

Un processus débutant l'exécution de deux sous-buts ET-parallèles crée, sur sa pile de contrôle, une structure de données appelée *fork-point*, accessible depuis les autres processeurs. Il démarre ensuite l'exécution du premier sous-but dit de gauche. S'il se trouve un processeur inactif, celui-ci utilise les données du *fork-point* pour créer un processus qui exécute alors le sous-but de droite. Les deux processus se synchronisent en fin d'exécution des sous-buts par rendez-vous. En raison du mécanisme de gestion des liaisons, présenté ultérieurement, c'est en général la branche droite qui continue le calcul après le rendez-vous.

Combinaison des parallélismes OU et ET

La solution du modèle PEPSys est complexe et nous recommandons au lecteur interessé la lecture de [Hailperin 86]. Lorsque les parallélismes OU et ET sont combinés, il est nécessaire de réaliser le produit croisé entre l'ensemble des solutions produites par la partie gauche et de celles produites par la partie droite. Les choix principaux faits à la définition du modèle sont d'éviter la recopie des solutions des branches droites ou gauches, au prix de calculs éventuellement redondants, ainsi que de favoriser la poursuite du calcul de la première solution possible en formant les produits croisés dès qu'un processeur est libre et qu'un couple de solutions est disponible.

Le produit croisé de PEPSys est effectué par les processus de la partie droite[2]. Lorsqu'un processus de la partie droite termine l'exécution du sous-but droit, il crée un *extensible branch-point*, qui offre autant de possibilités de travail aux processeurs oisifs qu'il y a de solutions à la partie gauche. Il accouple ensuite la solution qu'il a produite avec l'une des solutions du sous-but de gauche, et continue l'exécution. Les produits croisés entre "sa" solution droite et les autres solutions gauches seront effectués par des processeurs oisifs s'il y en a ou par lui-même lors d'un retour-arrière. Pendant ce temps là les processus ayant produit une solution au sous-but gauche sont bloqués. Les solutions qu'ils ont produites

[1] Rien n'est prévu dans la définition du langage pour déclarer le déterminisme. Cependant, un cas particulièrement simple à détecter est celui où le prédicat de chacun des sous-buts ET-parallèles a la propriété *solutions(one)*

[2] En raison du mécanisme de gestion des environnements partagés, présenté ultérieurement...

sont répertoriées dans la **liste de solutions courante**. Ces processus peuvent éventuellement trouver du travail en "aidant" la partie droite.

Lorsque l'ensemble des produits croisés entre la solution d'un processus de la partie droite avec les solutions de la liste courante de la partie gauche a été calculé, le processeur de la partie droite commence à calculer d'autres solutions au sous-but droit, en effectuant un retour-arrière. **Simultanément**[3], la liste de solutions courante est bloquée, c'est-à-dire que les solutions au sous-but gauche produites après ce blocage, seront répertoriées dans une **nouvelle liste de solutions**. Lorsque l'ensemble des solutions à la partie droite a été calculé et combiné avec l'ensemble des solutions à la partie gauche - cette situation est détectée par un retour arrière à la racine de la partie droite -, les processus de la partie gauche, ayant produit les solutions de la liste courante, sont **débloqués**. Ils entament alors le calcul de nouvelles solutions pour la partie gauche, en effectuant des retour-arrières. **Simultanément**, la nouvelle liste de solutions devient liste courante. La partie droite est **recalculée**, les produits croisés entre les solutions de la partie droite et celles de la partie gauche se faisant de la même façon que lors du calcul précédent.

La partie droite est donc calculée autant de fois qu'il y a de listes de solutions à la partie gauche. Le calcul s'interrompt lorsque toutes les solutions à la partie gauche ont été produites, - cette situation est détectée par un retour arrière à la racine de la partie gauche -. Les processus de la partie droite sont alors **tués** et le calcul se poursuit par un retour arrière du processus ayant créé le *fork-point*.

Une alternative au recalcul des solutions de la partie droite serait leur stockage par "recopie", qui est équivalent à la compilation du parallélisme ET en parallélisme OU présentée à la section 3.4.7. L'optimum dépend de la taille des solutions et donc du programme considéré. Seule l'expérience permettrait de choisir la technique adaptée au plus grand nombre de cas. On peut noter que dans le cas le plus défavorable, où la partie gauche produit plusieurs solutions à raison d'une seule par liste de solutions, la partie droite est recalculée autant de fois qu'elle le serait par un système Prolog séquentiel.

La priorité donnée dans le modèle à la production de la première solution est motivée par le fait que la conjonction de sous-buts ET-parallèles peut être incluse dans une partie du programme dont l'utilisateur ne désire qu'une seule solution. Dès que celle-ci est produite, il est alors possible de libérer les ressources occupées à calculer les autres solutions. Ce n'est pas possible dans les modèles qui calculent l'ensemble des solutions aux parties gauche et droite avant d'effectuer le produit croisé. La contrepartie est la grande complexité du modèle. Il n'est d'ailleurs pas sûr que cette complexité en permette une implémentation efficace.

[3] Le début du retour-arrière et le basculement des listes de solutions sont effectués de façon *atomique*.

4.4.2 Partage des environnements: gestion des liaisons pour le parallélisme OU

La conception du modèle PEPSys a essentiellement profité du modèle Kabu-Wake et des travaux de Borgwardt [Sohma et al. 85,Borgwardt 84]. Chaque processeur virtuel exécute une instance d'une machine abstraite dérivée de la WAM et sur chacun de ces processeurs virtuels peuvent cœxister plusieurs processus PEPSys à la condition que seul le plus jeune soit actif. Les principales caractéristiques du modèle PEPSys sont que les liaisons conditionnelles sont faites par des liaisons superficielles (voir paragraphe 3.4.2) et qu'il y a une distinction **explicite** entre une adresse propre à un processus (**adresse locale**) et une adresse relative à un autre processus (**adresse non-locale**).

Hash-window

Une *hash-window* est une zone de données permettant d'associer une valeur de liaison à une adresse non-locale de cellule de variable. Un élément de *hash-window* comporte deux champs : le premier renferme l'adresse d'une cellule de variable non-locale et est utilisé par une fonction d'adressage dispersé pour calculer l'index de l'élément dans la *hash-window* du processus ; le second contient la liaison de la variable logique. Plusieurs processus peuvent identifier des variables logiques différentes par l'adresse d'une même cellule de variable, à condition de disposer chacun d'une *hash-window*.

Mécanisme de liaison

Toute liaison d'une variable locale est réalisée par une liaison superficielle étiquetée par une **date** appelée *OBL (Or Branch Level)* ; la valeur de l'*OBL* est le nombre de *branch-points* créés par le processus. Pour lier une variable non-locale, on utilise le principe de liaison profonde. Chaque processus possède à cet effet une *hash-window* dont chaque entrée peut contenir une liaison profonde. Cette liaison profonde est, elle aussi, étiquetée par l'*OBL* courant. Des exemples de liaisons dans le modèle PEPSys sont donnés à la figure 4.3.

Une **adresse non-locale** contient **explicitement** l'information indiquant qu'elle n'est pas locale au processus dans lequel elle est utilisée. En outre, elle comporte les informations permettant d'accéder le processeur où se trouve la cellule désignée, qui peuvent être par exemple un couple nom-de-processeur nom-de-processus, ce dernier étant local au processeur distant. Enfin, elle contient la valeur de l'*OBL* sur le processeur distant, au moment ou le *branch-point* conduisant à la branche courante a été créé.

Mécanisme de déréférencement

Le déréférencement d'une variable locale est identique au déréférencement d'une variable dans une implémentation séquentielle. Lorsqu'une référence non-locale est

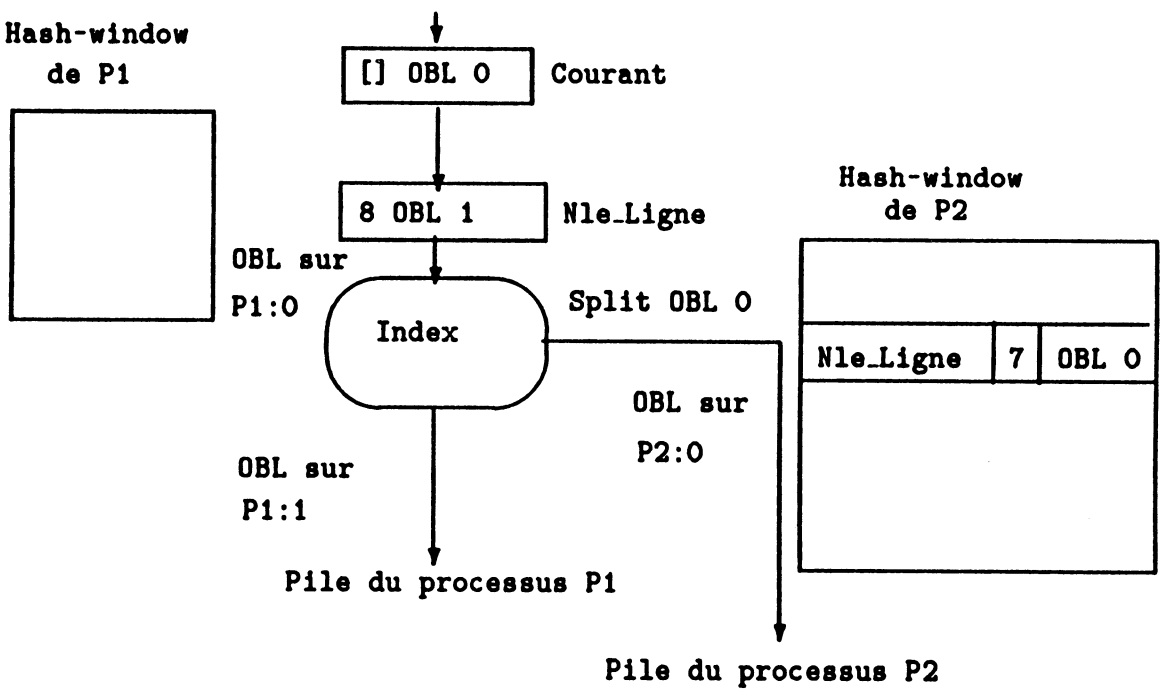

Figure 4.3: Liaisons multiples dans le modèle PEPSys

Cette figure illustre l'accès aux variables *Courant* et *Nle_ligne* du programme de la figure 3.2 dans les mêmes conditions que celles de la figure 3.3. La liaison de *Courant* à *[]* dans la pile de p_1 est valide pour p_2 car elle a été effectuée avant la création de l'alternative dont est issue p_2 (valeur de *binding OBL* : 0). Ce n'est pas le cas de *Nle_ligne*, liée pour p_1 dans sa pile à la valeur 8 ; cette liaison n'est pas valide pour p_2 (valeur de *binding OBL* : 1) qui doit lier *Nle_ligne* à la valeur 7 dans sa *hash-window*.

rencontrée l'opération est plus complexe. Une référence non-locale est une structure de données qui contient les informations permettant d'accéder la cellule de la variable non-locale considérée (numéro de processeur, etc...) ainsi que le niveau de l'alternative (*OBL*) d'où est issue la branche courante. La variable non-locale est accédée et l'*OBL* étiquetant une éventuelle liaison superficielle est comparé avec celui se trouvant dans la référence non-locale. Le résultat de cette comparaison permet de déterminer la validité de la liaison superficielle, c'est à dire de savoir si elle a été effectuée avant ou après la création de l'alternative. Cependant une variable, dont la liaison superficielle n'est pas valide pour un processus, peut avoir été l'objet d'une liaison profonde valide pour ce processus. La *hash-window* du processus courant est donc accédée et, si aucune liaison pour cette variable ne s'y trouve, les *hash-windows* intermédiaires des processus parents sont alors explorées. La figure 4.4 donne un exemple de liaisons superficielles et profondes dans le modèle PEPSys.

La longueur d'une chaîne de *hash-windows* n'est pas bornée et il s'agit là clairement d'un inconvénient du modèle. Cependant on peut remarquer que la longueur des chaînes de références qui sont parcourues dans l'opération de déréférencement, en Prolog séquentiel ou parallèle, n'est pas non plus bornée. Les résultats expérimentaux montrent pourtant que cette chaîne est rarement plus longue que un (voir section 8.3). En outre les analyses dynamiques de programmes Prolog existants ont montré que ces derniers présentaient une bonne localité de référence à l'exécution, ce qui tend à indiquer que les chaînes de *hash-windows* seront courtes. Les références non-locales doivent donc être peu fréquentes. Les résultats expérimentaux qui seront présentés ultérieurement confirment les hypothèses faites à la définition du modèle. Les déréférencements non-locaux sont habituellement peu fréquents. Ceux qui impliquent une recherche dans les *hash-windows* sont encore plus rares et dans ce cas la longueur de la chaîne de *hash-windows* est très courte.

Installation d'une tâche

Le modèle PEPSys permet de transmettre des tâches à un coût très réduit. Une tâche potentielle est indiquée par un *branch-point*, structure de données similaire à un point de choix, qui peut également être utilisé par le retour-arrière. Lorsqu'un processeur libre a trouvé du travail :

- il ne recopie que le sous-but générateur de parallélisme OU. Les références contenues dans ce sous-but sont transformées en références non-locales afin de permettre l'utilisation du mécanisme de gestion des variables partagées de PEPSys. La même transformation est faite pour le pointeur vers la pile des environnements.

- il crée en outre une *hash-window* pour y inscrire ses liaisons profondes. Cette création peut éventuellement être retardée jusqu'à la première liaison profonde.

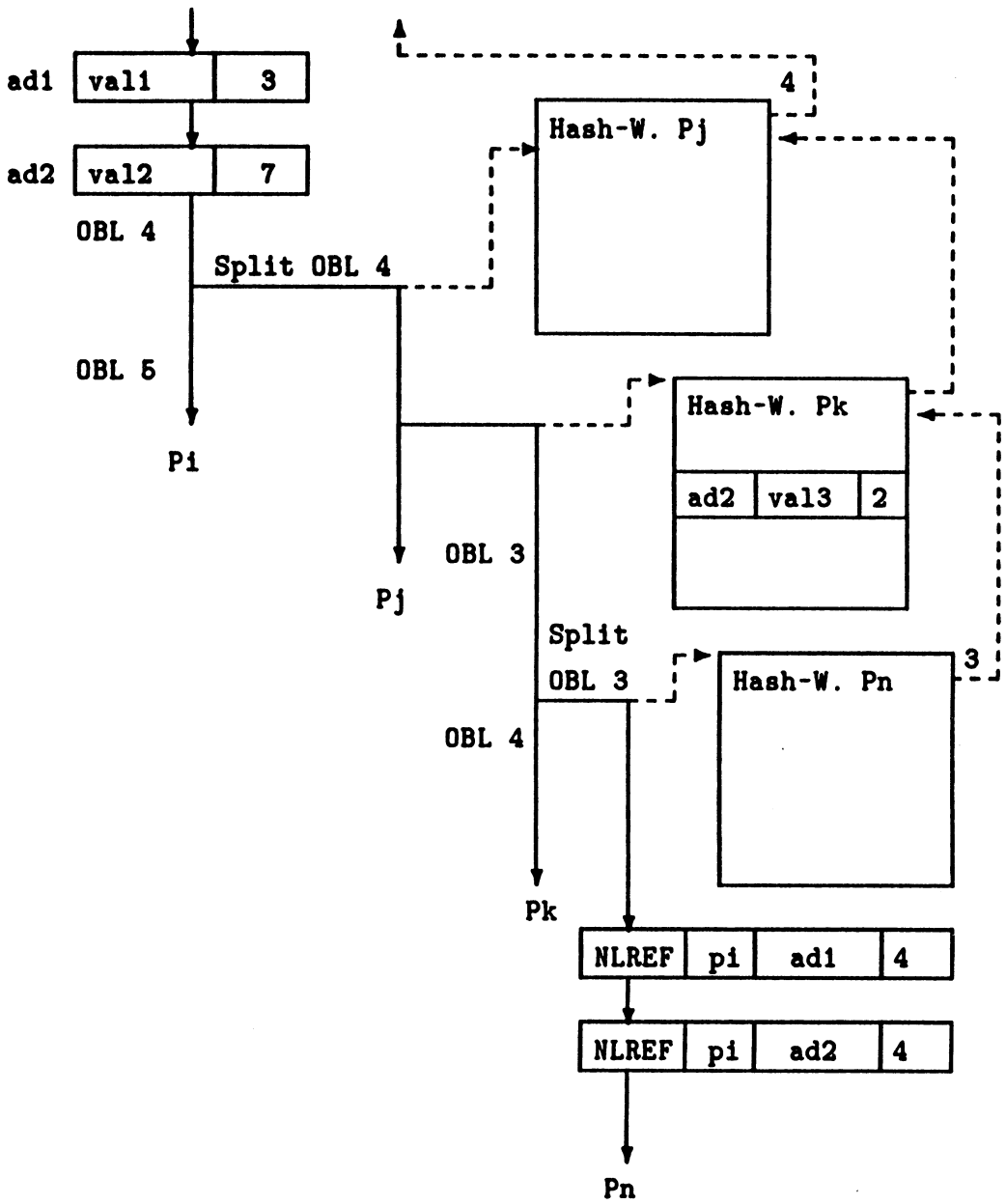

Figure 4.4: Liaisons superficielles et liaisons profondes dans PEPSys

La variable à l'adresse ad_1 a été liée à la valeur val_1 tandis que la valeur de l'*OBL* sur P_i était de 3 ; la séparation qui donnera finalement naissance à P_n s'est produite ultérieurement (valeur de l'*OBL* de P_i 4) et la liaison superficielle à l'adresse ad_1 est donc valide pour P_n. La variable à l'adresse ad_2 n'était pas liée quand cette séparation s'est produite et sa liaison superficielle à la valeur val_2 n'est donc pas valide pour P_j, P_k et P_n. Cependant cette variable a été liée de façon profonde dans la *hash-window* de P_k tandis que l'*OBL* dans P_k valait 2, c'est-à-dire avant la création de P_n qui s'est produite quand l'*OBL* de P_k valait 3. La liaison profonde de ad_2 à la valeur val_3, dans la *hash-window* de P_k, est donc valide pour le processus P_n.

4.4.3 Comparaison du parallélisme OU de PEPSys avec les modèles SRI et Kabu-Wake

Le modèle PEPSys intègre les parallélismes OU et ET ainsi que la combinaison des deux, ce qui n'est pas le cas des deux autres modèles. Cependant, il est possible de comparer la gestion du parallélisme OU dans PEPSys avec les deux autres modèles.

Mécanisme de liaison

La liaison des variables est aussi coûteuse dans les modèles Kabu-Wake et PEPSys où chaque liaison doit être étiquetée par la valeur d'un compteur. Cette opération n'introduirait aucun surcoût sur du matériel specialisé (traitement de *tag*). Dans le modèle SRI, les liaisons conditionnelles sont enregistrées dans un tableau de liaisons, et le surcoût effectif du à ce mécanisme dépend du pourcentage de ce type de liaisons et donc du programme considéré. D'autre part, il ne semble pas qu'un matériel spécialisé puisse réduire le surcoût du à un accès indirect au tableau de liaisons.

Mécanisme de déréférencement

Le modèle Kabu-Wake est le plus efficace pour cette opération car il utilise le même algorithme que l'implémentation séquentielle de Prolog. Le modèle PEPSys est aussi très efficace dans le cas du déréférencement de variables locales, cas le plus fréquent en raison de la bonne localité de références à l'exécution de programmes Prolog. Pour le déréférencement d'une variable non-locale, le modèle PEPSys peut impliquer l'exploration d'une "chaîne de *hash-windows*" de longueur non bornée, alors que l'opération correspondante dans le modèle SRI consiste en un accès indirect à un élément de tableau. Les résultats expérimentaux qui sont donnés ultérieurement indiquent que dans la quasi totalité des programmes plus de 90% des chaînes de recherches dans les *hash-windows* sont de longueur 1 (référence présente dans la *hash-window* locale), et que le pourcentage de chaînes de longueurs supérieures à 5 est très inférieur à 1%.

Installation de tâches

Le modèle Kabu-Wake est très inefficace puisqu'il implique dans ce cas une complète recopie de la résolvante (pile d'environnements). Le modèle SRI implique l'effacement et l'installation de liaisons dans le tableau de liaisons du *worker* qui installe la tâche. La complexité de cette opération dépend de la proximité de la tâche trouvée avec la tâche précédement exécutée par le *worker*. C'est pour cette raison qu'une importance particulière a été accordée au développement d'ordonnanceurs sophistiqués dans le projet Gigalips [Butler et al. 88,Brand 88a]. L'installation de tâche dans PEPSys est très rapide et ne dépend pas de la tâche sélectionnée.

L'implémentation actuelle de PEPSys obtient d'ailleurs des performances comparables à celles du système Aurora[4], malgré un ordonnanceur très rudimentaire (voir paragraphe 8.2.2).

Synthèse

Les critères ci-dessus sont insuffisants pour comparer les trois méthodes. Il faudrait au moins prendre en compte la complexité des modèles, qui semble supérieure dans le modèle PEPSys, la consommation mémoire, qui doit être supérieure dans les modèles PEPSys et Kabu-Wake, et la portabilité sur diverses architectures, qui doit être meilleure pour les modèles Kabu-Wake et PEPSys. Les informations disponibles à l'heure actuelle ne permettent cependant pas de conclure formellement sur ces divers points.

4.4.4 Gestion des liaisons : combinaison des parallélisme OU et ET

En PEPSys, il ne se pose pas de problème de gestion de liaisons en présence de parallélisme ET seul. Cela provient du choix de n'offrir que le parallélisme ET indépendant dans le langage PEPSys : les sous-buts exécutés en parallélisme ET ne sont pas susceptibles d'instancier une même variable et peuvent donc n'effectuer que des liaisons superficielles. Ici encore, les problèmes les plus complexes se posent lors de la combinaison des parallélismes ET et OU.

Sans rentrer dans les détails, nous pouvons signaler que si la création de *hash-window* n'est pas nécessaire lors de la création d'un processus pour exécuter la branche droite d'une conjonction de deux sous-buts, cette création devient nécessaire dès que la branche droite crée une possibilité de parallélisme OU. Dans ce cas, les processus fils du processus de droite doivent distinguer, parmi les liaisons faites par le processus de droite dans la pile du processus père qui exécute la branche de gauche, les liaisons faites avant le *branch-point* (valides) des autres.

Une nouvelle structure de données est introduite dans le cas du produit croisé : il s'agit de la *join-cell*. Une *join-cell* est associée à chaque processus formé par produit croisé entre une solution de la branche gauche et une solution de la branche droite. Chaque *join-cell* contient trois pointeurs vers des *hash-windows* : la dernière de chacun des processus gauche et droit, et la dernière commune aux deux branches (voir figure 4.5). Une *join-cell* est liée à la chaîne des *hash-windows*. L'algorithme de déréférencement donné pour le OU-parallélisme est modifié par la rencontre d'une *join-cell*. Les chaînes de *hash-windows* des branches droite puis gauche sont alors successivement explorées, et ceci récursivement s'il existe plusieurs niveaux d'imbrication entre les parallélismes ET et OU.

Le modèle de calcul de PEPSys [Hailperin 86] étudie l'ensemble des cas possibles, suivant l'ordre dans lequel les branches parallèles sont créées. Certaines situations sont très complexes à gérer et sont susceptibles, si elles se produisent,

[4] Implémentation du modèle SRI.

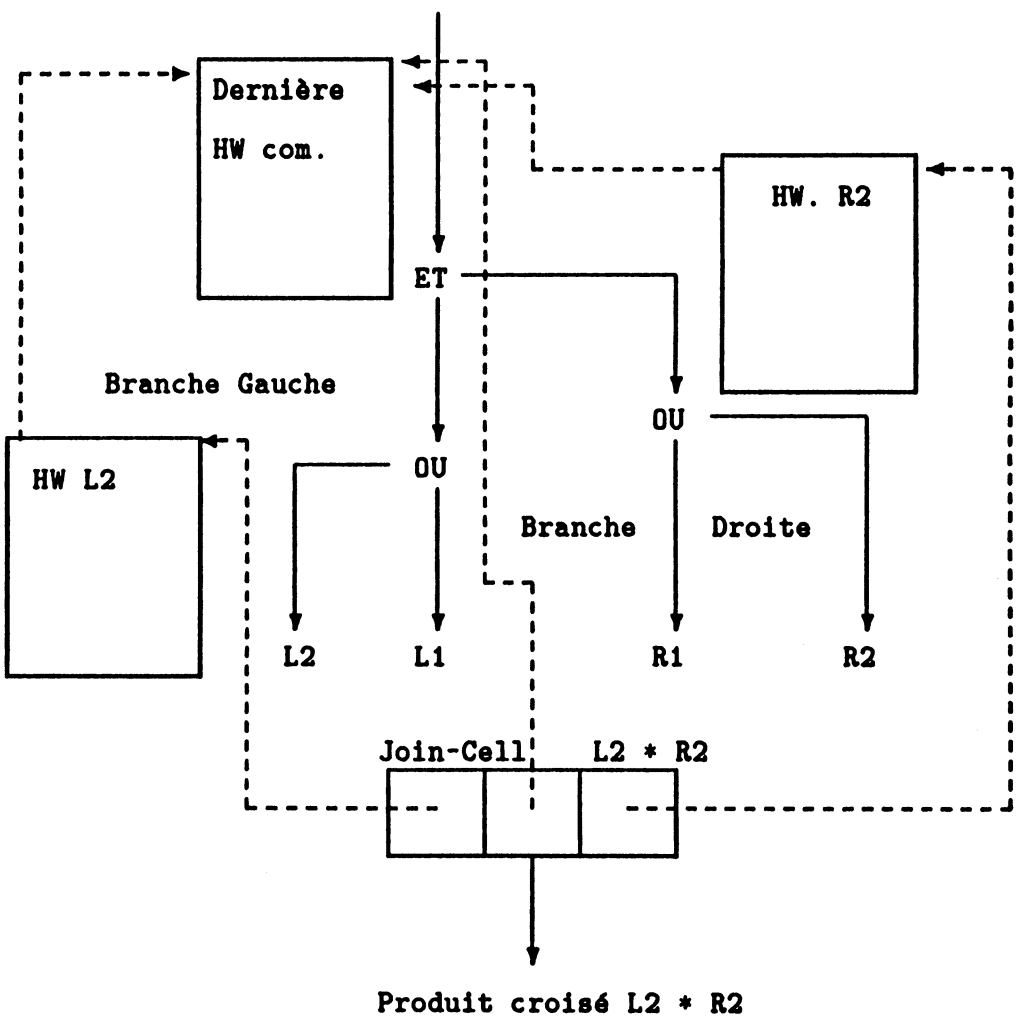

Figure 4.5: Utilisation des *join-cells* dans les produits croisés

Le processus correspondant au produit croisé des processus L_2 et R_2 est identifié par sa *join-cell*. Dans le processus $L_2 * R_2$, la recherche des *hash-windows* consiste à explorer la branche droite jusqu'à la dernière *hash-window* commune, puis à reprendre l'exploration "normale" des *hash-windows* en suivant la branche gauche si besoin est.

d'alourdir notablement le déréférencement. La combinaison des parallélisme OU et ET n'ayant pas encore été testée, il est difficile de dire si ce modèle peut être implémenté efficacement. Si besoin est, de nombreuses simplifications sont envisageables au prix d'une redondance des calculs dont il est difficile d'évaluer les conséquences sur les performances du système.

4.5 Machine Abstraite, Implémentation parallèle

La machine abstraite PEPSys [Chassin et Robert 88] fait l'objet du chapitre 5 de cette thèse. Le compilateur du langage PEPSys y sera présenté à cette occasion. L'implémentation parallèle [Chassin et al. 88] et les mesures qu'elle a permis de faire font l'objet du reste de la thèse.

4.6 Simulateur de PEPSys

Le simulateur de PEPSys [Baron et al 88] permet de tester le modèle de calcul sur une architecture multi-processeurs extensible. L'architecture retenue est celle de grappes (*clusters*) de processeurs. Chaque grappe est composée de plusieurs processeurs interconnectés par une mémoire commune. L'un de ces processeurs est un processeur de communication spécialisé qui communique par messages avec les autres grappes (voir figure 4.6). Cette architecture est très similaire à celle de la machine PIM du projet japonais de Cinquième Génération [Chikayama 87]. Les configurations simulées introduisent la notion de distance, l'accès à une autre grappe étant plus coûteux que l'accès à la grappe locale. De même le système simulé n'a pas d'espace d'adressage global puisque la communication entre grappes doit se faire par messages. La configuration simulée est paramétrée par le nombre de grappes interconnectées et le nombre de processeurs par grappe. Pour une configuration ne comportant qu'une grappe, le multiprocesseur simulé offre la même architecture que celui qui est utilisé par l'implémentation, ce qui a permis de valider le simulateur en retrouvant les résultats de l'implémentation. De même, il est possible de simuler des architectures complètement distribuées en limitant le nombre de processeurs par grappe à une unité.

Le **processeur de communication** est appelé pour déréférencer et rechercher du travail dans les grappes distantes. Lors du déréférencement d'une variable non locale à la grappe où il se trouve, il contacte le processeur de communication de la grappe où se trouve la variable. Ce processeur effectue lui-même l'opération de déréférencement. Ce déréférencement sur une grappe distante peut lui-même impliquer l'accès à d'autres grappes et ce récursivement. En fin de déréférencement, la liaison de la variable est retournée au processeur de communication de la grappe d'où est partie la demande initiale, qui la transmet au processeur de la grappe ayant initialisé l'opération de déréferencement. Les processeurs de

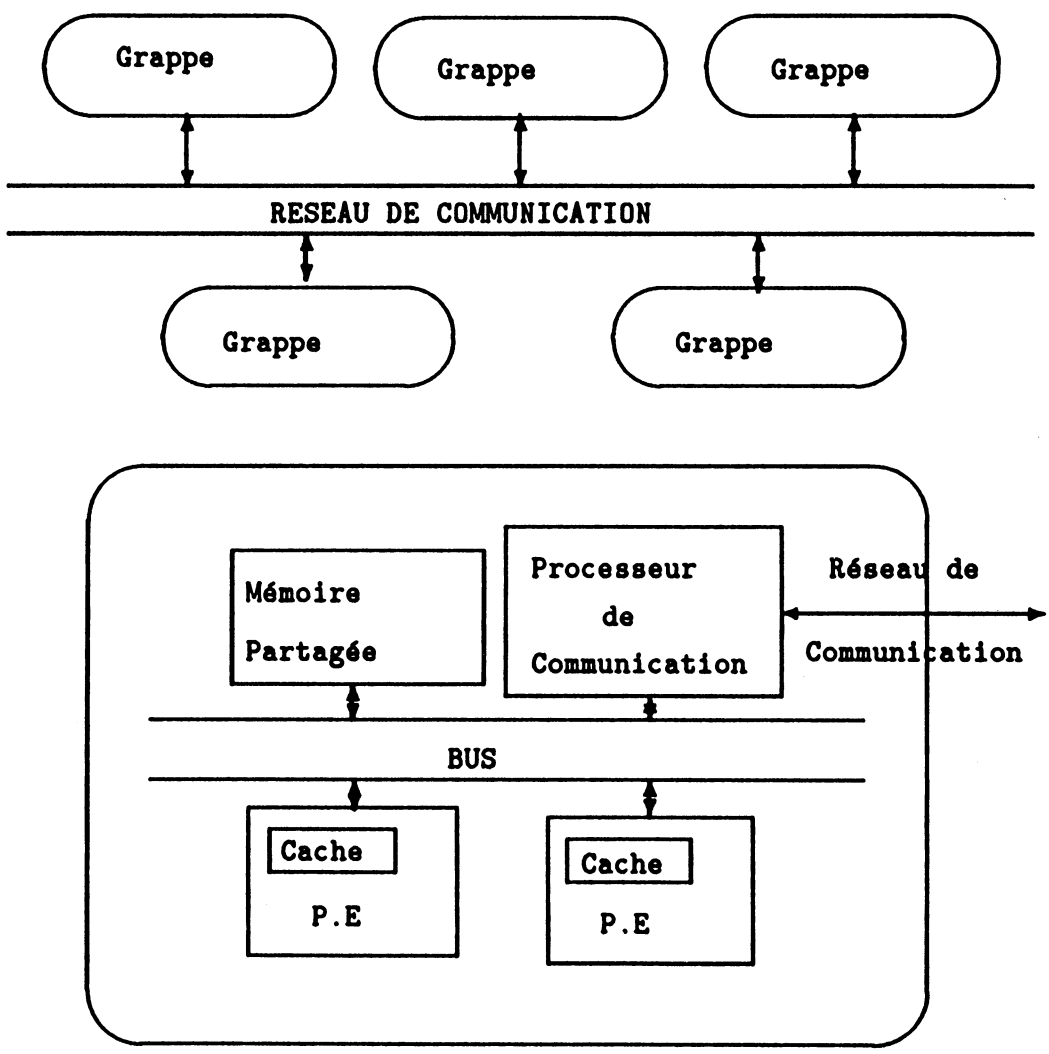

Figure 4.6: Architecture multi-grappe simulée dans PEPSys

Chaque grappe est composée d'un ensemble de processeurs, de un à dix, connectés par un bus commun à une mémoire commune. Les grappes sont interconnectées par un réseau de communication. Chacune des grappes est connectée à ce réseau par un processeur de communication qui joue en outre le rôle de serveur de déréférencement et de serveur de tâches à exécuter pour les autres grappes.

communication coopèrent également entre eux afin d'équilibrer la charge de travail entre les différentes grappes. Différentes stratégies ont été testées pour le partage de charge entre grappes [Baron 89].

Le simulateur peut être configuré pour simuler par une seule grappe l'architecture du multiprocesseur utilisé pour implémenter PEPSys. Les résultats obtenus sont alors très voisins de ceux obtenus par l'implémentation présentés ultérieurement. Les résultats obtenus en faisant coopérer des processeurs répartis sur plusieurs grappes sont dans l'ensemble médiocres. Des expériences limitées portant sur l'allocation initiale de travail aux grappes indiquent que ces résultats pourraient être améliorés de façon significative en utilisant des algorithmes plus sophistiqués pour le partage de charge. De plus il semble que la taille des problèmes résolus ne soit pas suffisante pour tirer pleinement parti d'une configuration importante. Il n'a cependant pas été possible de simuler l'exécution de problèmes de grande taille sur de grandes configurations, à cause de l'importance du temps et de la mémoire nécessaires à la simulation de grosses configurations.

4.7 Ecriture d'applications parallèles. Analyse du parallélisme

L'écriture d'applications parallèles en langage PEPSys a été considérée comme fondamentale pour tester et valider les idées développées dans le projet. Test et validation du langage tout d'abord, puisque l'écriture de programmes permet de mettre en évidence le pouvoir expressif du langage tout comme ses points faibles ; des améliorations peuvent ainsi être suggérées. Test et validation du modèle également puisque les applications peuvent être utilisées comme programmes de test pour le simulateur et l'implémentation, permettant ainsi de montrer l'efficacité des solutions proposées par PEPSys sur des applications "en vraie grandeur". De nombreux programmes "jouets" ont été tout d'abord écrits. Une application de plus grande envergure a également été réalisée [Ing 87a]. Nous reviendrons ultérieurement sur les applications écrites en PEPSys quand nous présenterons les programmes de test utilisés pour valider l'implémentation.

Un outil d'analyse du parallélisme potentiel des applications a été développé simultanément. Le simulateur et l'implémentation parallèle n'étant alors pas disponibles, il s'agissait de mesurer le parallélisme des algorithmes afin de le maximiser. L'outil d'analyse [Ing 87b] suppose que la création de processus et leur exécution parallèle se fait sans déperdition de performance. L'outil considère également que l'unification d'un sous-but avec une tête de clause quels qu'ils soient se fait en une unité de temps. La dernière hypothèse simplificatrice est que les ressources du système sont infinies et donc que toutes les possibilités de parallélisme sont exploitées. La figure 4.7 montre un exemple d'analyse d'un programme parallèle.

L'outil d'analyse du parallélisme est demeuré utile après les réalisations du simulateur et de l'implémentation. En effet, il permet de donner une "trace" de programme parallèle indépendante du système sur lequel il sera exécuté. A ce titre,

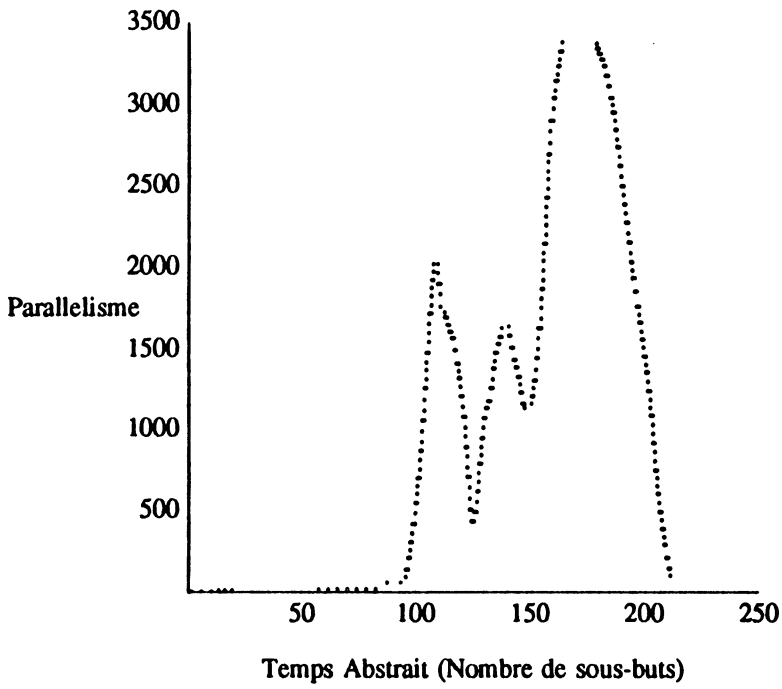

Figure 4.7: Analyse abstraite du parallélisme du programme *TInAS*

Le programme *TInAS* est un programme de 150.000 inférences environ. Compte tenu des hypothèses de l'analyse, 3400 processeurs au maximum sont susceptibles de s'exécuter en parallèle produisant un gain de performances maximum d'un facteur 100 environ. Dans la pratique, les différentes sources de surcoût réduisent nettement le nombre de processeurs utilisables efficacement et le facteur de gain de performances.

il permet un travail d'optimisation des algorithmes. Une tentative de corrélation entre les l'analyse abstraite et les résultats d'implémentation et de simulation est donnée dans [Chassin, Baron et al. 89].

4.8 Résumé du chapitre

Le projet PEPSys est un projet qui aborde l'ensemble des aspects liés à la parallélisation de la programmation logique, qui comprennent le langage, le modèle de calcul et les applications parallèles. Les choix faits pour le langage et le modèle de calcul sont fondés sur des analyses statiques et dynamiques d'applications Prolog existantes. Ils ont été validés par une implémentation parallèle et un simulateur d'une architecture extensible, tous deux réalisés en utilisant les techniques les plus efficaces d'implémentation développées pour Prolog. Un outil d'analyse du parallélisme des applications permet en outre de développer celles-ci indépendamment d'une implémentation ou d'une architecture de multiprocesseur.

Chapitre 1

Introduction

Le cadre de cette thèse est la recherche en programmation logique parallèle dans le but d'améliorer son efficacité. Les motivations pour la recherche dans ce domaine se trouvent au confluent de deux évolutions. D'une part l'utilisation de langages de programmation logique pour contribuer, en raison de leur grand richesse sémantique, à la solution de ce que l'on appelle communément la "crise du logiciel". La programmation logique souffre cependant d'un important handicap qui est sa relative inefficacité sur les machines actuelles. D'autre part, la construction de multiprocesseurs incorporant plusieurs micro-processeurs à bas prix de revient unitaire afin d'obtenir des machines puissantes à faible coût. Le problème est ici de tirer parti de toute la puissance offerte. La sémantique opérationnelle des langages de programmation logique n'est que faiblement séquentielle, ce qui fait des langages de ce type de bons candidats pour la programmation parallèle et ce qui explique l'importance de la recherche dans ce domaine, plus particulièrement depuis l'annonce du projet japonais de Cinquième Génération.

1.1 Intérêt de la programmation logique

Si l'utilisation des langages de programmation logique provoque un intérêt croissant en dépit de leur relative inefficacité, c'est qu'ils présentent des caractéristiques les rendant particulièrement adaptés à la solution d'un grand nombre de problèmes. En particulier, l'aspect **déclaratif** des langages de programmation logique permet d'abstraire un peu plus les programmes des machines sur lesquelles ils sont exécutés [Abbott 87].

On peut recenser quelques uns des problèmes auxquels la programmation logique apporte une solution "naturelle" :

- l'écriture des compilateurs se fait de façon relativement simple étant donnée la ressemblance des programmes Prolog avec les W-grammaires [Simonet 81]. [Cohen 87] montre combien Prolog permet d'accroitre la productivité des programmeurs de compilateurs.

Partie II

Techniques d'implémentation

Deux droites parallèles ne se rencontrent jamais.
Géométrie euclidienne.

Note de l'auteur : ce n'est malheureusement pas le cas des processus parallèles.

Chapitre 5

Machine Abstraite PEPSys

Comme nous l'avons expliqué dans la section 3.4, les systèmes multiséquentiels tels que PEPSys sont basés sur les techniques d'implémentation les plus efficaces de Prolog séquentiel. La *WAM*, qui joue le rôle de standard pour l'implémentation de systèmes Prolog efficaces, a donc été étendue afin de permettre la compilation du langage PEPSys et d'en supporter le modèle de calcul. Le plan du chapitre est le suivant. Après quelques généralités, la machine abstraite PEPSys [Robert 88,Chassin et Robert 88] est présentée en tant qu'extension de la *WAM*. Cette présentation est divisée en deux parties consacrées au contrôle et à la gestion des résolvantes en parallélisme OU. La fin du chapitre décrit l'adaptation d'un compilateur Prolog existant au langage et à la machine abstraite PEPSys.

5.1 Généralités

La machine abstraite PEPSys est basée sur la machine abstraite ICM3 [Noye 87], conçue dans le cadre d'un projet de mise en œuvre matérielle de machine abstraite Prolog, et qui est elle-même dérivée de la *WAM*. Par rapport à la *WAM*, ICM3 comporte certaines optimisations telles que le retour-arrière superficiel (*shallow backtracking*), mais les différences avec la *WAM* ne changent en général rien à la machine abstraite PEPSys. Nous emploierons donc fréquement le mot *WAM* pour désigner la machine abstraite ayant servi de base à la machine abstraite PEPSys. Une exception toutefois : l'optimisation du retour-arrière de ICM3 qui a pour conséquence une implémentation non optimale du parallélisme OU dans PEPSys.

La machine abstraite PEPSys est conçue pour s'exécuter sur chacun des processeurs d'un multiprocesseur exécutant une implémentation de PEPSys. Ainsi, chaque processeur[1] dispose de ses propres registres et zones de données. Nous supposerons également que chacun des processeurs exécute l'ensemble des instructions de la machine abstraite et qu'il dispose en outre de l'ensemble du programme PEPSys en cours d'exécution. Chaque processeur est géré par noyau de gestion

[1] Appelé également parfois *worker* pour tenir compte du fait que l'on utilise un système d'exploitation pour programmer le multiprocesseur et que chaque "processeur" sera en fait un processus du système d'exploitation, *process* de UN*X ou *thread* de Mach par exemple.

de processus ou ordonnanceur assurant recherche de travail, création, terminaison, suspension et réactivation des processus. Comme nous l'avons dit en présentant les systèmes multi-séquentiels, l'exécution d'une portion de programme PEPSys crée des sources de parallélisme susceptibles d'être utilisée par l'ordonnanceur d'un processeur "inactif" pour créer un processus. La définition d'un ordonnanceur est indépendante de la machine abstraite. Plusieurs expériences ont été faites dans le cadre du projet PEPSys tant sur le simulateur que sur le système parallèle. Elles seront décrites au chapitre suivant.

La machine abstraite PEPSys ou PAM (*Pepsys Abstract Machine*) comprend l'ensemble des composants de la *WAM* : objets étiquetés, piles locale, globale et trace, structures de contrôle - environnements et points de choix alloués sur la pile locale - identiques, mêmes registres d'état, registres arguments et pointeur de structure. La PAM incorpore également l'ensemble des instructions de la *WAM*.

Un certain nombre d'extensions ont été faites à la *WAM* pour supporter le modèle de calcul de PEPSys. Tout d'abord, l'immersion du contrôle du modèle PEPSys dans celui de la *WAM* a nécessité la définition de nouvelles structures de contrôle et d'instructions opérant sur ces structures. Dans tous les cas, les mécanismes de contrôle spécifiques à la PAM peuvent être définis comme des extensions à l'un des deux mécanismes de contrôle de la *WAM* : la **continuation** ou le **retour-arrière**. Une fois déterminé auquel des deux mécanismes l'extension voulue se rattache, celle-ci se fait simplement. Par ailleurs, la représentation des résolvantes et donc des liaisons de variables du modèle PEPSys a impliqué la définition de nouveaux objets **non-locaux**, l'extension des instructions *get*, *put* et *unify* portant sur des variables logiques et surtout, de profondes extensions aux opérations élémentaires d'unification et de déréférencement.

5.2 Contrôle

5.2.1 Extensions au contrôle de la *WAM*

La PAM étend le mécanisme de **continuation** lorsqu'une opération doit être effectuée après le succès d'un sous-but. C'est le cas lorsque l'exécution du sous-but gauche d'une conjonction ET-parallèle réussit et produit une solution. Une synchronisation avec les processus exécutant la partie droite devient alors nécessaire. C'est aussi le cas lorsqu'un processus produit une solution à un prédicat déclaré *solutions (one)* ; dans ce cas, une synchronisation avec les autres processus exécutant le prédicat est nécessaire afin que seul l'un d'entre eux poursuive le calcul. Pour qu'une opération soit exécutée comme sa continuation, une instruction :

- pousse en sommet de pile locale une structure de données contenant les paramètres de l'opération.
- insère cette structure de données dans la chaine des environnements. Pour cela on sauvegarde la valeur courante du registre pointeur d'environnements

E dans la structure de données et on fait pointer **E** sur la structure nouvellement créée.

- charge le registre continuation **CP** à l'adresse de l'instruction exécutant l'opération désirée. Cette instruction trouvera ses paramètres dans la structure de données de la pile locale désignée par le registre **E**.

Le mécanisme de **retour-arrière** est étendu pour les prédicats OU-parallèles pour permettre au processus ayant fourni le travail en créant un *branch-point* de se synchroniser, lorsqu'il effectue un retour-arrière, avec les autres processus exécutant le prédicat[2]. De façon analogue, des extensions au retour-arrière de la *WAM* permettent de mélanger exécution séquentielle et parallèle des éléments d'un produit croisé de deux sous-buts ET-parallèles. Le retour-arrière est également utilisé pour synchroniser un processus qui termine avec son père. Pour faire exécuter une opération au retour-arrière, une instruction :

- pousse en sommet de pile locale une structure de données contenant les paramètres de l'opération à exécuter au retour-arrière.
- insère la structure de données dans la chaine des points de choix, ce qui est fait en sauvegardant l'ancienne valeur du registre **B** dans la structure de données et en donnant pour valeur à **B** l'adresse de la structure.
- définit comme instruction de retour-arrière l'adresse de l'instruction effectuant l'opération désirée. Cette instruction trouvera ses paramètres dans la structure de données de la pile locale désignée par le registre **B**. Dans la *WAM*, l'adresse de cette instruction est un élément des points de choix. Dans la machine ICM3, il s'agit au contraire d'un registre appelé **BP** (*Backtrack Pointer*).

5.2.2 Contrôle du parallélisme OU

La machine abstraite considère comme OU-parallèles les prédicats ayant les propriétés suivantes :

-properties[**clauses (unordered)**, *execution(eager)].*

Par contre, un prédicat ayant les propriétés :

-properties[clauses (ordered), execution(eager)].

sera considéré comme séquentiel. Un mécanisme permettant de réordonner des solutions produites en parallèle, nécessiterait de sérialiser les processus parallèles de façon analogue à l'implémentation du *cut* et des effets de bord Prolog

[2]Ces processus ont été créés à partir du *branch-point*.

dans un système OU-parallèle [Hausman 88]. Les gains d'efficacité que l'implémentation OU-parallèle des prédicats ordonnés est susceptible d'apporter (voir [Butler et al. 88] à ce sujet) ne semblent pas justifier la complexité supplémentaire qu'impliquerait cette implémentation.

L'implémentation du contrôle du parallélisme OU par la machine abstraite PEPSys est un cas typique d'extension du mécanisme de contrôle utilisé par la *WAM* pour le retour-arrière. Ce type de contrôle est utilisé par les processus créateurs de *branch-points* au retour-arrière et par les processus OU-parallèles, créés à partir d'un *branch-point* lorsqu'ils terminent.

Instructions et structures de données

Un processus qui exécute un prédicat OU-parallèle crée un *branch-point* sur la pile locale, qui peut être utilisé ou non par des processeurs oisifs. Le travail restant est exécuté par le processus séquentiel lors du retour-arrière. Une synchronisation entre processus est nécessaire si l'on veut éviter d'exécuter plusieurs fois la même clause ou de détruire des structures de données partagées. De nouvelles instructions *par_try* et *par_retry* remplacent la séquence classique de *try, retry, trust* de la *WAM*; ces instructions utilisent un *branch-point*, qui est en fait une extension d'un point de choix comportant les informations nécessaires à la synchronisation (voir figure 5.2).

L'instruction *par_try* pousse un *branch-point* sur la pile locale. Ce *branch-point* est lié à la chaine des points de choix puisqu'on y sauve la valeur courante du registre B avant d'y faire pointer ce même registre B. Il est rendu accessible aux autres processeurs et le pointeur d'instruction de retour-arrière (registre BP) est positionné sur l'instruction *par_retry* suivante. La création de *branch-points* est limitée par les instructions d'indexation qui précèdent les instructions *par_try* et *par_retry*, de la même façon que la création de points de choix.

Retour-arrière à un *branch-point*

L'instruction *par_retry* est exécutée par le processus créateur du *branch-point* lorsqu'il effectue un retour arrière. Le *branch-point* est accédé en exclusion mutuelle et, suivant le nombre de clauses en cours d'exécution et terminées, plusieurs cas sont possibles :

- toutes les clauses ont été exécutées : le processus courant continue le retour-arrière en restaurant les registres correpondants, B et BP depuis le *branch-point.*

- toutes les clauses ont commencé à s'exécuter mais elles ne sont pas toutes achevées : des processus sont encore occupés au calcul des branches OU-parallèles de ce nœud de l'arbre de réfutation. Il n'est pas possible au processus courant de poursuivre son retour-arrière, sous peine d'altérer des données partagées avec les processus actifs[3]. Le processus courant se suspend et rend

[3]Ceci en raison du partage des résolvantes pratiqué par le modèle PEPSys.

```
Programme PEPSys                    Code PAM

-properties( [ solutions ( all ),   p/0:   test_or_par  seq_0
             clauses ( unordered ),         par_try
           execution ( eager ) ] ).         par_retry

p :- cl1.                           seq_0: try   cl1
p :- cl2.                                  retry cl2
p :- cl3.                                  trust cl3

                                    cl1:   <code clause cl1>
                                    cl2:   <code clause cl2>
                                    cl3:   <code clause cl3>
```

Figure 5.1: Compilation d'un prédicat OU-parallèle

L'instruction *test_or_par* est générée pour permettre d'adapter l'exploitation du parallélisme aux conditions d'exécution. Suivant la charge du système, des sources de parallélisme seront créées (*par_try*) ou non (*try*). Si la charge du système diminue ultérieurement, il deviend à nouveau possible de créer des sources de parallélisme mais vraisemblablement de plus faible granularité (plus bas dans l'arbre de réfutation). Cette instruction n'a été utilisée que pour pallier à l'absence d'un véritable mécanisme de création de source de parallélisme "à la demande".

le contrôle à l'ordonnanceur local afin qu'il trouve du travail pour le processeur. L'exécution du processus suspendu sera reprise et le retour-arrière poursuivi lorsque le nouveau travail sera achevé et que tous les processus fils seront terminés.

- toutes les clauses commencées sont terminées et il reste une clause à exécuter. L'instruction *par_retry* est alors équivalente à l'instruction *trust* de la *WAM*.
- dans les autres cas, l'instruction *par_retry* est équivalente à l'instruction *retry* de la *WAM*.

Vie et mort d'un processus OU-parallèle

Comme il a été dit dans la présentation du langage, tous les appels aux modules parallèles sont encapsulés par l'un des prédicats prédéfinis *bagof*, *setof* ou *oneof*. Ainsi, tout processus parallèle est contraint de se terminer par un retour-arrière, soit parce qu'il n'a pas trouvé de solution, soit pour rechercher une autre solution, soit parce qu'une seule solution suffit. Un processus OU effectue les actions suivantes lorsqu'il se termine :

- il informe de sa terminaison le processus père en incrémentant le compteur de processus achevés du *branch-point* dont il est issu.
- l'ordonnanceur du processeur est appellé pour terminer le processus.

Le retour-arrière final est en fait préparé dès la création du processus. Il s'agit d'une extension à la *WAM* que l'on peut classer comme extension au retour-arrière. Lorsqu'un processus OU est créé, son registre pointeur d'instruction de retour-arrière, **BP**, est initialisé à l'instruction *terminate_or_branch* - qui n'est pas générée par le compilateur -, et le registre **B** est initialisé à l'adresse d'une *root-frame* qui contient les paramètres nécessaires à la terminaison du processus. La *root-frame* est la première structure de données créée sur la pile locale du processus (voir figure 5.2).

Critique du contrôle du parallélisme OU

La solution retenue dans la PAM pour implémenter le contrôle du parallélisme OU est inutilement complexe en raison du choix de suivre la machine abstraite ICM3 dans ses moindres détails. Dans celle-ci, l'instruction de retour-arrière est stockée dans un registre, **BP**. N'étant pas accédé sous exclusion mutuelle pour des raisons d'efficacité, ce registre est inaccessible aux autres processeurs, lors d'une exécution parallèle. L'implémentation du parallélisme OU qui en résulte complique notablement l'exécution du processus créateur de parallélisme OU, si celui-ci n'est pas utilisé.

Une solution plus simple serait de stocker, comme dans la *WAM*, l'instruction de retour arrière dans un champs des points de choix et *branch-points*, de ce fait accessible aux autres processeurs. La compilation du parallélisme OU pourrait alors

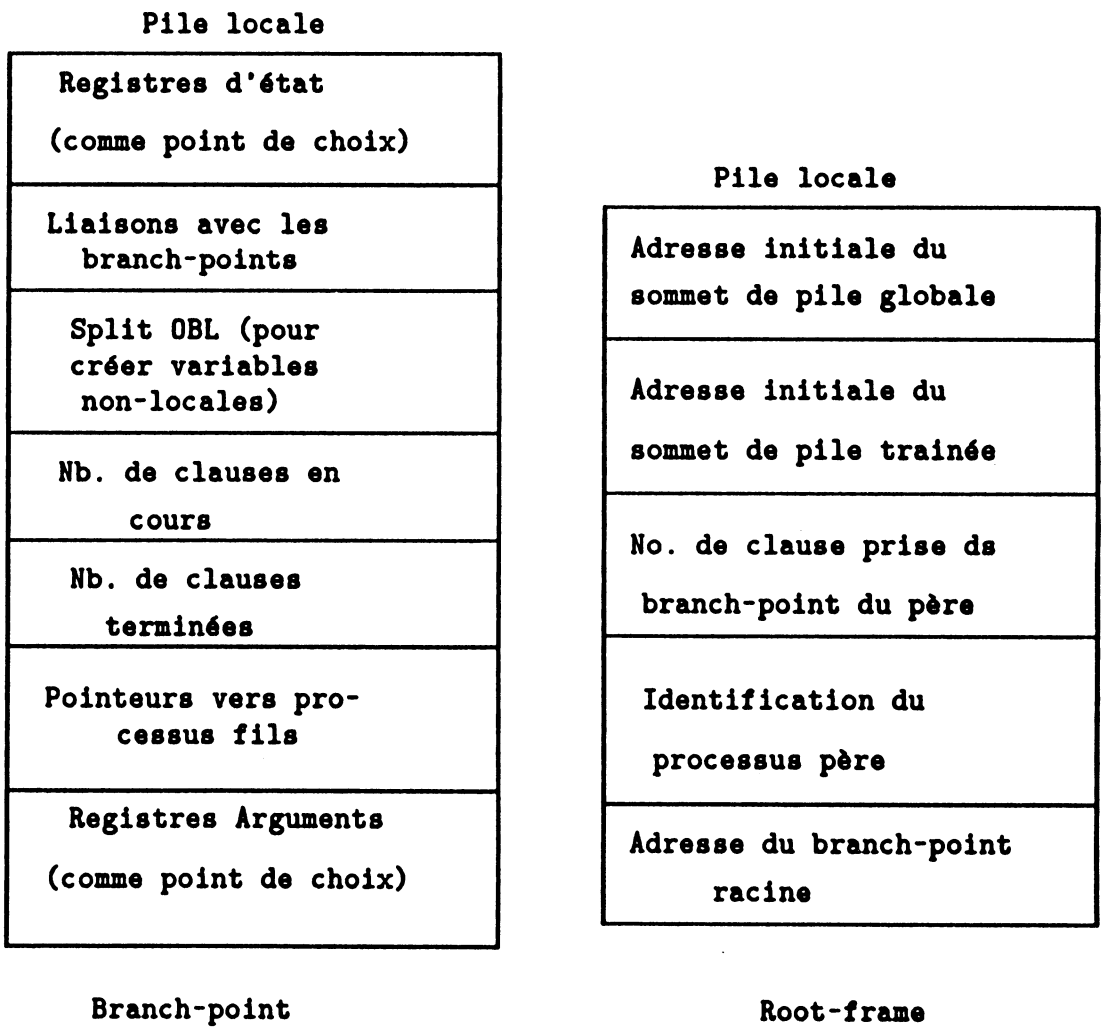

Figure 5.2: Structures de données utilisées pour le parallélisme OU

se faire en remplaçant les instructions *try*, *retry* et *trust* des prédicats OU-parallèles par des instructions *par_try*, *par_retry* et *par_trust* ; la sémantique de ces instructions serait très proche de celle des instructions séquentielles correspondantes dans le cas, fréquent, où le parallélisme potentiel reste inexploité. L'exclusion mutuelle ne serait nécessaire que lors de l'accès aux *branch-points* comme c'est actuellement le cas.

5.2.3 Contrôle du parallélisme ET

Dans cette section, nous présentons les extensions faites à la *WAM* pour supporter le parallélisme ET seul ou combiné avec le parallélisme OU et l'exécution séquentielle. L'implémentation de la combinaison des parallèlismes OU et ET définie par le modèle de calcul PEPSys est trop complexe pour être présentée intégralement ici. Nous montrons simplement comment une clause contenant deux sous-buts ET-parallèles est compilée et le cheminement du contrôle dans les instructions ET-parallèles. Le lecteur avide de détails est invité à se reporter à [Robert 88,Chassin et al. 87].

Instructions et structures de données

Un processus débutant l'exécution d'une conjonction de sous-buts ET-parallèles exécute une instruction *fork* qui pousse un *fork-point* sur la pile locale, puis il commence l'exécution du sous-but gauche. Le *fork-point* est rendu accessible aux autres processeurs, de façon analogue aux *branch-points* du parallélisme OU, ce qui permet aux processeurs oisifs de calculer le sous-but droit en parallèle avec le calcul du sous-but gauche. A la fin du sous-but gauche, tout processus qui l'exécute est synchronisé avec le ou les processus exécutant le sous-but droit ; cette synchronisation est opérée en tant que continuation du sous-but gauche, en exécutant une instruction *check* dont les paramètres sont stockés dans le *fork-point* ; il est donc nécessaire de lier le *fork-point* à la chaine des environnements, afin qu'il soit accessible lors de l'exécution de l'instruction *check*. Si le sous-but gauche échoue, le *fork-point* est retiré de la liste de travail offert aux autres processeurs et s'il existe des processus en cours d'exécution de la branche droite, ils doivent être tués, ce qui est une forme de retour-arrière semi-intelligent. La branche gauche se termine lors du retour-arrière au début de cette branche (voir section 4.4.1). Cette terminaison est implémentée par l'instruction *terminate_left* dont les paramètres se trouvent dans le *fork-point*. Le *fork-point* est donc également lié à la chaine des points de choix, ce qui le rend accessible lors du retour-arrière.

Le but de l'instruction *check* est de déterminer si la partie droite est en cours de calcul. Si c'est le cas, le processus courant se suspend ; dans le cas contraire, il commence l'exécution du sous-but droit. Dans ce cas la continuation du sous-but droit est positionnée à l'instruction *join*, dont les paramètres sont également stockés dans le *fork-point*.

Programme PEPSys

```
c1 :- ..., 1 # r, s, ...
```

Code PAM

```
c1:  .
     .
     test_and_par
     fork @r
     put ...
     put ...
     ...
     call 1
     check
@r:  put ...
     put ...
     ...
     call r
     join
     call s
```

Figure 5.3: Compilation d'un prédicat ET-parallèle

L'exemple ci-dessus montre la compilation simplifiée d'une clause ET-parallèle. En particulier il existe de nombreuses instructions spécialisées qu'il serait trop long de décrire ici. De plus, une instruction *test_and_par* est également générée pour permettre d'adapter l'exploitation du parallélisme aux conditions d'exécution.

5.2.4 Comportement d'un processus ET-parallèle

Un processus ET-parallèle calcule le sous-but droit d'une paire ET-parallèle. L'initialisation d'un processus ET-parallèle est semblable à celle d'un processus OU-parallèle ; le processus nouvellement créé prépare son retour-arrière final en chargeant dans le registre **BP** l'adresse de l'instruction *terminate_right*. Il pousse sur la pile locale une *rootframe*, dans laquelle sont stockés les paramètres du *terminate_right*, liée à la chaine des points de choix et des *branch-point*. De façon analogue, lorsqu'une solution au sous-but droit est produite, une instruction *join* est exécutée, accédant ses paramètres dans le *fork-point* du processus père ; pour cette raison, le registre pointeur d'environnement **E** est chargé initialement à l'adresse du *fork-point* dont le processus est issu.

Le produit croisé des solutions de deux sous-buts ET-parallèles est calculé par l'instruction *join* qui pousse sur la pile locale un *extensible-branch-point*, source éventuelle de parallélisme OU s'il y a plusieurs solutions au sous-but gauche. L'opération de produit croisé est similaire au parallélisme OU pur : les combinaisons d'une solution au sous-but droit avec les solutions au sous-but gauche peuvent être soit exécutées par de nouveaux processus OU-parallèles créés par des processeurs oisifs, soit par le processus ayant produit une solution à la partie droite lors d'un retour-arrière. L'instruction *cross_product* est utilisée par chacun des processus ayant produit une solution à la partie droite afin de se synchroniser avec les processus exécutant les autres alternatives. En ce sens elle est analogue au *par_retry* du parallélisme OU. Elle est exécutée pour la première fois en tant que retour-arrière du premier produit-croisé ; ce comportement est obtenu en chargeant le registre **BP** à l'adresse de l'instruction *cross_product* lors de l'exécution du *join*. Les paramètres de l'instruction *cross-product* se trouvent dans l'*extensible_branch_point*, qui doit donc être lié à la chaine des points de choix et des *branch-points*. Les instructions *join* et *cross_product* connectent par une *join-cell* deux des processus ayant produits des solutions à chacune des branches et exécutent la continuation de la conjonction des sous-buts ET-parallèles.

Optimisations

La combinaison des parallélismes OU et ET est complexe. Elle risque également dans de nombreux cas d'être peu efficace, si des mécanismes généraux sont mis en œuvre inutilement. Nous avons avons vu dans la section 4.4.1 que le parallélisme ET est simple si chacune des branches est déterministe. Lorsqu'une seule branche est déterministe, en l'exécutant comme branche gauche de la conjonction de prédicats ET-parallèles, on évite de constituer des listes de solutions de la partie gauche. De la même façon, si l'on peut déterminer statiquement que l'une des branches sera exécutée séquentiellement, on l'exécute comme branche gauche, ce qui rend inutile la création d'*extensible-branch-point*. Pour tenir compte des cas particuliers simplificateurs, des instructions *fork*, *join* et *check* spécialisées ont été définies. Elles prennent en compte les deux propriétés de séquentialité et de déterminisme dans une ou dans les deux parties ET-parallèles. Le compilateur génère ces instructions

```
Programme PEPSys                          Code PAM

-properties( [ solutions ( one ),...] ).  p/n:   allocate_oneof
pred
                                                  sync_oneof
p :- cl1.                                 pred:   < instructions
p :- cl2.                                           du predicat p
>
  ...

p :- cln.
```

Figure 5.4: Compilation d'un prédicat une-solution

L'exécution du prédicat p est "encapsulée" par les instructions *allocate_oneof* et *sync_oneof*.

spécialisées, à partir d'analyses simples basées sur les déclarations de propriétés des prédicats (voir la section 5.4).

5.2.5 Prédicats une-solution

Le langage PEPSys offre la possiblité de déclarer que certains prédicats ne doivent produire qu'une seule solution. Si plusieurs processus sont en compétition pour la solution du prédicat, un seul d'entre eux doit être autorisé à poursuivre le calcul, habituellement le premier à produire une solution. La synchronisation entre processus en compétition doit donc se faire à l'exécution de la **continuation** du prédicat. Un drapeau (*flag*) est associé à chaque prédicat une-solution : après le calcul d'une solution au prédicat, chacun des processus en compétition accède ce drapeau en exclusion mutuelle : le premier positionne le drapeau et continue le calcul ; les autres processus procèdent à un retour-arrière lorsqu'ils trouvent le drapeau positionné.

La machine abstraite PEPSys étend le contrôle de la *WAM* afin d'exécuter cette action de synchronisation en tant que continuation du prédicat une-solution. Cette extension est réalisée par deux instructions, *allocate_oneof* et *sync_oneof* qui utilisent un *oneof-point* contenant le drapeau de synchronisation (voir figure 5.4).

L'instruction *allocate_oneof* pousse un *oneof-point* sur la pile locale, le lie à la chaine des environnements, charge le registre de continuation **CP** à l'instruction suivante *sync_oneof* puis entame l'exécution du prédicat. Un processus exécutant un *sync_oneof* teste le drapeau de synchronisation du *oneof-point*, le positionne et

continue s'il est le premier (*test-and-set*, procède à un retour-arrière dans le cas contraire.

L'algorithme général mobilise des ressources inutilement, à partir du moment où la première solution a été obtenue. En effet, après la production de la première solution, les processus participant à son calcul continuent l'exécution jusqu'à ce que l'ensemble du sous-arbre de réfutation du prédicat ait été exploré, calculant inutilement l'ensemble des solutions à ce prédicat. Trois optimisations permettent une meilleure utilisation des ressources :

- *backbranching :* les processus contraints au retour-arrière suivent la chaine des *branch-points* actifs, c'est-à-dire qui sont à l'origine de processus encore actifs, jusqu'au *oneof-point*. Cette optimisation évite d'exécuter chacune des alternatives restantes des points de choix et *branch-points* devenus inutiles ; elle évite donc le calcul inutile de plus d'une solution par processus concerné. Cette solution a été implémentée sur le système multiprocesseur et sur le simulateur.

- *killing :* le processus qui produit la première solution provoque la terminaison brutale des processus en compétition qui ne participent pas à la solution. Ceux qui participent à la solution, ce qui est par exemple le cas des processus parents du "vainqueur", sont interrompus et forcés au *backbranching*. L'implémentation du *killing* est très complexe même si on se limite au parallélisme OU. Elle a été faite dans ce cas pour le simulateur [Baron 89].

- prédicats une-solution séquentiels : s'il est possible de déterminer statiquement, à la compilation, que le prédicat une-solution sera exécuté séquentiellement car il n'appelle aucun sous-but OU-parallèle, le prédicat est compilé de la même façon que s'il s'agissait d'un prédicat Prolog séquentiel auquel un *cut* terminal aurait été ajouté à chaque clause. Une telle analyse, utilisant les déclarations de propriétés du langage PEPSys, a été implémentée dans le compilateur PEPSys (voir section 5.4). Cette analyse permet d'optimiser l'exécution de la plupart des prédicats une-solution[4].

5.3 Gestion des liaisons

Contrairement à la gestion du contrôle, la gestion des liaisons ne nécessite pas la définition d'instructions n'existant pas dans la *WAM*. Par contre, elle se traduit par des extensions très importantes à de nombreuses instructions existantes. De même, les opérations basiques de la *WAM*, unification et déréférencement, sont très profondément modifiées. Deux structures de données nouvelles ont dû être définies pour la machine abstraite PEPSys : les *hash-windows* et les *join-cells*. Enfin, les objets manipulés par la machine abstraite PEPSys sont plus complexes et plus nombreux que les objets manipulés par la *WAM*.

[4] Ce qui n'empêche pas l'exécution séquentielle de la majorité des autres prédicats une-solution non optimisés.

5.3.1 Objets manipulés

La machine abstraite PEPSys utilise tous les types d'objets de la *WAM*. En plus des informations utilisées dans la *WAM*, ces objets comportent un champ destiné à contenir la valeur du compteur *OBL* (voir paragraphe 4.4.2) lorsqu'ils sont liés. En plus de ces types d'objets, de nouveaux types d'objets sont définis pour distinguer explicitement les liaisons non-locales. Pour ce faire, il existe dans la PAM un équivalent "non-local" pour chaque type d'objet de la *WAM* qui comporte un pointeur, à savoir : libre non-local[5] (variable Prolog), référence non-locale, liste non-locale, structure non-locale. Les objets PEPSys non-locaux contiennent les informations additionnelles suivantes : identification du processus créateur de l'objet et *split-OBL*, c'est-à-dire, valeur de l'*OBL* dans le processus créateur lorsque la branche courante s'en est séparée (voir figure 4.4).

5.3.2 Algorithme de liaison

L'opération de liaison de Prolog séquentiel est étendue dans PEPSys afin de prendre en compte les objets non-locaux ; ces objets sont *plus anciens* que n'importe quel objet local. Lorsqu'une variable locale est liée à une variable libre non-locale, elle est transformée en référence non-locale. Une variable libre non-locale est liée dans la *hash-window* du processus. Lorsque deux variables libres non-locales sont liées entre elles, elles sont toutes deux liées à une variable libre, créée localement sur la pile globale, ceci afin d'accroitre la localité de référence lors de la suite de l'exécution.

5.3.3 Algorithme de déréférencement

L'algorithme de déréférencement de la PAM implémente l'algorithme défini par le modèle de calcul. De la même façon qu'en Prolog séquentiel, le déréférencement suit la chaine locale de références. L'opération de déréférencement local s'arrête à la rencontre d'un objet autre qu'une référence. Si l'objet est une référence non-locale, le déréférencement se poursuit en **mode non-local** : la validité de chaque liaison est testée en comparant l'*OBL* de la référence non-locale avec celle de la liaison (voir figure 4.4).

Si la variable non-locale est liée dans la pile du processus créateur, mais que cette liaison n'est pas valide pour le processus déréférenceur, il est possible que cette variable soit liée, validement pour le processus déréférenceur, dans une *hash-window*. Les *hash-windows* à explorer sont celles du processus déréférenceur ainsi que celles de tous les processus fils du processus créateur de la variable qui sont aussi ancêtres du processus déréférenceur (voir figure 4.4). Si une liaison contenant un pointeur, local au processus où elle a été faite, est trouvée valide par l'opération de déréférencement non-local, elle est transformée en son équivalent non-local

[5] Aucun objet des piles ne possède ce type. Par contre, un objet de ce type est produit par l'algorithme de déréférencement dans certaines circonstances.

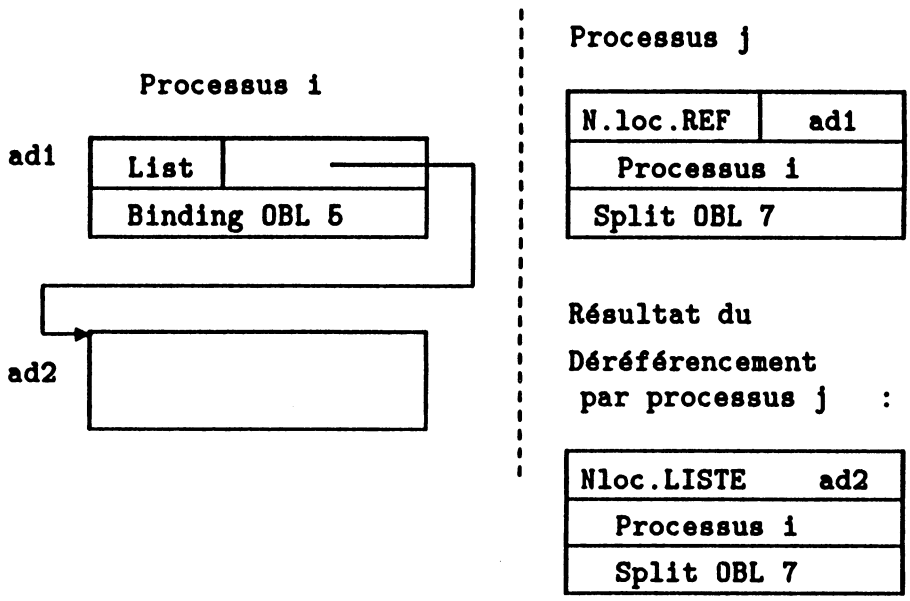

Figure 5.5: Déréférencement non-local

La référence non-locale à ad_1 est liée validement pour le processus j car la valeur de l'*OBL* de liaison (5) est inférieure à celle du *split-OBL* (7). Le déréférencement de ad_1 par le processus j retourne une liste non-locale ayant la même identification de processus i et le même *split-OBL* (7) que la référence non-locale.

avant d'être retournée au processus déréférenceur (voir figure 5.5). Lorsqu'une variable non-locale n'est pas liée validement pour un processus, l'algorithme de déréférencement retourne un objet **non-local libre**, susceptible d'être lié dans la *hash-window* du processus déréférenceur.

5.3.4 Extension de l'unification séquentielle

Unification générale

Une variable libre, locale ou non-locale, est susceptible de s'unifier à n'importe quel terme PEPSys. Dans le cas d'une variable non-locale, la liaison est faite dans la *hash-window* du processus. L'unification d'une liste (resp. structure) non-locale avec une liste (resp. structure) locale ou non-locale, est plus complexe. Prenons l'exemple de l'unification d'une liste non-locale avec une liste. De la même façon que pour une unification séquentielle, les têtes et queues de listes sont unifiées récursivement. Cependant, il faut transformer les références à la tête et à la queue de la liste non-locale en références non-locales, avant de les déréférencer ; cette transformation se fait en utilisant les identifications de processus ainsi que la valeur du *split-OBL* de l'objet liste non-locale (voir figure 5.6).

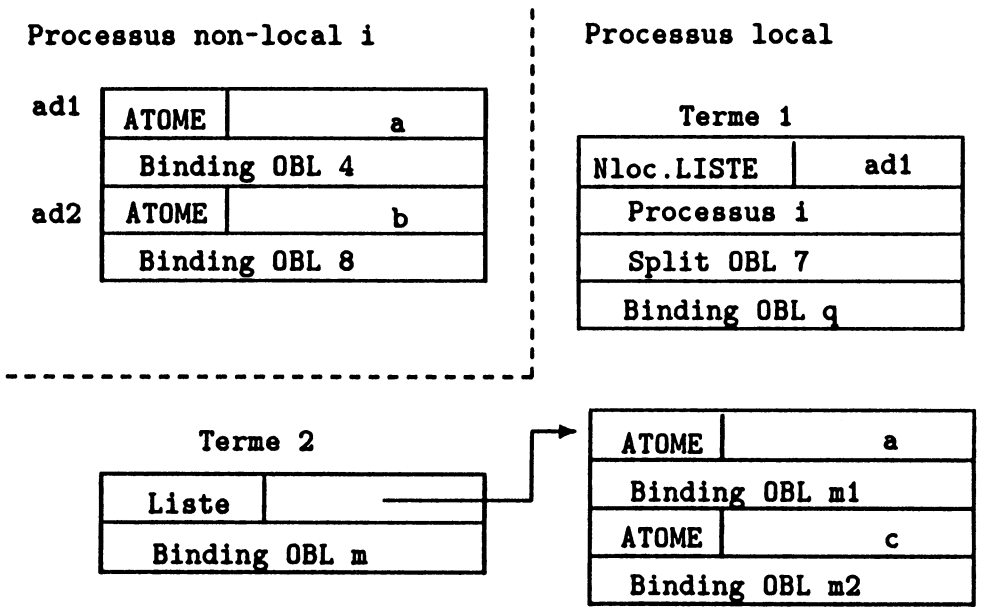

Figure 5.6: Unification d'une liste non-locale et d'une liste

L'unification de $Terme_1$ et de $Terme_2$ est pratiquée par le processus local. Avant l'unification, $Terme_1$ est lié à [a | _] puisque la liaison de la queue de la liste non-locale est invalide. L'unification provoque la liaison de l'adresse non-locale ad_2 à la constante c, dans la *hash-window* du processus local.

Unification compilée

Bien souvent l'unification est partitionnée en plusieurs instructions par les compilateurs Prolog. Encore une fois, la PAM étend les algorithmes de la *WAM* au cas des objets non-locaux. Prenons à nouveau l'exemple de l'unification de listes. Cette unification est habituellement compilée en instruction *get_list* suivie de deux instructions *unify*. La fonction du *get_list* est de tester si l'argument déréférencé est bien de type cellule de liste, puis d'affecter au registre pointeur de structure, S, l'adresse de la tête de liste. Les instructions *unify* unifient les têtes et queues de listes, accédées à l'aide du registre S. Dans la PAM, l'instruction *get_list* réussit également si le résultat du déréférencement est une cellule de liste non-locale. Cependant il faut alors initialiser le registre S à une référence non-locale, de telle sorte que les déréférencements de la tête et de la queue de la liste, pratiqués dans les instructions *unify* qui suivent, possèdent les informations suffisantes pour pratiquer les tests de validité et les transformations nécessaires. En conséquence, dans la PAM, le registre *S* peut être de type **local ou non-local.**

5.3.5 Création d'objets non-locaux

Les objets non-locaux manipulés par un processus sont initialisés à la création du processus. Cette opération se fait en deux phases : d'une part les registres arguments du premier sous-but à exécuter par le nouveau processus sont transformés, si besoin est, en objets non-locaux. D'autre part, le registre pointeur d'environnement E est également transformé en référence non-locale contenant l'adresse de l'environnement courant du processus père, ce qui permettra d'opérer la même transformation sur les arguments de tous les sous-buts restant à exécuter (voir figures 5.7 et 5.8) ; de cette façon les environnements d'appel créés par un processus sont partagés par tous les processus qui en sont issus. Une précaution particulière doit être prise lors de la libération d'un environnement non-local ainsi partagé[6] ; en Prolog séquentiel, le registre pointeur d'environnement E est restauré à sa valeur précédente, sauvegardée dans l'environnement libéré ; en PEPSys, il est nécessaire de conserver, lors de cette restauration, le caractère non-local du registre E, afin de pouvoir "transformer" en arguments non-locaux les arguments des sous-buts de la continuation, selon le mécanisme décrit précédemment. Les instructions de la *WAM* qui ont été étendues pour assurer cette transformation sont les instructions de queue de clause *put* et *unify*.

[6] L'espace de pile non-locale correspondant n'est pas physiquement libéré puisqu'il appartient à un processus ancêtre.

```
-properties( [ ... ] ).
(Cl0)  parallel( Parametre, Resultat ) :-
            creer_parallelisme( Parametre, Intermediaire ),
            utiliser_parallelisme( Intermediaire, Resultat ).

-properties( [ execution ( eager ), clauses ( unordered ) , ...] ).

(Cl1)  creer_parallelisme( Source, Objet ) :- ...
(Cl2)  creer_parallelisme( Source, Objet ) :- ...
```

Figure 5.7: Création d'objets non-locaux

L'exécution parallèle de cet exemple résulte en la création de références non-locales à deux occasions. La première est lors de la création d'un processus pour exécuter la clause Cl_2 : les arguments stockés dans le *branch-point*, qui sont des références locales à l'environnement de la clause Cl_0, sont transformées en références non-locales. La seconde est lorsque le nouveau processus exécute la continuation de Cl_2 (utiliser_parallélisme) : le registre pointeur d'environnement, **E**, est alors non-local, permettant aux instructions *put* d'initialiser les registres arguments à des références non-locales.

5.4 Compilateur PEPSys

5.4.1 Présentation générale

Le travail fait dans le projet ICM3 [Noye 87] a été utilisé autant que possible dans le projet PEPSys. Comme il a été mentionné, la spécification détaillée de la machine abstraite ICM3 a servi de base à la définition de la machine abstraite PEPSys. Le compilateur Prolog du projet ICM3 a également été étendu pour compiler des programmes PEPSys. Le compilateur ICM3 est écrit en Prolog. Il génère du code objet très efficace en raison d'une indexation et d'une allocation de registres sophistiqués. Son utilisation permet de bénéficier pour l'implémentation parallèle des techniques de compilation les plus efficaces de Prolog séquentiel.

Le compilateur PEPSys a été originellement conçu pour ne compiler que les **modules parallèles** des programmes PEPSys et produire du code exécutable par le simulateur et le système parallèle. Le code généré par le compilateur PEPSys comprend, en plus des instructions de la machine abstraite ICM3, les instructions nouvelles de la machine abstraite PEPSys, définies pour traiter le parallélisme OU et ET. Comme il apparait dans les exemples de compilation donnés dans la section précédente, ces instructions sont "insérées" à des emplacements spécifiques du code objet généré par le compilateur ICM3. En plus de ces insertions et de la prise en compte des déclarations de propriétés et de l'opérateur d'indépendance, les principales extensions au compilateur ICM3 ont été des analyses statiques de programmes sources, afin de générer des instructions optimisées pour les prédicats

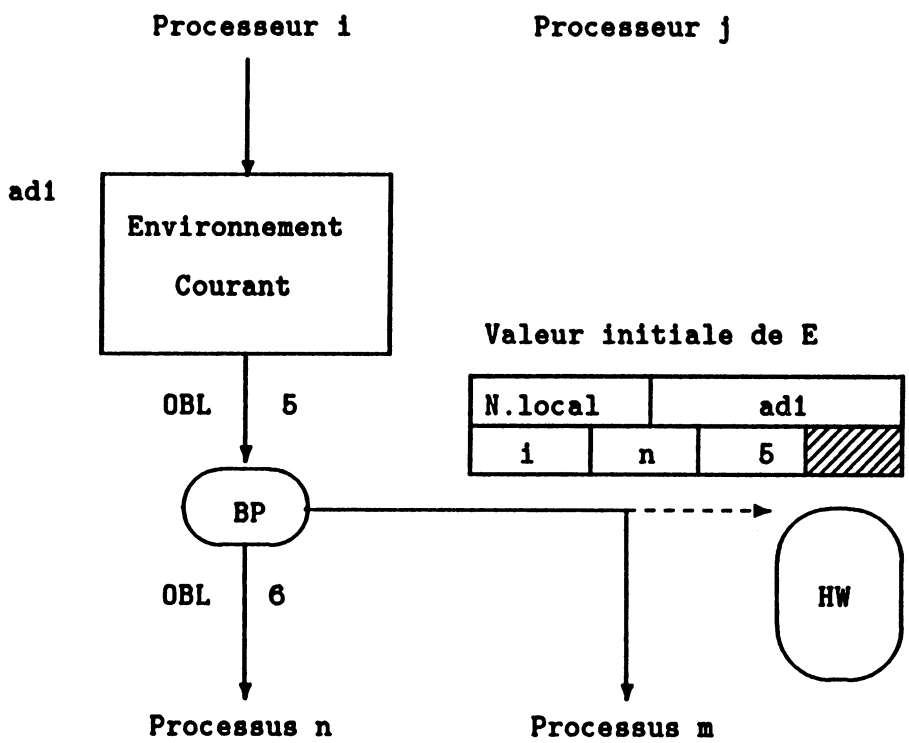

Figure 5.8: Exemple de typage du registre pointeur d'environnement **E**

Le contenu initial du registre E du processus *m* s'exécutant sur le processeur *j* permet de tester la validité des liaisons des cellules de variables de l'environnement courant (non local) ainsi que de transformer les arguments des sous-buts de la continuation. Il contient également les informations permettant d'accéder au processeur non local (processeur *i*, processus *n*) lorsque l'architecture utilisée n'a pas de mémoire commune (architecture de type NORMA).

"une-solution" et les conjonctions ET-parallèles.

5.4.2 Langage reconnu par le compilateur

Le compilateur PEPSys compile les modules parallèles. Il prend en compte les déclarations de propriétés, la modularité entre modules parallèles uniquement ainsi que l'opérateur d'indépendance. Mises à part ces extensions, le langage reconnu est le Prolog "standard" utilisé dans le projet ICM3.

Déclarations de propriétés

Chaque prédicat d'un programme source PEPSys doit être précédé d'une déclaration de propriétés, éventuellement vide //, qui est alors compilé par défaut avec la sémantique de Prolog :

[solutions(all), clauses(ordered), execution(lazy)]

Les instructions permettant le parallélisme OU sont générées à la rencontre des propriétés :

[clauses(unordered), execution(eager)]

La conservation de l'ordre des solutions de Prolog lors d'une exécution OU-parallèle n'étant pas définie dans la machine abstraite PEPSys, la combinaison de propriétés :

[clauses(ordered), execution(eager)]

sera compilée comme :

[clauses(ordered), execution(lazy)]

et le prédicat correspondant considéré comme séquentiel.

Opérateur d'indépendance

Pour des raisons purement techniques liées à la façon dont sont réalisées les analyses statiques (voir section 5.4.3), il n'est pas possible de relier plus de deux sous-buts par l'opérateur d'indépendance. Un utilisateur désireux d'écrire :

$$l(X) \# r_1(Y) \# r_2(Z)$$

devra transformer cette conjonction en :

$$l(X) \# r(Y, Z)$$

avec

$$r(Y, Z) :- r_1(Y) \# r_2(Z)$$

Il est par contre possible d'avoir, dans une même clause, plusieurs conjonctions ET-parallèles :

$$...l_1(X) \# r_1(Y),...,l_2(X) \# r_2(Z), ...$$

Modularité

Seule la modularité entre modules parallèles a été implémentée [Rapp 88]. Le compilateur vérifie que tous les prédicats exportés sont bien définis dans le module et que tous les prédicats non définis dans le module sont bien importés. En cas d'erreur, seul un message d'avertissement est imprimé. De plus, le compilateur génère des tables permettant la vérification des connections inter-modules, à l'édition de liens.

Cut

Les *cuts* présents dans le programme source sont compilés suivant les techniques définies pour ICM3. Cette technique permet d'optimiser la compilation des prédicats une-solution dont l'exécution ainsi que celle de leurs fils est séquentielle, - un *cut* terminal est inséré à la fin de chaque clause qui le compose -. Elle permet aussi de compiler, sous la responsabilité de l'utilisateur, des programmes Prolog existants, dans lesquel des *cut* ne sont utilisés que dans des prédicats exécutés séquentiellement.

5.4.3 Analyses statiques

L'intérêt d'analyses statiques simples a déjà été mentionné à plusieurs reprises. Elles permettent d'optimiser l'implémentation des prédicats une-solution utilisés dans des branches séquentielles, ainsi que de générer des instructions simplifiées pour le parallélisme ET. Des analyses statiques très simples, basées sur les déclarations de propriétés, ont été implémentées dans le compilateur PEPSys.

L'analyse de déterminisme est faite pour tous les prédicats reliés par l'opérateur d'indépendance. Elle consiste à tester les déclarations de propriétés des prédicats concernés. Si elle peut être établie pour les deux prédicats d'une conjonction elle supprime la nécessité du produit croisé ; si c'est le cas d'un seul d'entre eux, elle le simplifie considérablement.

L'analyse de séquentialité est faite pour les prédicats une-solution, les prédicats paramètres du prédicat prédéfini *oneof* et les prédicats reliés par l'opérateur d'indépendance, si aucun d'entre eux n'est déterministe[7]. Les déclarations de propriétés du prédicat analysé et l'ensemble des prédicats que son exécution est susceptible d'appeler, - fermeture transitive de la relation "fils" -, sont testées. Si aucun d'entre eux n'est déclaré :

[clauses(unordered), execution(eager)]

le prédicat analysé est considéré comme séquentiel.

Les analyses statiques sont limitées à un seul module parallèle. Malgré leur simplicité, elles permettent d'optimiser le code généré dans la plupart des cas.

[7] Le déterminisme est une propriété plus "forte" que la séquentialité, pour le parallélisme ET. Si l'un des prédicats ET-parallèles est déterministe, il est donc inutile d'effectuer une analyse de séquentialité, longue à cause de la recherche de tous les "descendants".

Elles n'en demeurent pas moins insuffisantes puisqu'elles ne trouvent pas toutes les optimisations que le programmeur pourrait indiquer. De plus elles devraient être étendues à plusieurs modules.

5.4.4 Assemblage, édition de liens

Le compilateur PEPSys génère des instructions de machine abstraite sous forme symbolique. Deux formats sont possibles suivant que le programme source est compilé pour le simulateur ou pour l'implémentation parallèle. Le simulateur, écrit en Lisp, traite directement le programme objet, mis sous forme de S-expressions. Par contre, les modules compilés au format "assembleur" doivent être assemblés et édités pour générer du code binaire chargeable par le système parallèle. L'assemblage réordonne les paramètres des instructions d'indexation des "gros" prédicats "bases de données" afin que le choix de la clause candidate se fasse par adressage dispersé (*hash-code*) portant sur la valeur du premier argument. L'édition de liens vérifie que les prédicats importés par certains modules sont bien exportés par d'autre. Elle vérifie également les doubles définitions. De même qu'à la compilation, les erreurs ne font l'objet que d'avertissements.

5.5 Résumé du chapitre

Une machine abstraite Prolog dérivée de la *WAM*, ICM3, a servi de base à la définition d'une machine abstraite pour la compilation de PEPSys. Les extensions ont tout d'abord consisté à définir de nouvelles instructions et structures de données permettant aux processus qui ont du travail à offrir de le mettre à la disposition des autres processeurs tout en étant perturbés le moins possible. Les nouvelles instructions suivent le contrôle défini par le modèle de calcul. D'autres extensions modifient considérablement les opérations d'unification et de déréférencement de la *WAM*, afin d'y incorporer la gestion variables partagées définie par le modèle de calcul. Enfin des extensions moins visibles ont été faites à certaines instructions existantes, afin de tenir compte de la gestion des variables du modèle de calcul. Le compilateur ICM3 a été étendu pour accepter le langage PEPSys et générer les nouvelles instructions. Il procède en outre à des analyses statiques, afin d'optimiser l'implémentation des prédicats "une-solution" et du mélange du parallélisme ET avec le parallélisme OU.

Chapitre 6

Implémentation de PEPSys sur MX500

6.1 Motivations

Pour valider les concepts du langage et du modèle de calcul PEPSys, une implémentation et un simulateur ont été réalisés. Le modèle de calcul de PEPSys n'impose pas d'architecture multiprocesseur particulière ; en effetles références aux variables non-locales y sont explicites ce qui rend possible son implémentation sur toute architecture multiprocesseur, indépendamment des connexions entre processeurs et mémoire. Etant donnée la généralité du modèle, il a tout d'abord semblé logique d'évaluer le modèle en simulant une large variété d'architectures, afin de mettre en évidence celle qui s'adapte la mieux au modèle PEPSys. Cependant il est rapidement apparu qu'une implémentation sur un multiprocesseur constituerait un complément utile au travail de simulation. En effet, la simulation ralentit habituellement de plusieurs ordres de grandeur le temps d'exécution, ce qui rend difficile le test de longs programmes dont le parallélisme vise précisément à accélérer l'exécution. De plus des résultats de simulation ne sont pas aussi probants que des résultats d'implémentation ; s'ils peuvent fournir une bonne approximation des gains de performances qu'on est en droit d'attendre du parallélisme (facteur d'accélération), ils peuvent difficilement donner des résultats "absolus". Enfin, une implémentation sur multiprocesseur est le seul moyen d'explorer l'ensemble des problèmes que posent le développement et la mise au point de programmes s'exécutant en parallèle ; en effet le parallélisme d'un simulateur séquentiel n'est pas réel et son exécution déterministe rend sa mise au point plus simple que celle d'un système réellement parallèle.

Ce chapitre décrit l'implémentation de PEPSys sur le multiprocesseur à mémoire commune Siemens MX500. Cette implémentation comporte les parallélismes OU et ET déterministe. Les prédicats "une-solution" ont été également implémentés, en utilisant la technique du *backbranching* (voir section 5.2.5). Le système décrit a été ultérieurement connecté au système Prolog séquentiel SEPIA [Meier 89]. Il a ensuite servi de base à une implantation parallèle d'une partie du système de pro-

grammation logique avec contraintes CHIP. Par manque de temps, la combinaison des parallélismes OU et ET n'a pu être implémentée et testée. Pour cette raison, l'implémentation des prédicats "une-solution" ne comporte pas la destruction des processus devenus inutiles.

Le plan de ce chapitre est le suivant. On présente tout d'abord les raisons qui ont présidées au choix du Siemens MX500 pour l'implémentation de PEPSys. On donne ensuite l'architecture du système PEPSys, adaptée à celle du MX500. Les données et les algorithmes utilisés pour la mise en œuvre efficace de la machine abstraite PEPSys sont ensuite décrits. On aborde ensuite le partage du travail entre processeurs, implémenté par un noyau d'ordonnancement s'exécutant sur chaque processeur. La section suivante est consacrée aux techniques utilisées pour la mise au point du système parallèle. Enfin, les extensions au système de base, prédicats prédéfinis du système parallèle, "connexion" du système parallèle à un système Prolog séquentiel et implantation des variables domaines pour permettre la programmation logique avec contraintes, font l'objet de la dernière section de ce chapitre.

6.2 Choix d'un multiprocesseur

Les deux principaux critères de choix d'un multiprocesseur pour implémenter PEPSys ont été, outre le coût, l'adaptation au modèle et la facilité de mise en œuvre, afin d'obtenir des résultats dès que possible. Nous avons vu que le modèle PEPSys a été défini pour permettre son implémentation sur une grande variété d'architectures multiprocesseurs MIMD. Cela ne veut pas nécessairement dire qu'une implémentation sera efficace ou facile à réaliser sur n'importe quel multiprocesseur. Nous allons donc tout d'abord donner les principales caractéristiques des différentes classes de multiprocesseurs MIMD existants actuellement et indiquer les raisons du choix du MX500 en fonction des critères ci-dessus. Nous conclurons cette section par une brève description de la machine retenue.

6.2.1 Classification des multiprocesseurs MIMD

On peut distinguer trois principales classes de multiprocesseurs MIMD :

- **Uniform Memory Access (UMA)** dans lesquels le bus et la mémoire sont communs à tous les processeurs. Pour éviter que ces éléments ne constituent des goulots d'étranglement, chaque processeur est doté d'une mémoire cache. La définition d'algorithmes de cohérence de caches performants est un important sujet de préoccupation des chercheurs en architecture de machines. La taille des machines reste limitée à quelques dizaines de processeurs. Plusieurs constructeurs produisent des machines de ce type, les principaux étant Sequent (modèles Balance et Symmetry) et Encore (modèle Multimax).

- **Non Uniform Memory Access (NUMA)** dans lesquels la mémoire est physiquement distribuée mais logiquement partagée entre les unités de traitement (Processing Element). Différentes connexions sont possibles :

  - arborescence de commutateurs reliant les unités de gestion mémoire de chaque unité de traitement. C'est le cas du Butterfly de BBN dans lequel il y a deux temps d'accès possibles à une donnée suivant qu'elle est locale (0,5 micro-seconde) ou non (5 micro-secondes).
  - RP3 [Pfister 85] combinaison de deux réseaux, l'un à faible temps de réponse, composé de quatre niveaux de connecteurs 4*4 et connectant 64 processeurs disposés en H, l'autre à haut débit, composé de six niveaux de connecteurs 2*2 et reliant huit grappes de 64 processeurs. Les rapports entre les temps d'accès à une donnée située dans le cache local, dans la mémoire locale, ou accédée à travers le réseau sont respectivement de 1, 10 et 15.
  - arborescence de grappes de processeurs. C'est le cas de l'Encore Ultramax [Wilson 87], de la machine PIM du projet japonais Cinquième Génération [Chikayama 87] et de la Data Diffusion Machine [Haridi 89]. Le temps d'accès à une donnée dépend de la proximité du processeur qui l'accède : locale, autre unité de traitement de la même grappe, grappe voisine, etc.

  L'architecture NUMA permet de connecter quelques centaines de processeurs. La mise en œuvre des machines de ce type est plus délicate que celle des machines UMA car l'efficacité des algorithmes dépend de la localisation des programmes et des données qui est transparente à l'utilisateur.

- **NOn Remote Memory Access (NORMA)** composée d'unités de traitements communiquant entre elles par message uniquement. Chaque unité de traitement comprend un processeur et sa mémoire privée. Plusieurs topologies sont possibles pour le réseau d'interconnexion des unités de traitement. L'accès à une donnée locale est plus rapide que dans les autres architectures puisqu'il est possible d'utiliser les mêmes techniques de caches que pour les mono-processeurs (temps d'accès de quelques dizaines de nano-secondes). L'accès à une cellule mémoire non locale suppose par contre un échange de messages. Le temps de transfert d'un message est de l'ordre de 1 milliseconde sur les multiprocesseurs existants, la plus grande partie de ce temps étant consommée par le logiciel de composition du message. L'incorporation de primitives de communication au jeu d'instruction devrait permettre de réduire ce délai de façon considérable (voir la J-machine[Dally et Wills 89]). Un certain nombre de multiprocesseurs ont choisi ce type d'architecture. Outre la J-machine en cours de réalisation, on peut citer l'iPCS/2 de Intel ainsi que les architectures à base de transputers. Une architecture de type NORMA est susceptible de connecter plusieurs milliers de processeurs.

6.2.2 Choix du MX500

La seule contrainte imposée par le modèle est la possiblité d'accéder depuis un processeur à des données gérées par d'autres processeurs. Les accès distants étant supposés peu fréquents, en raison de la bonne localité à l'exécution indiquée par les analyses dynamiques de programmes Prolog, il est admissible qu'ils soient plus coûteux que les accès aux données locales.

Il ne semble pas que les multiprocesseurs NORMA soient bien adaptés à l'implémentation de PEPSys. En effet, le rapport entre l'accès à une variable locale et une variable distante est de mille environ. Ce qui signifie qu'un pourcentage de 1 % d'accès non-locaux multiplierait par 10 le temps d'accès aux variables, le facteur de ralentissement passant à 100 pour 10 % d'accès non-locaux[1]. On peut limiter le ralentissement occasionné par les accès aux données distantes en ayant recours à la multiprogrammation sur chacun des éléments de traitement mais c'est alors au prix d'un surcoût et d'un accroissement de complexité supplémentaires. La recherche de travail par un processeur oisif nécessiterait également de nombreux échanges de messages et semble difficile à implémenter efficacement sur ce type d'architecture. Les systèmes d'exploitation et outils de développement disponibles sur ces machines sont encore rudimentaires. La mise au point sur ce type de machines est particulièrement délicate. En effet, il semble difficile d'arrêter l'ensemble de la machine rapidement. A supposer que cela soit possible, l'inspection de l'état de la machine depuis un processeur nécessite l'envoi de messages aux autres processeurs, ce qui a pour effet de changer l'état du système de communication. Enfin rien ne permet d'affirmer, faute d'expérience sur des configurations plus limitées, que les applications parallèles existantes sont à même de tirer parti de plusieurs milliers de processeurs.

Les multiprocesseurs NUMA semblent bien se prêter à l'implémentation de PEPSys, surtout dans le cas des programmes présentant une bonne localité. Cependant les multiprocesseurs les mieux adaptés étaient soit inaccessibles (l'IBM RP3 est un prototype, l'Ultramax est en cours de développement et il n'est pas encore prévu de construire la Data Diffusion Machine), soit chers et difficilement utilisables (Butterfly avant que le système MACH n'y soit implémenté. Voir à ce sujet [LeBlanc 88]).

Les architectures UMA sont celles qui donnent les meilleures perspectives de gain d'efficacité en utilisant le parallélisme puisque l'accès à une variable non-locale n'implique pas d'autre sur-coût que celui qui provient du modèle de calcul. Elles sont également les plus faciles à utiliser en raison du niveau de développement de leurs systèmes d'exploitation, proche de celui des machines séquentielles. Enfin une machine de ce type figure au catalogue d'un des actionnaires de ECRC : le Siemens MX500. Notre choix s'est donc naturellement porté sur cette machine.

[1] Les mesures faites ultérieurement ont montré que ce pourcentage n'était pas rare dans les programmes de tests utilisés (voir chapitre 8).

6.2.3 Configuration du MX500 utilisé

Le Siemens MX500 est en fait le Sequent Balance 8000 construit sous license par Siemens. Il est composé de plusieurs unités de traitements connectées par un bus commun à une mémoire également commune. Le nombre maximum de d'unités de traitement, 12 pour le Balance 8000, a été limité à 8 par Siemens. Chaque unité de traitement comporte un processeur NS32032 et une mémoire cache de 8 K-octets. L'adressage du NS32032 se fait sur 3 octets, ce qui lui donne un espace d'adressage de 16 Mega-octets. Chaque processeur est crédité d'une vitesse variant entre 0,5 et 0,7 (VAX 780) MIPS selon les auteurs. Le bus, défini par Sequent, a un débit de 27 Mega-octets par seconde. La configuration acquise par ECRC comporte 8 processeurs et 16 Mega-octets de mémoire commune, maximums possibles pour le MX500.

Le système d'exploitation fourni par Siemens sur le MX500 est Sinix, version Siemens de UNIX. Cependant, ce système ne supporte pas la librairie parallèle pour le langage C ni le debogueur symbolique PDBX, tous deux réalisés par Sequent sur le système Dynix. Le matériel du MX500 ne permet pas non plus de démarrer (*bootstrap*) le système Dynix. En conséquence, il a fallu utiliser le système Sinix combiné avec un certain nombre de composants du système Dynix : le compilateur C, la librairie parallèle C et le debugger symbolique PDBX.

6.3 Architecture du système parallèle

6.3.1 Processeurs virtuels

L'implémentation utilise le système d'exploitation du MX500, combinaison de Sinix et Dynix. Sur chaque processeur s'exécute un processus du système d'exploitation ; ce processus, jouant le rôle de **processeur virtuel**, sera appelé processeur dans le reste de la thèse, la notion de processus étant réservée à des structures de contrôle de niveau plus fin. Chaque processeur virtuel exécute la machine abstraite PEPSys décrite au chapitre précédent.

6.3.2 Implémentation de la machine abstraite PEPSys

La machine abstraite PEPSys est émulée par un émulateur d'instructions de la machine abstraite, écrit en C. Une telle implémentation représente un bon compromis entre la simplicité de mise en œuvre et l'efficacité. Les implémentations existantes de Prolog (voir figure 2.3) montrent que ce type d'implémentation est deux à trois fois plus efficace qu'un interprète classique. Un autre résultat important est qu'il est possible d'en multiplier les performances par un facteur compris entre deux et trois en réécrivant en assembleur la boucle d'émulation. Un autre facteur du même ordre peut à nouveau être gagné si le compilateur est modifié pour générer du code de la machine cible.

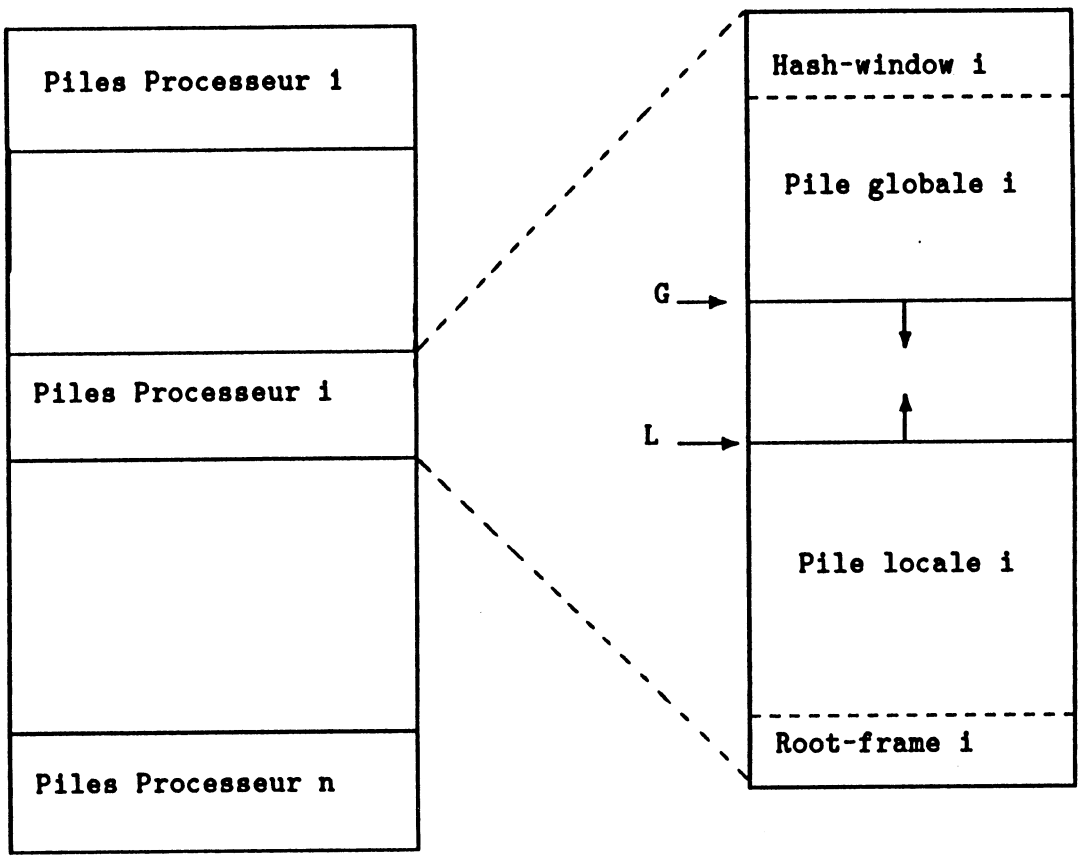

Figure 6.1: Piles locales et globales de l'implémentation

Le code de l'émulateur est partagé entre les processeurs, ainsi que l'espace des piles locales et globales. Un large espace est alloué pour ces dernières. Cet espace est divisé en autant de portions qu'il y a de processeurs utilisés. En cas de débordement de capacité pour les piles de l'un des processeurs, il n'est pas possible de modifier les limites entre les espaces alloués à chaque processeur pour rééquilibrer l'espace consommé. Logiquement, chaque processeur a accès exclusif en écriture à une portion et en lecture à l'ensemble de l'espace (voir figure 6.1). A l'intérieur de chaque portion, les piles locales et globales croissent en sens inverse afin de n'avoir de débordement de capacité que lorsque l'ensemble de la portion est utilisé. Les *root-frames* et les *hash-windows* sont allouées sur les piles locale et globale du processeur, à la création de chaque nouveau processus.

Le programme source est également partagé entre les processeurs. Comme dans les implémentations de la WAM, il se décompose en une liste d'instructions et un dictionnaire contenant les symboles de fonctions et atomes (symboles de fonctions d'arité zéro) du programme. Le code opération de chaque instruction occupe un mot machine. L'intérêt de cette technique (*threaded code*) est d'accélérer la boucle d'émulation. Elle a évidemment pour inconvénient d'accroître la taille des programmes objets en mémoire. Néanmoins, la taille du code objet demeure inférieure

à celle qu'elle ferait si le compilateur générait des instructions de la machine cible.

Les registres de la machine abstraite ainsi que la pile de restauration sont propres à chaque processeur.

6.3.3 Noyau d'ordonnancement

Un noyau d'ordonnancement s'exécute sur chacun des processeurs. Il a pour but d'alimenter le processeur en travail "utile", c'est-à-dire contribuant à l'exécution du programme utilisateur. Pour ce faire le noyau créé des **processus** qui s'exécutent en fait comme des coroutines, c'est-à-dire qu'un nouveau processus n'est créé qu'à la suspension du processus courant, ce dernier n'étant relancé que lorsque le nouveau processus a fini de s'exécuter.

Le code du noyau est partagé entre les processeurs. Les données qu'il utilise sont essentiellement des structures de contrôle créées sur la pile par les processus (*branch-point* et *fork-point*) ou par lui-même (*rootframes*) à la création des processus. Ces structures sont chainées entre elles, formant un arbre correspondant à l'arbre de réfutation du programme (voir figure 6.2). La recherche de travail par le noyau d'ordonnancement, pour le compte d'un processeur oisif, consiste en un parcours de cet arbre afin de sélectionner, parmi tous les travaux possibles (*branch-points* offrant des alternatives dans le cas du parallélisme OU), celui qui présente les meilleures perspectives de granularité importante. Différentes heuristiques de choix de travail ont été expérimentées et sont décrites dans la section 6.5.

6.4 Implémentation de la Machine Abstraite PEP-Sys

L'implémentation de la machine abstraite PEPSys suit étroitement le schéma décrit au chapitre précédent. Les principaux problèmes à résoudre ont été la définition de structures de données adaptées au modèle et l'implémentation efficace du déréférencement.

6.4.1 Données

Représentation d'élément de piles locales ou globales

Comme nous l'avons vu à la section 5.3, les objets manipulés sur les piles peuvent être subdivisés en objets locaux et non-locaux. Tous ces objets disposent d'un champ valeur. Ils sont identifiés par une étiquette (*tag*). A l'exception des variables libres, ils comportent un champ *OBL* indiquant la "date" de la liaison. En plus de ces champs, les objets non-locaux contiennent un *split-OBL* et l'identification du processus ayant créé la variable non-locale, que nous représentons par un couple processeur-processus. Sur un multiprocesseur à mémoire partagée, ce couple n'est utilisé que pour tester la fin de la chaine de *hash-windows* à explorer, dans les

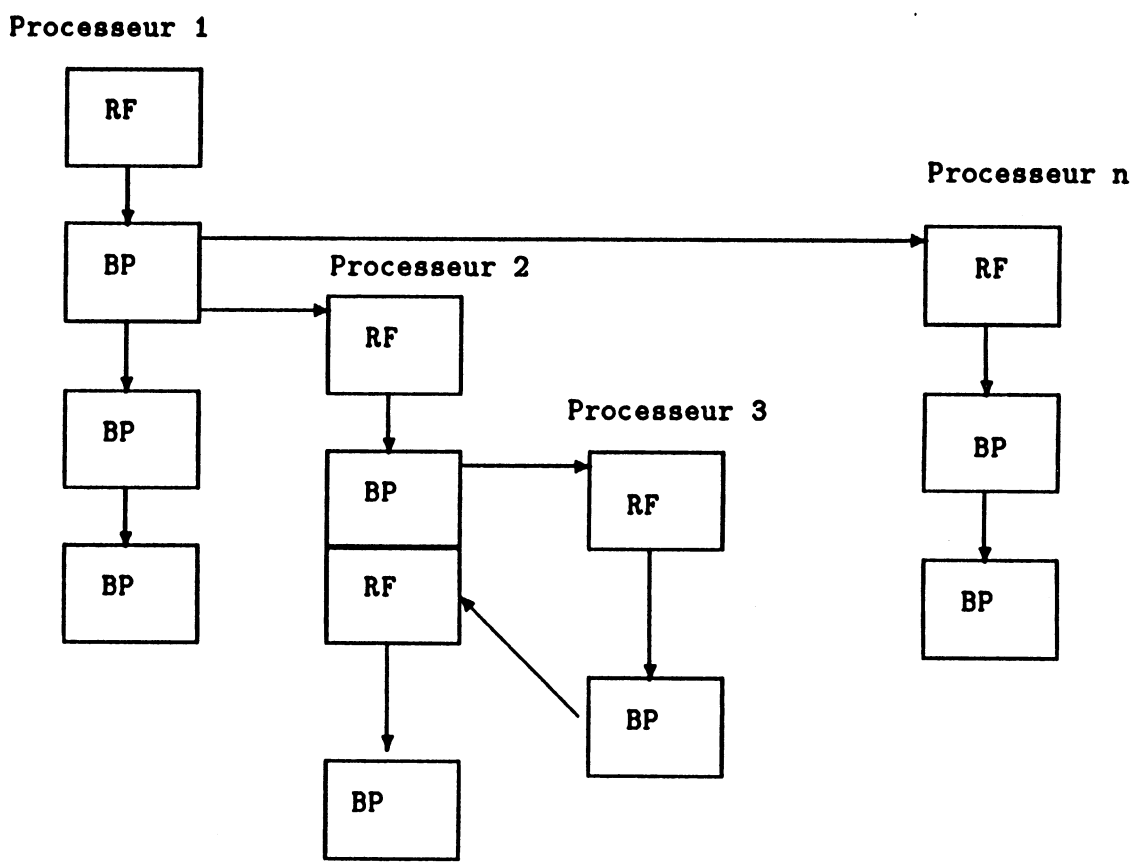

Figure 6.2: Arborescence des structures de contrôle utilisées pour la recherche de travail dans le cas du parallélisme OU

Les *rootframes* **RF** sont construites par le noyau, lors de la création d'un nouveau processus. Les *branch-points* **BP** sont créés par les processus, quand ils exécutent l'instruction *par_try*. Le processus initial du processeur 2 étant suspendu dans son retour-arrière, le processeur 2 "aide" le processus qui bloque le retour-arrière et qui s'exécute sur le processeur 3.

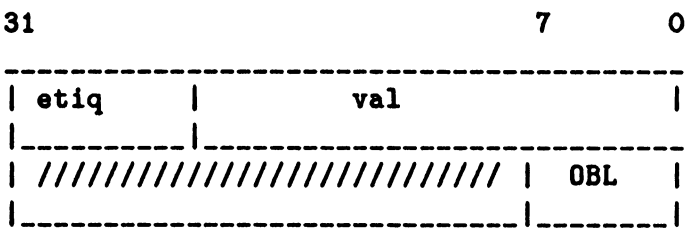

Figure 6.3: Format d'un objet local de pile (locale ou globale)

Les objets locaux sont les mêmes que ceux de la WAM : variables libres, références, atomes, entiers, listes, structures, etc.

déréférencements qui le nécessitent ; il peut donc éventuellement être remplacé par l'adresse de la *hash-window* du processus non-local.

Pour définir l'organisation d'un élément des piles locale ou globale, nous disposons d'un certain nombre de contraintes. Le champ valeur doit pouvoir contenir une adresse c'est-à-dire au moins trois octets. Le nombre d'étiquettes différentes est de dix au minimum mais doit pouvoir être étendu pour supporter plus de types de données. Le nombre de processeurs possibles doit être d'au moins huit. Nous n'avons pas d'indication sur le nombre maximum de processus par processeur ni sur la valeur maximale des champs *OBL* et *split-OBL* qui dépendent du nombre maximum de *branch-points* susceptibles d'exister simultanément sur chaque processeur.

Compte tenu des contraintes ci-dessus, deux mots mémoires de 32 bits sont nécessaires pour chaque objet non-local (voir figure 6.4). A condition de limiter la création de *branch-points* et conséquement la croissance de l'*OBL*, il est possible de faire tenir les objets locaux sur un seul mot. Cependant, cette solution ne permet pas d'affecter directement un objet non-local à un objet local, par exemple lors de l'unification d'une variable libre à une liste ou une structure non-locale. La solution adoptée par le simulateur [Robert 88] consiste à affecter à l'objet local un relais d'indirection vers un emplacement créé sur la pile globale. Pour éviter cette indirection, nous avons choisi d'utiliser également deux mots pour les objets locaux (voir figure 6.3). Cette solution a de surcroît l'avantage d'être moins limitative pour la valeur maximale du champ *OBL* et par là du nombre de *branch-points* par processus. En outre, la taille de deux mots est bien adaptée au matériel puisque c'est celle du bus de communication et que la lecture ou l'écriture de deux mots mémoire[2] est une opération atomique sur le MX500.

La solution retenue n'est pas directement portable sur une machine à 32 bits d'adresse. Un tel portage ne pose cependant aucun problème puisqu'il suffit de transférer le champ étiquette sur le second mot des objets de pile. Les résultats

[2] Alignés sur une frontière de double mot.

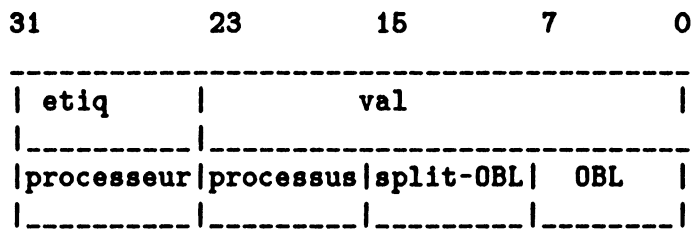

Figure 6.4: Format d'un objet non-local de pile (locale ou globale)

Les objets non-locaux de ce type sont : variables non-locales libres (objet retourné par le déréférencement mais jamais empilé), références, listes et structures non-locales.

expérimentaux collectés par l'implémentation parallèle (voir section 8.5.3) permettent de redéfinir la taille de chacun des champs des objets de pile afin de réaliser ce portage sans perte de fonctionnalité.

Registres

Registres d'état de la machine abstraite Les registres d'état de la machine sont les mêmes que les registres de la WAM à l'exception des registres **E** et **S** qui sont typés. Les deux types possibles sont :

référence : en plus du champ valeur qui contient l'adresse de l'environnement courant, seul le champ *etiq* a une sens.

référence non-locale : le champ valeur contient alors l'adresse de l'environnement courant, sous le même format qu'une adresse locale, étant donné que la mémoire est commune. Les champs *split-OBL*, processeur et processus sont nécessaires pour convertir les variables non-locales d'un environnement en objets non-locaux.

La figure 5.8 montre un exemple d'utilisation du type "référence non-locale" par un registre pointeur d'environnement (non local) **E**.

Registres arguments Les registres arguments sont chargés avec des variables des piles locales et globales. Ils peuvent donc avoir les mêmes types que les éléments de piles et occupent donc également deux mots mémoire de 32 bits.

Implémentation des *hash-windows*

Une *hash-window* est créée et empilée sur la pile globale par le noyau, à la création de chaque processus. Un élément de *hash-window* comprend deux éléments de pile. Lorsqu'il est utilisé, le premier élément de pile est constitué par la référence non-locale liée et le second par l'objet auquel elle est liée (voir figure 6.5).

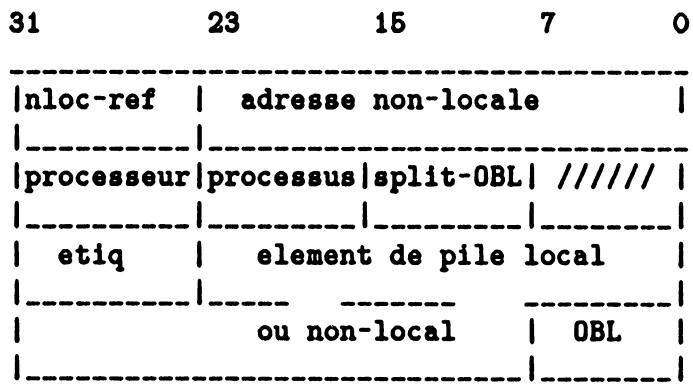

Figure 6.5: Format d'un élément de *hash-window*

Les liaisons dans les *hash-windows* sont étiquetées par la valeur courante de l'*OBL*, comme les liaisons locales.

Les liaisons dans les *hash-windows* sont toutes enregistrées dans la pile de restauration car elles concernent toujours des variables logiques antérieures à la création du processus. En cas de retour arrière, les liaisons concernées sont réinitialisées de façon transparente, en même temps que les liaisons locales de la pile. L'utilisation du mécanisme de restauration standard implique donc que le premier mot d'une entrée libre est étiqueté variable libre. A la création d'une *hash-window*, le noyau initialise donc l'ensemble de ses entrées de cette façon.

La taille des *hash-windows* est un multiple de deux. L'algorithme de dispersion (*hashing*) calcule le reste de la division de la partie significative de l'adresse non-locale par la taille de la *hash-window*. Différents algorithmes ont été testés pour le traitement des collisions. Le plus simple consiste à incrémenter de un l'index calculé. L'algorithme utilisé actuellement incrémente cet index d'un nombre premier proche du tiers de la taille de la *hash-window*, afin de provoquer une meilleure dispersion. Aucune différence significative de performances n'a pu être mesurée lors du changement d'algorithme en raison probablement du faible taux d'utilisation des *hash-windows* (voir section 8.3.2).

Une seule *hash-window* est allouée par processus et elle est de taille fixe. La taille des *hash-windows* est actuellement de 128 entrées, ce qui correspond à 2 Koctets de mémoire. Comme nous le verrons ultérieurement (section 8.3.2), peu de programmes utilisent une portion significative de leur *hash-window*. Dans les versions ultérieures du système, il serait donc souhaitable de traiter le débordement de capacité en permettant à un processus, en cas de débordement de capacité d'une *hash-window*, d'en créer une nouvelle et de la lier à la *hash-window* courante. Tout en éliminant le risque d'erreur du système pour débordement de capacité, cette technique permettrait de réduire notablement la taille des *hash-windows* utilisées.

Structure des *branch-points*

Les *branch-points* jouent plusieurs rôles dans l'implémentation. Vus sous l'angle du processus exécutant la machine abstraite PEPSys, ce sont essentiellement des points de choix partagés. Vus depuis l'ordonnanceur, ils sont des nœuds permettant d'évoluer dans l'arbre des processus, - lorsqu'ils ont été à l'origine de la création de processus -, ainsi que des sources de travail potentiel. Ces divers rôles compliquent la structure des *branch-points* (voir figure 5.2) puisqu'en plus des informations contenues habituellement dans les points de choix, ils doivent contenir les informations permettant d'offrir un travail et de créer un processus à partir du travail offert, ainsi que contenir les informations de chaînage dans l'arbre des processus. Pour des raisons de simplicité, un *branch-point* contient une entrée par processus fils potentiel, ce qui augmente l'espace utilisé pour les programmes ayant des prédicats parallèles comportant un grand nombre de clauses.

6.4.2 Emulateur

L'émulateur est codé très classiquement par une boucle d'interprétation dont les entrées sont indexées par les codes opérations des instructions de la machine abstraite qui occupent chacun un mot mémoire (*threaded code*). Le pointeur d'instructions est déclaré comme un registre de la machine (facilité du langage C) ce qui permet un branchement très rapide au traitement d'une instruction. Pour des raisons d'encombrement mémoire et de mise au point, certaines instructions complexes et peu utilisées sont réalisées par des appels de procédure. C'est également le cas de l'unification et du déréférencement non-local.

6.4.3 Déréferencement

L'efficacité de l'opération de déréférencement est cruciale dans les systèmes Prolog. Le déréférencement classique de Prolog, appelé en PEPSys déréférencement local, est implémenté classiquement par une macro définition qui est donc étendue par l'émulateur. Le déréférencement non-local, - accès à la pile d'un autre processeur, vérification de la validité de la liaison, recherche dans les *hash-windows*, changement éventuel de la présentation de l'objet déréférencé si la liaison est valide -, est trop complexe pour être étendu en ligne dans la boucle d'émulation. Il a donc été divisé en deux procédures récursives dont l'une est spécialisée dans la recherche dans les *hash-windows*.

Des efforts importants ont été faits pour améliorer l'efficacité du déréférencement. Le choix des étiquettes (*tag*) des éléments de pile a été fait pour optimiser les tests faits dans le déréférencement local. De même, contrairement au déréférencement Prolog classique qui manipule directement les objets accédés, le déréférencement PEPSys manipule autant que possible des pointeurs sur les éléments de piles. Le gain d'efficacité qui en résulte, - manipulation de mots au lieu de doubles mots -, est de près de 20 %, au détriment de la simplicité. Une autre

amélioration très sensible du temps d'exécution en parallèle des programmes pratiquant de nombreuses recherches dans les *hash-windows* a été obtenue en réduisant la longueur des chaînes de références. Les liaisons valides trouvées dans les *hash-windows* non-locales sont cachées dans les *hash-windows* locales. L'importance de cette optimisation est évaluée dans la section 8.3.2.

6.4.4 Unification

Pour faciliter la mise au point, l'unification a été implémentée comme une procédure récursive. Transformer en une procédure itérative gérant explicitement une pile ne poserait aucun problème et devrait améliorer l'efficacité de l'implémentation de 10 % environ. La taille de cette procédure interdit par contre de l'inclure dans la boucle d'émulation par macro-expansion.

6.4.5 Instructions de contrôle

L'implémentation des instructions de la machine abstraite permettant le contrôle du parallélisme suit étroitement la définition qui en a été donnée. Toutefois lorsqu'un processus OU-parallèle se termine et que le *branch-point* dont il est issu a encore du travail à offrir, le processus qui se termine prend du travail sans se réinitialiser. Si ses registres arguments n'ont pas changé de valeur au cours de l'exécution qui s'achève, - dont la durée de vie a donc été très courte -, il n'est pas non plus nécessaire de les recharger[3]. Cette optimisation n'est intéressante que pour les programmes contenant des prédicats OU-parallèles comportant plus de deux alternatives. Bien que ce soit le cas de certains des programmes de test utilisés pour les mesures - programmes *TInAs* et *Chat80* (voir section 7.4) -, l'influence de cette optimisation n'a pas été quantifiée.

6.4.6 Prédicats une-solution

Les prédicats une-solution, c'est-à-dire comportant la déclaration de propriété *solutions(one)*, sont implémentés suivant la technique du *backbranching* décrite à la section 5.2.5. Lorsque le *backbranching* d'un processus devenu inutile est bloqué par un *branch-point* actif, le processus bloqué reste en attente active. Cette solution s'est révélée meilleure que celle consistant à aider le ou les processus actifs à terminer. Cette dernière stratégie a été testée par le système de simulation et provoque le développement de longues branches de calcul entrelacées, - les processus qui devraient se "suicider" s'entraidant mutuellement -, au lieu de hâter la terminaison des branches mortes.

[3] Il s'agit ici d'une optimisation voisine de celle qui est pratiquée sous le nom de retour-arrière superficiel (*shallow backtracking*) dans les systèmes Prolog séquentiels.

6.4.7 Implémentation du parallélisme ET déterministe

Le système parallèle PEPSys comporte une implémentation simplifiée du parallélisme ET déterministe car faite après l'implémentation du parallélisme OU de façon trop étroitement calquée sur celle-ci.

Les mêmes mécanismes d'initialisation de processus, de leurs registres arguments, et d'accès aux environnements non-locaux ont été utilisés. La conséquence principale est qu'il devient nécessaire de créer des *hash-windows* dans les processus résolvant les branches droites, ce qui est inutile dans le modèle PEPSys. Eviter ce surcoût inutile aurait nécessité une modification de la gestion des environnements non-locaux dans l'émulateur, afin de distinguer les variables non-locales du parallélisme ET, des variables non-locales du parallélisme OU. Dans le premier cas, la validité des liaisons n'a pas besoin d'être testée par les processus fils et il est possible à un processus fils de lier des variables dans la pile de son père. Dans le second cas au contraire les liaisons faites par le père doivent être testées et en présence de liaison non valide ou en cas d'absence de liaison, la variable non-locale ne peut qu'être liée dans une *hash-window*. La modification de l'émulateur permettant d'optimiser le parallélisme ET n'a pu être menée à bien faute de temps.

La principale structure de données utilisée par le parallélisme ET, le *fork-point*, peut être vue comme un sous-ensemble du *branch-point* puisqu'elle ne comporte pas de sauvegarde des registres arguments et ne peut donner naissance à plus d'un processus.

6.5 Ordonnancement

Dans PEPSys comme dans tous les systèmes multi-séquentiels, la recherche de travail est à la charge des processeurs oisifs. Comme nous l'avons signalé, un noyau d'ordonnancement s'exécute sur chacun des processeurs, dans le but de le maintenir actif, c'est-à-dire en cours d'exécution d'une portion du programme utilisateur. Le noyau d'ordonnancement utilise l'arbre des *rootframes* et des *workframes* (*branch-points ou fork-points*) pour trouver une portion de programme à exécuter. Dans toutes les stratégies d'ordonnancement décrites ci-après, le parcours de cet arbre se fait sans exclusion mutuelle. Celle-ci n'est prise que lorsqu'un travail est effectivement trouvé, pour l'acquérir effectivement. De nombreuses précautions doivent être prises pour éviter que les modifications de la structure de l'arbre des processus, se produisant durant son parcours par un processeur oisif, ne provoquent une erreur dans la recherche de travail. Le coût de ces précautions a été estimé inférieur à celui de l'utilisation systématique de verrous d'exclusion mutuelle.

Pour limiter le nombre de verrous utilisés, un seul verrou est associé à l'ensemble du travail offert par un processus. Ce verrou est alloué dans la *rootframe* du processus et doit être pris avant toute opération modifiant une *workframe* du processus. Dans le système parallèle, les *rootframes* contiennent en outre des pointeurs sur la chaîne des *workframes* du processus. Si le processus n'a pas de travail

à offrir, ses *workframes* "actives"[4] permettent de descendre le sous-arbre de ses processus fils.

Initialement, un processeur est oisif. Un processeur actif redevient oisif lorsque :

- le processus qu'il exécute se termine, par retour arrière à sa racine ainsi que nous l'avons expliqué au paragraphe 5.2.2.
- le processus qu'il exécute se suspend. Dans le parallélisme OU, cette situation peut se produire lors d'un retour-arrière, si la poursuite du retour-arrière par le processus courant est susceptible de libérer des portions de piles utilisées par ses fils. Dans le parallélisme ET déterministe, c'est le cas pour les processus exécutant la partie gauche lorsqu'ils ont trouvé une solution. En cas de suspension du processus courant, le modèle PEPSys [Hailperin 86] prévoit de rechercher du travail dans les fils du processus uniquement, afin d'éviter des trous noirs sur la pile. Cette stratégie est appelée "aider ou attendre" (*wait or help*).

6.5.1 Ordonnanceur originel

La recherche de travail se fait en parcourant l'arbre des *rootframes* et *workframes* en profondeur d'abord, en partant du processus créé au début de l'exécution du programme. De plus, cet ordonnanceur implémente la stratégie d'aide des processus fils en cas de suspension d'un processus. Le parcours du sous-arbre des processus fils se fait alors également en profondeur d'abord. Le premier travail possible est utilisé pour créer un nouveau processus. La principale raison d'être de la recherche de travail en profondeur d'abord est la simplicité. Ce type de parcours d'arbre se code facilement par quelques procédures récursives.

Avec cet ordonnanceur, les performances des programmes OU-parallèles dépendent de l'ordre des clauses dans les prédicats "parallèles". Etant donné que la recherche de travail se fait d'abord dans le premier processus créé, il est beaucoup plus efficace de renverser l'ordre "naturel" des clauses du programme. Pour un prédicat parallèle composé de deux clauses, l'une récursive et l'autre terminale, il est préférable que le processus créant le *branch-point* exécute la clause récursive et "offre" la clause terminale aux processeurs oisifs. Un example de ce type est le prédicat *index* du programme des n-reines de la figure 4.1. L'ordre naturel des clauses est celui de la figure 4.1 :

```
-properties([solutions(all), clauses(unordered), execution(eager)]).
index( Size, Size ).
index( N, Size ) :- S1 is Size - 1, S1 > 0, index( N, S1 ).
```

Cependant le programme est beaucoup plus efficace en parallèle (gain de temps proche de 20 % sur huit processeurs) si l'ordre des clauses est renversé :

[4] Ayant donné naissance à au moins un processus encore en vie.

```
-properties([solutions(all), clauses(unordered), execution(eager)]).
index( N, Size ) :- S1 is Size - 1, S1 > 0, index( N, S1 ).
index( Size, Size ).
```

6.5.2 Autres stratégies d'ordonnancement

En plus du problème de programmation évoqué ci-dessus, l'ordonnanceur originel parcourt l'arbre de réfutation dans un ordre qui ne semble pas optimal. En effet, dans la majorité des programmes, les processus à plus forte granularité sont créés à partir des *workframes* les plus hautes de l'arbre. Il semble donc souhaitable de rechercher du travail en largeur d'abord. Cependant une stratégie de ce type n'est pas simple à réaliser car elle suppose de maintenir efficacement de nombreuses informations relatives à la structure de l'arbre de réfutation[Brand 88a]. Un autre défaut de l'ordonnanceur originel est de se limiter au sous-arbre des fils, en cas de suspension d'un processus, afin de limiter la consommation mémoire en évitant les trous noirs. Cette stratégie peut conduire à laisser des processeurs oisifs, alors qu'il y a du travail dans des processus autres que les fils du dernier processus suspendu.

Sans tenter de réaliser un véritable ordonnanceur en largeur d'abord, plusieurs expériences ont été menées. Pour ces expériences, la contrainte d'éviter la création de trous noirs dans les piles a été relachée, au prix d'un surcoût en mémoire qui sera évalué ultérieurement. Cette liberté supplémentaire a été mise à profit pour :

- rechercher un travail dans l'ensemble de l'arbre, lorsqu'un processus est suspendu et qu'aucun de ses fils n'offre du travail. Dans ce cas, la recherche se fait en profondeur d'abord, à partir de la racine de l'arbre. Le résultat de cette expérience a été négatif, les performances étant inférieures aux performances de l'ordonnanceur originel.

- prendre du travail au processus qui en a le plus à offrir. Cette stratégie avait déjà été expérimentée sur le simulateur, sous le nom de "Robin des Bois" ("voler aux riches pour donner aux pauvres..."[Baron 89] [Scott 1819]). Un compteur de travail disponible est maintenu pour chaque processus. L'assignation du travail initial à un processeur n'ayant pas encore créé de processus, se fait depuis le processus ayant le compteur le plus élevé. Pour la recherche de travail consécutive à la suspension d'un processus, deux stratégies ont été testées :

  - en profondeur d'abord dans le sous-arbre des processus fils. Les résultats sont alors comparables à ceux de l'ordonnanceur originel.
  - en profondeur d'abord dans le sous-arbre des processus fils, suivie si l'on n'a pas trouvé de travail, d'une recherche de type Robin des Bois. Les résultats obtenus sont meilleurs que ceux de l'ordonnanceur originel, confirmant les bons résultats obtenus sur le simulateur pour des architectures mono-grappe. Sur l'ensemble des programmes les performances sont accrues de 5 à 10 %. En particulier, les performances dépendent

moins de l'ordre des clauses. Ainsi, le programme des n-reines déjà cité n'est que 5 % plus lent en utilisant l'ordre "naturel" des clauses du prédicat parallèle que si cet ordre est inversé. Avec l'ordonnanceur originel, la différence d'efficacité est de 20 %. Avec un ordonnanceur "optimal", les résultats devraient être indépendants de l'ordre des clauses. Les gains de performances dus à cet ordonnanceur sont particulièrement significatifs pour les programmes ET-parallèles déterministes. Un plus grand nombre de processus sont créés que si l'on utilise la stratégie *d'aider ou attendre*, ce qui réduit l'oisiveté des processeurs et accroît les performances. L'ordonnanceur "Robin des Bois" étant celui qui fournit les meilleures performances, c'est lui qui a été utilisé pour les mesures de performances des chapitres qui suivent.

- expérimenter une forme de recherche de travail en largeur d'abord pour le parallélisme OU. L'idée de cette stratégie est d'utiliser la valeur du compteur *OBL* pour représenter la profondeur dans l'arbre d'évaluation. A chaque *branch-point* offrant du travail est associé ce compteur, qui mesure le nombre de *branch-points* créés sur la branche de l'arbre qui le relie à la racine. Pour obtenir une estimation de la profondeur des *branch-points* dans l'ensemble de l'arbre de réfutation, on initialise l'*OBL* d'un nouveau processus à la profondeur de la branche qui le relie à la racine en lui donnant la valeur de celui du processus père, lors de la création du *branch-point* dont le processus est issu[5]. Le noyau d'ordonnancement d'un processeur oisif sélectionne le travail le plus "haut" possible dans l'arbre, c'est-à-dire provenant du *branch-point* ayant le plus faible *OBL*.

  Les résultats obtenus en utilisant cette stratégie se sont montrés décevants. D'une part les performances ne sont pas meilleures que celles obtenues avec l'ordonnanceur originel et d'autre part elles dépendent tout autant de l'ordre des clauses des prédicats parallèles. Il n'a pas été possible, faute de temps, d'analyser l'origine de ces résultats médiocres : principe erroné oumise en œuvre maladroite? Un autre problème avec cette approche est qu'elle n'est pas généralisable au parallélisme ET pur car il n'utilise pas les *OBL*.

On peut résumer les expériences faites pour l'ordonnancement en remarquant que pour un nombre limité de processeurs, comme c'est le cas avec le MX500, les performances ne semblent pas dépendre de façon significative de l'ordonnanceur. Cependant, comparé à l'ordonnanceur originel, l'ordonnanceur "Robin des Bois" retarde l'infléchissement des performances lorsque le nombre de processeurs augmente - ce résultat a principalement été mis en évidence par le simulateur -, et donne des résultats plus indépendants de l'ordre des clauses dans les prédicats parallèles. Un phénomène semble toutefois se manifester lors de l'utilisation de cet ordonnanceur. Lorsque la machine est chargée, il semble que les résultats deviennent plus imprévisibles. Aucune étude systématique n'a été entreprise pour

[5]La valeur initiale de l'*OBL* d'un processus est habituellement nulle.

tenter de préciser et de quantifier ce phénomène, mais on pourrait l'expliquer en remarquant que l'identité du processus ayant le plus de travail à offrir a plus de chances de changer, dans un système chargé, entre le moment où un processeur inactif décide de lui prendre du travail et celui où il est effectivement en mesure de le faire (c'est le cas si ce "processeur" est suspendu par le système d'exploitation entre temps). Ce phénomène devra être pris en considération si on réalise un système parallèle destiné à un environnement de multiprogrammation.

6.6 Mise au point

La mise au point d'un système parallèle est une tâche très difficile en raison du caractère fugitif des erreurs qui s'y produisent. Fort heureusement, il n'a pas été nécessaire de déboguer les programmes utilisateurs écrits en PEPSys. Ceux-ci ayant été mis au point séquentiellement, en utilisant le simulateur ou la transformation en Prolog séquentiel [Ing 87b], aucune erreur nouvelle n'est apparue lors de l'exécution parallèle. La mise au point du système parallèle a par contre nécessité trois outils différents :

- le debogueur symbolique *pdbx* du système Dynix.
- un outil de mise au point au niveau des instructions de la machine abstraite, développé en même temps que l'implémentation.
- une implémentation de l'*Instant Replay* [LeBlanc 87,Westphal 87] rendant possible la réexécution déterministe de programmes parallèles. Cet outil est complémentaire des deux précédents.

6.6.1 pdbx

pdbx est la version parallèle du debogueur symbolique *dbx*. Pour un ensemble d'un des processus UNIX s'exécutant en parallèle, *pdbx* offre la possibilité de mettre des points d'arrêt pour tous les processus ou une partie d'entre eux, et d'inspecter les variables de chacun des processus. L'exécution d'un des processus UNIX du système parallèle en pas à pas est également possible.

6.6.2 Debogueur PEPSys

Un outil de mise au point adapté à la machine abstraite PEPSys a été réalisé. Cet outil utilise un registre d'interruption[6]. Il permet d'arrêter l'exécution au niveau de chaque instruction de la machine abstraite. Il est également possible d'exécuter en pas à pas les instructions de la machine abstraite. L'arrêt ou la trace peuvent

[6] L'implémentation utilise un système d'interruptions pour la mise au point et le traitement d'erreur. Il serait aussi utilisé par une implémentation de la terminaison brutale des processus inutiles pour les prédicats "une-solution" et la combinaison des parallélismes OU et ET.

en outre se faire au niveau des prédicats, à l'exécution des instructions *call* et *retry, par_retry*, correspondant à deux des quatre ports des debogueurs Prolog traditionnels. Le debogueur PEPSys connait également les structures de données PEPSys, - points de choix, *branch-points, hash-windows*, etc -, qu'il montre sous forme lisible. C'est aussi le cas des variables (par exemple une liste est écrite [a,b]), des registres et des instructions de la machine abstraite. Il est aussi possible de suivre des chaînes de pointeurs en utilisant des désignations symboliques. L'arbre des processus peut également être tracé graphiquement, mettant en évidence son évolution.

En cas d'erreur détectée par le système, le metteur au point est appelé. Il stoppe les autres processeurs, ce qui permet d'inspecter l'état de l'ensemble du système.

6.6.3 Instant Replay

Principe de l'*Instant Replay*

Un système parallèle étant non-déterministe, il est difficile de reproduire deux exécutions identiques. En conséquence, des erreurs intermittentes sont susceptibles d'apparaitre. De plus, l'utilisation de techniques de trace ou d'outils de mise au point, *pdbx* ou le debogueur PEPSys dans notre cas, sont de nature à faire disparaitre l'erreur ou à en modifier la manifestation. L'*Instant Replay* est un mécanisme permettant de réexécuter de façon déterministe un programme parallèle. Le principe est d'enregistrer, durant une première exécution du programme parallèle, le minimum d'informations permettant de réexécuter le programme de façon déterministe.

L'idée de base de l'*Instant Replay* est qu'il suffit d'enregistrer l'ordre des évènements significatifs, à savoir les communications inter-processus, plutôt que les évènements eux-mêmes, à savoir les données échangées. Dans le cas d'un multiprocesseur à mémoire commune, la communication entre processus se produit à chaque accès à des données partagées. A chaque donnée partagée est associé un **compteur de version**, et à chaque processeur un **historique**. En mode **enregistrement**, lorsqu'un processeur accède une donnée partagée, la valeur du compteur de version est enregistrée sur l'historique du processeur. En mode **réexécution**, l'historique est redéroulé ; à chaque accès à une donnée partagée, le processeur attend que la valeur du compteur associé à la donnée soit la même que la valeur courante de l'historique. Lorsque c'est le cas, il accède la donnée et progresse d'un élément dans la lecture de l'historique. Lorsque l'accès à la donnée partagée a pour effet de la modifier, en mode enregistrement ou réexécution, la valeur du compteur qui lui est associé est incrémentée.

Implémentation de l'*Instant Replay* dans PEPSys

Une implémentation simplifiée de l'*Instant Replay* a été faite dans PEPSys. L'accès aux variables logiques dans les piles locales et globales n'a pas besoin d'être enreg-

istré. Il s'agit de l'accès fait par un processus aux variables d'un de ses ancêtres ; ces variables ne sont liées validement pour ce processus que si leur liaison est antérieure à la création du *branch-point* dont est issu le processus. Dans le cas contraire, la valeur lue ne sera pas retenue. Le test de validité utilise la valeur de l'*OBL* de la variable, conformément au modèle de calcul. Par contre, il est nécessaire d'enregistrer la modification des structures de données utilisées par l'ordonnanceur telles que les *branch-points* et *fork-points*. Pour ce faire, les verrous de synchronisation associés à ces structures de données sont doublés d'un compteur qui est incrémenté et enregistré sur l'historique à l'enregistrement. Lors de la réexécution, la prise du verrou est retardée tant que la valeur de l'historique ne correspond pas à celle du compteur associé au verrou.

Un problème particulier provient de ce que le parcours de l'arbre des processus par l'ordonnanceur à la recherche de travail se fasse sans exclusion mutuelle. L'accès aux données communes n'étant pas enregistré à cette occasion, un mécanisme plus complexe a dû être mis en œuvre pour enregistrer le type de l'opération effectuée ainsi que l'adresse des structures de données qui fournissent du travail (voir [Chassin et al 88]).

L'implémentation simplifiée de l'*Instant Replay* faite dans PEPSys est efficace : la déperdition de performances durant la phase d'enregistrement est limitée à 2 % ou 3 %. La perturbation introduite est suffisamment minime pour ne pas faire disparaitre les erreurs se produisant durant l'exécution. Il est possible de capturer les erreurs intermittentes en répétant les enregistrements du programme où elle apparait. Une fois l'exécution erronée enregistrée, on peut identifier les erreurs en faisant réexécuter le programme enregistré tout en utilisant les autres techniques de mise au point : *pdbx*, debogueur PEPSys, traces, etc. La réexécution est environ deux à trois fois plus lente que l'exécution normale sans que celà ne pose aucun problème.

L'*Instant Replay* a surtout été utilisé pour mettre au point l'ordonnanceur. En effet, sa correction suppose celle de la gestion des variables par l'émulateur. Une fois au point, il s'est montré extrèmement utile pour la mise au point de toutes les évolutions ultérieures du système décrites dans la section 6.8. Par contre, l'*Instant Replay* n'a pas été utilisé pour les mesures qui font l'objet des derniers chapitres de cette thèse. Ce choix sera justifié ultérieurement.

6.6.4 Mise au point systématique

Par mise au point systématique on entend la définition de programmes de tests simples, écrits en langage PEPSys, et qui permettent de tester isolément les composants critiques du système. Le programme de la section 7.1.2 constitue un bon exemple de test des fonctions de déréférencement non-local et de recherche dans les *hash-windows*. Il est à noter que le programme est écrit de façon à forcer les processeurs à exécuter les modules testés. S'ils comportent une erreur, celle-ci apparaitra nécessairement à chaque exécution, contrairement aux erreurs fugitives qui se produisent dans les programmes PEPSys "réels". La mise au point systématique

a été insuffisament utilisée pour le système PEPSys. Des programmes analogues à celui de la section 7.1.2 auraient permis de mettre au point le déréférencement PEPSys de façon plus facile que directement sur des programmes de test et sans pouvoir utiliser l'*Instant replay* comme cela a été souvent le cas malheureusement...

6.7 Prédicats prédéfinis

Comme tout système Prolog, le système parallèle PEPSys offre un certain nombre de prédicats prédéfinis. Le nombre de ces prédicats a été réduit au minimum pour des raisons de simplicité. Ainsi n'est réalisée que l'arithmétique entière sur entiers courts. Certains prédicats prédéfinis posent des problèmes particuliers en parallèle. Nous présenterons trois d'entre eux :*not*, *oneof* et *bagof*, implémentés directement en instructions de la machine abstraite PEPSys. Avec les autres prédicats prédéfinis similaires, ils sont liés aux programmes utilisateurs à l'édition de liens.

6.7.1 Implémentation du *not*

L'implémentation faite est celle utilisée habituellement en Prolog et connue sous le nom de "négation par l'échec" (*negation as failure*). Comme cette définition utilise un *cut*, non défini en PEPSys, deux prédicats ont été définis : *not_seq* et *not_par*. Le premier est la négation par l'échec classique, et est réservé aux prédicats connus pour être séquentiels[7]. Le second prédicat *not_par* est réservé à la négation de prédicats potentiellement parallèles. *not_seq* peut être défini en Prolog de la façon suivante :

```
not_seq(X) :- call(X), !, fail.
not_seq(X).
```

Le prédicat *not_par* est mis en échec par la première solution produite au prédicat nié, qui est donc appelé avec la propriété "une-solution". Il est définissable en PEPSys en utilisant des techniques de programmation analogues à celles données pour la conversion du *cut* dans le paragraphe 4.3.2 :

```
-properties([]).
not_par(X) :- not_test(X, Test_value), not_result(Test_value).

-properties([solutions( one ), clauses( ordered )]).
not_test(X, not_fail) :- call(X).
not_test(_, not_success).

-properties([]).
not_result(not_success).
```

[7]La définition de séquentialité a déjà été donnée à propos des prédicats "une-solution". Un prédicat est séquentiel si ni lui ni l'un des prédicats qu'il appelle n'est générateur de parallélisme.

Le compilateur PEPSys analyse la séquentialité de tout prédicat nié dans un module parallèle. Si celle-ci peut être établie, un *not_seq* est substitué au *not* originel. Dans le cas contraire, le compilateur substitue un *not_par*.

6.7.2 Implémentation du *oneof*

Le prédicat prédéfini *oneof* fournit la première solution (en temps) à un prédicat. Il peut être défini ainsi en PEPSys :

```
-properties([solutions( one )]).
oneof(X) :- call(X).
```

Un autre prédicat, *toponeof*, non défini dans le langage, a été introduit pour exploiter efficacement deux cas particuliers de *oneof* : il s'agit de l'appel à un prédicat d'un module parallèle depuis un module séquentiel, en utilisant le prédicat prédéfini *oneof*, ou encore de l'utilisation de ce prédicat prédéfini au plus haut niveau de la partie parallèle d'un programme, c'est-à-dire avant la création de processus parallèles. L'utilisation du prédicat dans le second cas est laissée à la responsabilité du programmeur. L'intérêt du *toponeof* est son efficacité : il suffit d'interrompre et de tuer tous les processus parallèles en cours d'exécution. Le *toponeof* est implémenté en instructions de la machine abstraite PEPSys, en changeant dans le code du *oneof* l'instruction *sync_oneof* par l'instruction *sync_toponeof* qui a pour fonction de tuer les autres branches du calcul.

6.7.3 Bagof

La syntaxe du *bagof* est :
bagof(X, p(X, Paramètres), Liste)

Sa sémantique est : *Liste* est la liste de toutes les valeurs de la variable X telle que *p(X, Paramètres)* est vraie. Lors du retour-arrière sur le prédicat *bagof*, la variable *Liste* est réinitialisée à libre. La sémantique du *bagof* de PEPSys est équivalente à celle du *findall* de la plupart des autres systèmes Prolog.

L'implémentation du *bagof* est un autre exemple d'extension des deux mécanismes de contrôle classiques de la WAM, la continuation et le retour-arrière. Elle utilise une structure de données sur la pile locale, appelée *bag-frame* pour stocker les paramètres utilisés par les processeurs qui coopèrent à son accomplissement ainsi qu'une zone non structurée de la pile globale servant à stocker les solutions. Le *bagof* est codé par trois instructions de la machine abstraite :

```
bagof/3:
        bag-init
        metacall X2
        extract-result
```

L'implémentation du *bagof* utilise en outre l'instruction *close-bag*.

Le *bag-init* crée et initialise la structure *bag-frame* sur la pile locale, y fait pointer le registre pointeur d'environnement **E**, ainsi que le registre de retour-arrière **B**, alloue un espace de taille fixe sur la pile globale pour les solutions[8] et enfin positionne le registre instruction de retour arrière **BP** à *close-bag*. L'appel au prédicat *p* est effectué par l'instruction *metacall*.

L'instruction *extract-result* est exécutée par chacun des processus concurrant à la solution de *p*. Ils trouvent dans la *bag-frame*, accédée grâce au registre **E**, les paramètres permettant de rajouter la solution qu'ils ont produite à la liste de solutions, construite dans la zone de taille fixe de la pile globale. Le registre **E** est local si la solution est produite par le processus ayant démarré l'exécution du *bagof*, non-local dans le cas contraire. L'instruction se termine en provoquant un retour-arrière destiné à calculer de nouvelles solutions.

Le *close-bag* est l'instruction que le processus ayant démarré le *bagof* exécute lorsque toutes les solutions ont été produites. Etant exécutée par retour-arrière elle accède ses arguments, dans la *bag-frame*, à travers le registre **B**. Elle clôt la liste de solutions et compacte la pile globale.

Le *bagof* de PEPSys est très efficace comparé à celui de nombreux systèmes Prolog. Il devrait toutefois être complété pour lever la limitation de la taille pour l'espace alloué à la liste de solutions.

6.8 Extensions

Les sections qui précèdent décrivent l'implémentation du module parallèle de PEPSys. Des extensions à cette version de base ont été faites, d'une part afin de donner aux utilisateurs l'ensemble des fonctionnalités des modules séquentiels en connectant l'implémentation parallèle à un système Prolog séquentiel, d'autre part en adjoignant au système une partie des fonctionnalités du système de programmation logique avec contraintes, CHIP.

6.8.1 Connection à un système Prolog séquentiel

Pour implémenter la modularité PEPSys, et en particulier offrir aux programmeurs l'ensemble des fonctionnalités de Prolog dans les modules séquentiels, deux solutions ont été envisagées. La première consistait à compléter l'implémentation parallèle en ajoutant les prédicats prédéfinis habituellement disponibles en Prolog. La seconde, retenue parce que moins coûteuse en temps de développement, consiste à connecter l'implémentation parallèle à un système Prolog séquentiel [Rapp 88]. Le système Prolog choisi est le système SEPIA [Meier 89], développé simultanément à l'implémentation du module parallèle de PEPSys.

Plutôt que de lier SEPIA et PEPSys pour faire un même système, il a été décidé de les conserver indépendants, en les reliant par des *pipes* UNIX. L'intérêt

[8] Cet espace occupe 40 Koctets dans la version actuelle du système

de ce type de connexion est d'éviter la multiplication des données et du code source de SEPIA dans les processus (UNIX) de PEPSys et de conserver l'indépendance entre les deux systèmes, qui étaient alors tous deux en phase de mise au point. Un certain nombre de prédicats prédéfinis tels que *parbag* et *parone* ont été ajoutés à SEPIA pour appeller PEPSys. Des procédures de conversion de données entre les deux systèmes ont également été mises en place de part et d'autre.

La connection de SEPIA à PEPSys a permis d'exécuter des programmes PEPSys complets [Rapp 88]. L'expérience accumulée sera très utile à la réalisation d'un autre système parallèle, même si les performances se sont souvent révélées décevantes. La lenteur de la connection par *pipes* n'est pas la cause de ces mauvaises performances puisque le temps passé dans le module de connection reste très limité comparé au temps passé dans le programme. L'inefficacité tient plutôt à l'inadaptation des programmes testés à ce type de connection. Ainsi, le module séquentiel originel du programme *TInA*, faisait des assertions de prédicats utilisés ultérieurement par le module parallèle. Ne pouvant depuis SEPIA compiler dynamiquement des prédicats du module parallèle, il a fallu réécrire une partie de chacun des modules. Les prédicats rajoutés par assertion dans la version initiale du module séquentiel sont maintenant stockés dans des listes, et un méta-interpréteur a été adjoint au module parallèle pour exploiter ces listes.

Le résultat des expériences menées indique qu'il est largement souhaitable, pour des raisons de performances, d'intégrer dans un même système les modules parallèles et séquentiels pour n'avoir aucune conversion à faire pour passer de l'un à l'autre. Pour ce faire deux possiblités : soit partir d'un système Prolog existant, - ce qui n'a pas été fait ici parce que SEPIA était en cours de développement -, pour réaliser le système parallèle, soit réaliser autour du système parallèle l'ensemble des fonctionnalités de Prolog, - ce qui n'a pas été fait ici par manque de temps -. Dans chacune de ces solutions, c'est au compilateur qu'il incomberait de vérifier que la syntaxe du langage PEPSys est bien respectée et qu'aucun effet de bord n'est utilisé dans les modules parallèles.

6.8.2 Implémentation de CHIP

Le système de programmation logique avec contraintes CHIP [Dincbas et al. 88] permet à la programmation logique de résoudre de façon élégante et efficace un grand nombre de problèmes combinatoires. Ces problèmes sont le plus souvent d'une grande complexité et il semble naturel de chercher à les faire bénéficier des gains d'efficacité que procure le parallélisme. Pour ce faire le système parallèle PEPSys[9] a été étendu pour supporter un sous-ensemble des possibilités de CHIP [Van Hentenryck 89].

Le système parallèle PEPSys constitue la base utilisée pour combiner le parallélisme OU avec les techniques de maintien de cohérence sur des domaines finis. L'extension de PEPSys a nécessité l'ajout de plusieurs nouvelles instructions à

[9]Les programmes qui ont été testés n'utilisaient aucun des prédicats prédéfinis qui auraient nécessité la séparation en modules séquentiels et parallèles

la machine abstraite PEPSys et l'introduction d'une nouvelle zone de données pour contenir les valeurs des domaines des variables s'étendant sur des domaines finis. Comme dans le cas des variables logiques, les liaisons des variables domaines d'un processus sont transmises à ses fils. Contrairement au cas des variables logiques, les variables domaines peuvent ensuite être modifiées par les processus fils en propageant des contraintes qui en réduisent les domaines. La zone domaine est gérée de façon analogue au tableau de liaisons du modèle SRI (voir paragraphe 3.4.4). Lorsqu'un processus est créé, la zone domaine du processeur sur lequel il s'exécute est mise à jour pour avoir le même contenu que celle du processus père. Cependant, à la différence du modèle SRI, tous les accès ultérieurs aux variables de la zone domaine demeurent purement locaux et ne nécessitent donc nullement un multiprocesseur à espace d'adressage global.

L'ordonnanceur originel a été utilisé pour cette implémentation. En effet la stratégie de recherche de travail, "aider ou attendre" permet de mettre à jour la zone domaine de façon simple lorsqu'un processeur prend du travail à l'un des fils de son dernier processus suspendu. Par contre, la stratégie de recherche de travail en profondeur d'abord ne permet pas de partage optimal du travail entre les processeurs. En dépit de ces problèmes, les résultats préliminaires montrent des gains de performances importants, parfois superlinéaires, pour des problèmes réels.

6.9 Résumé du chapitre

Afin d'obtenir des résultats réels et d'exécuter des programmes de grande taille, une implémentation sur multi-processeur a été réalisée. Pour des raisons d'efficacité et de simplicité de mise en œuvre, un multiprocesseur à mémoire commune a été choisi. L'implémentation du système parallèle est basée sur la machine abstraite PEPSys. L'ordonnanceur qui la complémente utilise des stratégies simples. Diverses techniques, dont une implémentation du mécanisme d'*Instant Replay* ont été utilisées pour la mise au point parallèlle. La connection du système parallèle à un système Prolog séquentiel (SEPIA) permet d'exécuter des programmes PEPSys complets, sans obtenir toutefois les performances que donnerait un système complètement intégré. L'extension du système parallèle pour la programmation avec contraintes procure des gains de performances importants pour la résolution de problèmes réels de très grande taille.

Partie III

Evaluation

Chapitre 7

Stratégie d'évaluation de l'implémentation PEPSys

Comme toute évaluation, celle du système parallèle PEPSys a deux buts principaux. Tout d'abord la validation des idées du modèle de calcul de PEPSys, et l'évaluation critique de leur mise en œuvre. Les deux points ne sont bien sûr pas indépendants puisqu'une mise en œuvre maladroite est susceptible de trahir les concepts les plus prometteurs, la réciproque étant par contre peu vraisemblable.

L'évaluation d'un système informatique pose un certain nombre de problèmes classiques. Il s'agit tout d'abord de définir les paramètres significatifs du système. Le second problème provient de la prise de mesures, qui perturbe le système étudié. Il importe d'évaluer l'incidence de ces perturbations sur les résultats des mesures tout en les réduisant autant que possible. Un autre problème est l'influence de l'environnement sur le système étudié : s'il n'est pas toujours possible de l'éliminer, on essaie de la réduire autant que possible et aussi d'évaluer son ordre de grandeur. Le dernier problème consiste à trouver des programmes de test significatifs, c'est-à-dire qui mettent en œuvre les fonctionnalités les plus importantes du système mesuré.

Ce chapitre est divisé en quatre parties. La première donne les paramètres importants du système parallèle et la façon dont ils sont mesurés. La seconde partie décrit les outils de mesure et évalue l'incidence des mesures sur le comportement du système mesuré. La troisième partie est consacrée aux conditions expérimentales, évaluant l'influence de l'environnement sur les résultats. Enfin la quatrième partie présente les programmes de test utilisés.

7.1 Principaux paramètres mesurés

7.1.1 Performances

La mesure de performances habituelle d'un système Prolog est le *Klip*. Nous avons signalé qu'elle était peu significative. La mesure des performances sera donc pour nous essentiellement le temps passé à exécuter les programmes de test.

L'objectif principal du projet PEPSys est l'amélioration des performances de la programmation logique par utilisation du parallélisme. Le gain d'efficacité procuré par celui-ci est habituellement mesuré par le facteur d'accélération, rapport entre le temps d'exécution d'un programme sur le système séquentiel **le plus efficace existant** et le temps d'exécution du même programme en parallèle. On étudie les variations du facteur d'accélération en fonction du nombre de processeurs utilisés en parallèle. Un système idéal procurerait un facteur d'accélération linéaire en fonction du nombre de processeurs, de pente unitaire, et ce quel que soit le programme testé.

Ne disposant pas du système Prolog séquentiel "le plus efficace" sur le MX500, nous avons retenu comme mesure de facteur d'accélération le rapport entre le temps d'exécution par le système parallèle, sur un seul processeur et le temps d'exécution en parallèle par le même système. Pour que cette mesure ait un sens, l'efficacité du système PEPSys s'exécutant séquentiellement a été soigneusement calibrée en tenant compte de la technique d'implémentation et de la puissance du processeur utilisé. En outre, les mesures de temps d'exécution ont été comparées à toutes les mesures disponibles obtenues avec un système Prolog efficace exécutant les mêmes programmes sur la même machine.

La mesure des performances du système est très simple puisqu'elle consiste seulement à mesurer le temps d'exécution des programmes. Cette mesure est faite en utilisant les procédures du système d'exploitation de la machine. Pour chacun des programmes de test, cette mesure de temps doit être faite pour l'ensemble des nombres de processeurs possibles.

7.1.2 Mesures du modèle de calcul: déréférencement non-local et recherche dans les *hash-windows*

Paramètres du modèle

Les mesures relatives à la gestion des résolvantes OU-parallèles par le modèle de calcul de PEPSys sont parmi les plus importantes pour valider les concepts développés dans ce projet. Le modèle repose sur une hypothèse de localité : les déréférencements impliquant des variables non-locales sont supposés peu fréquents. De plus, lorsque l'opération de déréférencement conduit à suivre une chaîne de *hash-windows*, on suppose que cette chaîne, de longueur non bornée, sera courte. Bien évidemment l'incidence de la longueur des chaînes de *hash-windows* sur la durée de l'opération de déréférencement est moins grande sur une machine à mémoire commune que sur une machine à mémoire distribuée. Même dans ce dernier cas, le temps d'accès à une *hash-window* est loin d'être négligeable (voir ci-dessous) et le bon fonctionnement du modèle exige que les chaînes soient courtes.

Pour évaluer la fréquence des déréférencements non-locaux et des recherches dans les *hash-windows*, les paramètres suivants ont été comptés : nombre total d'opérations de déréférencement, nombre de déréférencements non-locaux, nombre de déréférencements impliquant une recherche dans les *hash-windows*. Pour

évaluer la longueur des chaînes de *hash-windows*, un comptage des déréférencements impliquant l'exploration d'une chaîne de *hash-windows* a été pratiqué, par longueur de chaîne et pour des longueurs de un à vingt et au-delà. De plus, un compteur enregistrait la plus longue chaîne de *hash-windows* explorée.

Evaluation du surcoût du au déréférencement non-local et à la recherche dans les *hash-windows*

Le surcoût total introduit par le modèle de calcul de PEPSys pour la gestion des liaisons en parallélisme OU est difficile à évaluer. Une partie importante des instructions de la machine abstraite est plus complexe que les instructions correspondantes de la *WAM* : même au cours d'une section locale de calcul, certaines instructions doivent tester si une variable permanente est locale ou non (le registre pointeur d'environnement, **E**, étant non-local dans le second cas). Le déréférencement est également plus complexe : il est nécessaire de tester, en fin de déréférencement si l'objet déréférencé n'est pas une variable non-locale ; si c'est le cas, le déréférencement se poursuit en mode non-local, suivant le modèle de calcul de PEPSys. La seule façon de mesurer les sources de surcoût de grain très fin serait d'éliminer de l'émulateur tout ce qui ne fait pas partie de la machine abstraite ICM3 [Noye 87] sur laquelle est basée la machine abstraite PEPSys. Cette expérience n'a pas été faite. Cependant, d'autres résultats permettent d'estimer l'ordre de grandeur du surcoût du au modèle, lors de l'exécution séquentielle de programmes (voir section 8.2.1).

Le déréférencement non-local et la recherche dans les *hash-windows* génèrent un surcoût d'une granularité plus importante. Ces deux opérations sont codées dans des procédures, contrairement au déréférencement local qui est macro-expansé. En plus du temps passé en changements de contexte, ces procédures doivent vérifier la validité des liaisons non-locales et parfois construire pour le processus déréférenceur une représentation "non-locale" de l'objet déréférencé, avant de retourner le résultat.

Il est possible d'estimer précisément les surcoûts dus au déréférencement non-local et à la recherche dans les *hash-windows* en exécutant un jeu de programme construits pour ces mesures. Evidemment, l'optimisation consistant à cacher dans la *hash-window* locale les liaisons valides trouvées dans les *hash-windows* non locales (voir la section 6.4.3) a été supprimée lorsque cette mesure a été faite. Le programme PEPSys utilisé est le suivant :

```
-properties([]).
loc_test :-
par_loc(X, 30000).

-properties([solutions(all), execution(eager),clauses(unordered)]).
par_loc(X,N) :-
X = a,
waste_time.
```

```
par_loc(X, N) :-
Y = a,          /* Dereferencement local de longueur 1 */
loop(N, Y).

-properties([]).
nloc_test :-
X = a, par_nloc(X, 30000).

-properties([solutions(all), execution(eager),clauses(unordered)]).
par_nloc(X,N) :-
waste_time.
par_nloc(X, N) :-
loop(N, X).    /* Dereferencement non-local, liaison valide */

-properties([]).
hw_test1 :-
par_hw1(X, 30000).

-properties([solutions(all), execution(eager),clauses(unordered)]).
par_hw1(X,N) :-
waste_time.
par_hw1(X, N) :-       /* Dereferencement non-local. Liaison de X dans */
X = a,          /* la hash-window locale s'il n'est pas deja lie */
loop(N, X).    /* dans celle d'un processus parent */

-properties([]).
hw_test2 :-
par_hw2(X, 30000).

-properties([solutions(all), execution(eager),clauses(unordered)]).
par_hw2(X,N) :-
waste_time.
par_hw2(X, N) :-
X = a,           /* Liaison de X dans la hash-window locale */
par_hw1(X, N).   /* s'il n'est pas ... */
  ... ...

-properties([]).
hw_test7 :-
par_hw7(X, 30000).

-properties([solutions(all), execution(eager),clauses(unordered)]).
par_hw7(X,N) :-
waste_time.
par_hw7(X, N) :-
X = a,              /* Liaison de X dans la hash-window locale */
par_hw6(X, N).      /* s'il n'est pas ... */
```

```
-properties([]).
loop(0, X) :- time.
loop(N, X) :-                    /* Dereferencement de X, 30.000 fois */
N > 0,                  /* Suivant l'appel, X est local, non-local */
X = a,                  /* lie dans la hash-window locale ou une */
N1 is N -1,             /* hash-window parente */
loop(N1, X).

-properties([]).
waste_time :-
nreverse([a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,
q, r, s, t, u, v, w, x, y, z, a, b, c, d], R).
...
```

Chacun des prédicats de ce programme effectue trente mille déréférencements[1]. Si on appelle le prédicat *loc_test*, le déréférencement demeure local. Par contre, l'appel à *nloc_test* déclenche un déréférencement non-local mais sans recherche dans les *hash-windows* puisque la liaison en place est valide. Les appels à *hw_testi* provoquent des recherches dans des chaînes de *hash-windows* de longueur *i*, une longueur unitaire signifiant que la *hash-window* est locale.

Le surcoût du au déréférencement non-local par rapport au déréférencement local, calculée à partir des mesures précédentes est d'environ 50 microsecondes. Ce surcoût est le double du temps passé dans un déréférencement local, 25 microsecondes dans les mêmes conditions à savoir longueur de la chaîne de référence : 1. Cette dernière mesure a été faite par un programme C bouclant sur l'opération de déréférencement. Les deux mesures ont été corrigées de l'influence de la mémoire cache de chaque processeur (voir ci-dessous). Les résultats des mesures de surcoût du à la recherche dans les *hash-windows* sont donnés à la figure 7.1 ainsi que l'approximation linéaire qui en a été dérivée. Cette approximation a été utilisée pour mesurer le surcoût provenant de la recherche dans les *hash-windows* pour chacun des programmes de test et elle s'exprime par la formule suivante :

$$\text{overhead} = 46 * n + 50$$

dans laquelle n représente la longueur de la chaîne de référence et *overhead* est la valeur du surcoût en microsecondes.

Pour estimer le surcoût du au déréférencements non-locaux et à la recherche dans les *hash-windows*, l'influence des mémoires caches associées aux processeurs a été prise en compte. Nous avons supposé que durant les mesures de surcoût, le code et les données étaient présents dans les caches du processeur impliqué. Suivant la documentation de Sequent, on peut considérer que la présence dans le

[1]Ce nombre est proche de la valeur maximale des entiers courts, seuls à être implémentés sur PEPSys.

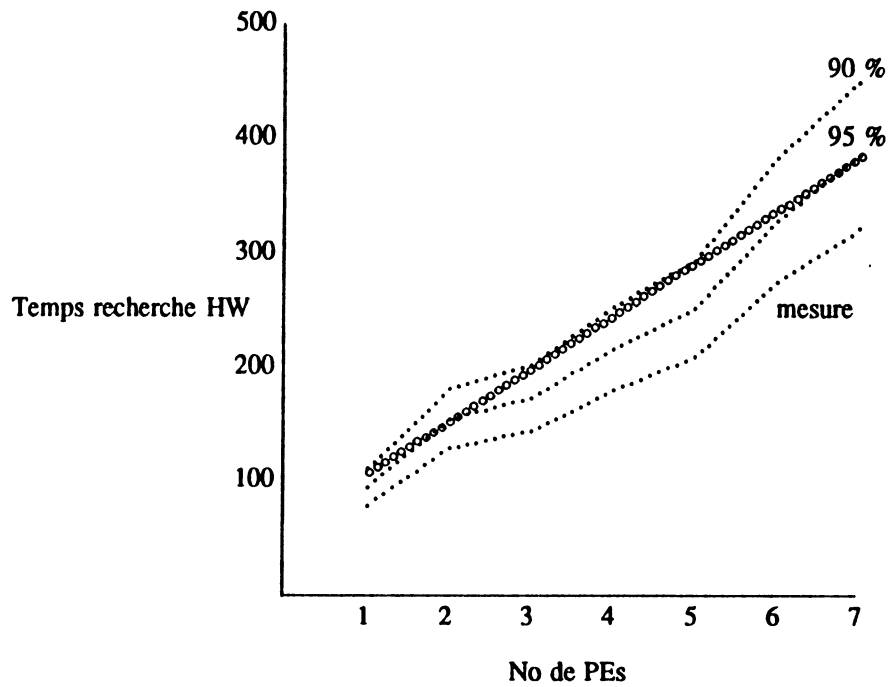

Figure 7.1: Mesure du surcoût provenant de la recherche dans les *hash-windows*

Les trois courbes en pointillés donnent les valeurs de surcoût mesurées ainsi que des valeurs calculées pour des taux de succès de 90 % et 95 %. Le courbe en o représente l'approximation linéaire utilisée pour les estimations de ce surcoût.

cache divise le temps d'exécution par un facteur cinq[2]. Dans la figure 7.1, deux taux de succès ont été considérés pour l'accès au cache, les plus vraisemblables pour des caches de cette taille selon la littérature, 90 % et 95 %. Les points des courbes correspondantes de la figure 7.1 ont donc été calculées à l'aide des formules :

```
Overhead = overhead-mesure*0.95 + (overhead-mesure*5)*0.05
         = overhead-mesure*1.2
Overhead = overhead-mesure*0.90 + (overhead-mesure*5)*0.10
         = overhead-mesure*1.4
```

L'approximation linéaire correspondant au taux de succès donné par Sequent, à savoir 95 %, a été retenue comme base pour calculer les surcoûts du aux déréférencements non-locaux et aux recherches dans les *hash-windows*.

Remarque importante : l'évaluation de le surcoût du aux déréférencements non locaux et à la recherche dans les *hash-windows* présentée ci-dessus correspond en fait à un minorant de ce surcoût. L'objet non local déréférencé dans ces mesures

[2]Ce chiffre a été vérifié en comparant les temps d'exécution de deux exécutions d'une même boucle, l'une d'entre elles "vidant" le cache avant de faire chaque opération élémentaire.

est en effet un atome ce qui n'implique pas de transformation par le système contrairement aux listes, structures et variables libres. Toutes les estimations fournies dans la section 8.3 et qui en sont dérivées sont donc des minorants du surcoût effectif du au modèle de calcul.

7.1.3 Activité des processeurs

Paramètres mesurés

L'une des principales sources de surcoût durant l'exécution de programmes parallèles est le temps passé par les processeurs à chercher du travail. Ce temps dépend de plusieurs facteurs. Tout d'abord du programme considéré : certains programmes offrent plus de travail que d'autres. Le second facteur est le nombre de processeurs utilisés. Sur un programme de petite taille, l'augmentation du nombre de processeurs tend à épuiser les possibilités de travail parallèle ou à raccourcir la taille des processus. Le pourcentage du temps calcul consacré à la recherche de travail et à la création de processus augmente donc avec le nombre de processeurs utilisés. Le troisième facteur important dans ce domaine est l'ordonnanceur utilisé. La granularité des processus augmente avec la qualité de l'ordonnanceur.

A la base de la mesure de l'activité des processeurs, on trouve l'enregistrement de la durée de vie de chaque processus. Deux compteurs sont associés à chaque processus, initialisés à leur création et enregistrés à leur terminaison : un compteur de temps d'exécution et un autre du nombre d'inférences exécutées. Les processus sont classés par intervalles semi-logarithmiques : de 0 à 10 millisecondes, de 10 à 20 ms, ..., de 90 à 100 ms., de 100 à 200 ms., etc. jusqu'à 100 secondes et au delà, pour la mesure du temps et de 1, 2, ..., 9 inférences, de 10 à 20 inférences, de 20 à 30, ..., jusqu'à 10.000 inférences et au delà. Lorsqu'un processus se termine, ses deux compteurs de durée de vie permettent de le classer dans les intervalles correspondants et d'incrémenter le nombre de processus de chacun des intervalles. Des compteurs sont également associés à chacun des processeurs pour compter le nombre d'inférences et le temps d'activité de chaque processeur. Les compteurs des processeurs sont également mis à jour à la fin de l'exécution de chaque processus.

La mesure du temps pose un problème de précision. En effet, les procédures du système UNIX permettant la mesure du temps ont une forte granularité (10 millisecondes), supérieure à la durée de vie des processus les plus courts (quelques inférences). En toute rigueur, il faudrait donc utiliser d'autres techniques de mesure du temps[3]. Cependant, les erreurs potentielles proviennent de processus qui pèsent peu dans le résultat global. De plus, étant donné le grand nombre de mesures de temps effectuées durant l'exécution d'un programme parallèle, il semble que les erreurs se compensent. La vraisemblance de la mesure de l'oisiveté apparaît lorsqu'on l'ajoute aux autres sources de surcoût identifiées et mesurées. Le surcoût

[3]On pourrait par exemple spécialiser un processeur à incrémenter un compteur partagé en lecture. Les autres processeurs se serviraient de la valeur de ce compteur comme d'une date, d'une précision de quelques microsecondes. C'est la solution qui a été adoptée par [Sseridi 89]

total obtenu est alors proche de l'écart entre le système réel et un système idéal.

7.1.4 Autres mesures liées au parallélisme

Pourcentage de travail parallèle effectivement utilisé

Au cours de l'exécution d'un programme parallèle les processus actifs créent des possibilités de travail qui peuvent ou non être utilisées par des processeurs inactifs pour créer des processus. On mesure ici le nombre de structures de données "parallèles" créés durant l'exécution des programmes ainsi que le nombre d'entre elles effectivement utilisées pour la création de processus. Ces mesures indiquent les risques de famine, - si la majorité des structures est utilisée pour créer des activités parallèles -. Elles permettent également de calculer le surcoût du à la création de ces possibilités de travail.

Surcoût du à la création de possibilités de travail

Les mesures classées dans cette rubrique évaluent le surcoût provoqué, au cours de l'exécution séquentielle, par la création de travail potentiel. Comme nous l'avons vu dans la section 5.2.2, la création de travail potentiel par les processeurs actifs se fait au moyen d'instructions spécialisées de la machine abstraite PEPSys, opérant sur les *branch-points* et *fork-points*. L'exécution de ces instructions introduit un surcoût, comparativement à l'exécution d'instructions équivalentes ne générant pas de parallélisme pour plusieurs raisons liées le plus souvent à des choix simplificateurs.

La première raison, commune aux deux types de parallélisme, est que toutes les structures de données offrant du parallélisme sont rendues publiques. Leur accès par le processeur local, au retour arrière dans le cas du parallélisme OU et en exécutant la continuation dans le cas du parallélisme ET, doit donc être protégé par un verrou d'exclusion mutuelle. Or, le coût de la prise de verrou n'est pas négligeable sur le MX500 : environ 50 microsecondes. Pour limiter le surcoût du à ce facteur, il serait nécessaire de ne rendre publique les structures parallèles que lorsque tout le travail offert par le processus en cours a été utilisé. Cette solution implique l'implémentation d'un mécanisme permettant au processeur prenant le dernier travail offert d'interrompre l'exécution de celui qui lui fournit ce travail, afin qu'il "relâche" d'autres travaux.

Les autres sources de surcoût diffèrent entre les parallélismes OU et ET. La création de travail potentiel ET parallèle est faite en exécutant des instructions et en créant des structures de données sans équivalents dans une exécution séquentielle. Ce n'est pas le cas du parallélisme OU pour lequel la création de travail se fait par des instructions analogues à celles qui seraient exécutées séquentiellement. Le choix non optimal des instructions a déjà été mentionné dans la section 5.2.2. Toujours dans le cas du parallélisme OU, la structure des *branch-points* regroupe toutes les parties fixes pour en faciliter l'accès, ce qui provoque un mélange entre les données communes aux points de choix, utilisées par le retour-arrière, et les

données utilisées par l'ordonnanceur. Cette structure simplifie l'implémentation de l'ordonnanceur au détriment du temps d'accès aux variables permanentes d'un *branch-point* lorsque celui-ci est utilisé comme un point de choix. La solution à ce problème consisterait à séparer nettement dans un *branch-point* la partie "point de choix" de la partie "parallèle", afin de ne pas alourdir le retour arrière sur *branch-point*. La partie "parallèle" serait accédée par indirection depuis la partie point de choix et ne serait initialisée qu'en cas d'utilisation effective du *branch-point*.

Pour mesurer le surcoût du à la création de possibilités de parallélisme, on compte le nombre de *branch-points* et *fork-points* non utilisés pour la création de processus et on multiplie par le surcoût que provoque chacun d'entre eux. Celui-ci est mesuré en exécutant séquentiellement une variante du même programme dans laquelle on remplace les déclarations de prédicats OU-parallèles (*execution(eager)*) par leur contrepartie séquentielle (*execution(lazy)*) et on remplace l'opérateur d'indépendance (#) par une virgule. On déduit des temps d'exécution et du nombre de structures "parallèles", créées par la version "parallèle" du programme, le surcoût par *branch-point* ou *fork-point*. Celui-ci est fixe pour les *fork-points* et dépend dans le cas des *branch-points* du nombre d'alternatives qu'ils offrent : plus ce nombre est important et plus le nombre de retour-arrières et donc le surcoût du à l'utilisation de *branch-points* au lieu de points de choix sont grands.

Dans les programmes OU-parallèles, on enregistre également la valeur maximale atteinte par l'*OBL* durant l'exécution. Cette mesure a deux intérêts : d'une part évaluer l'espace nécessaire à l'inscription de l'*OBL* dans les éléments de piles, d'autre part donner une indication de la profondeur maximale atteinte par les processus OU-parallèles[4], en indiquant le nombre maximal de *branch-points* empilés durant l'exécution. Dans les programmes ET parallèles, on enregistre une mesure équivalente : le nombre maximal de *fork-points* empilés par un processeur.

Compétition

La compétition (*contention*) pour l'accès aux ressources partagées est une source de surcoût classique dans les systèmes parallèles. On mesure ici le temps passé par les processeurs à attendre la libération des verrous d'exclusion mutuelle. Pour ce faire, la boucle d'attente active de la procédure de verrouillage a été modifiée afin d'en compter le nombre d'exécutions[5]. Par ailleurs, le temps d'exécution d'une boucle a été facilement mesuré en l'exécutant un grand nombre de fois. Le compteur du nombre de boucles d'attente active est enregistré dans des intervalles semi-logarithmiques similaires à ceux qui enregistrent la durée de vie des processus.

[4] Il ne s'agit bien sûr que d'une indication, les processus OU-parallèles de certains programmes étant susceptibles de développer de longues branches de calcul tout en ne créant que peu de *branch-points*.

[5] Pour ne pas saturer le bus de communication, la boucle teste la valeur d'une image, locale au cache du processeur, de la variable représentant le verrou. La libération du verrou provoque la mise à jour du cache et le test du verrou réel.

7.1.5 Consommation mémoire

Il est classique de considérer qu'une implémentation de système informatique résulte d'un compromis entre la consommation d'espace mémoire et l'efficacité. Le parallélisme apporte un nouvel exemple de ce type de compromis : on en attend des gains d'efficacité mais il est clair que la mise en parallèle de plusieurs machines abstraites Prolog se traduira par un accroissement de la consommation mémoire.

Une des principales sources de surconsommation mémoire de l'implémentation de PEPSys, par rapport à la majorité des systèmes Prolog séquentiels[6], provient de l'utilisation de deux mots mémoire par variable Prolog sur la pile. Ce surcoût dépend du pourcentage de variables par rapport aux informations de contrôle dans les piles. Il apparaît même lors de l'exécution séquentielle d'un programme PEPSys sur un seul processeur et il n'a pas été pris en compte ici. L'autre source de surconsommation mémoire provient de la création de structures de données spécifiques au parallélisme comme les *hash-windows, rootframes, branch-points* et *fork-points*. Enfin la présence de trous noirs dans les piles, provenant de l'utilisation d'une autre stratégie d'ordonnancement que "aider ou attendre", augmente la surconsommation mémoire durant l'exécution d'un programme en parallèle.

La mesure qui est faite ici est celle de la quantité de mémoire consommée par les piles des processeurs. Cette mesure constitue en réalité un majorant puisqu'il s'agit de la somme des maximums consommés par chacun des processeurs au cours de l'exécution parallèle, maximums qui ne sont pas nécessairement atteints au même instant.

7.2 Systèmes utilisés

Deux versions du système parallèle, différant par des options de compilation, ont été utilisées pour les mesures. La première version n'a été utilisée que pour les mesures de temps d'exécution des programmes. Elle ne comporte que la mesure du temps d'exécution, faite sur le processeur qui lance l'exécution du programme et le termine. Etant données les conditions expérimentales (voir la section 7.3.1) qui assurent qu'aucun processeur virtuel ne peut être suspendu par le système d'exploitation, il n'est pas nécessaire de mesurer les temps d'exécution des autres processeurs. Cette hypothèse de travail a été confirmée en calculant la différence entre les dates au début et à la fin de l'exécution.

La deuxième version du système effectue en outre toutes les autres mesures mentionnées dans la section précédente. Ces mesures sont enregistrées dans des variables locales par chaque processeur. En fin d'exécution, chacun des processeurs met à jour en exclusion mutuelle une structure unique qui rassemble toutes les mesures. Les mesures individuelles de chaque processeur peuvent être imprimées à l'écran, ce qui est utile pour vérifier des résultats surprenants, mais elles ne sont

[6]Bien que certains d'entre eux utilisent un mot mémoire particulier pour l'étiquette (*tag*) [Meier 89].

pas conservées à cause du trop grand volume de données qui serait alors généré. Les champs de la structure rassemblant les mesures sont organisés en tableaux permettant de stocker plusieurs mesures faites au cours d'une même session. En fin de session, les moyennes et écarts types de chacune des mesures sont calculés et stockés dans un fichier. Cette deuxième version du système est environ 10 % plus lente que la version la plus efficace.

7.2.1 Enchaînement des mesures, exploitation des résultats

Un système automatique d'enchaînement de mesures et d'exploitation de résultats a été mis en place pour pouvoir mesurer automatiquement le système pour l'ensemble des programmes de tests. Les programmes de test sont exécutés chacun cinq fois sur chaque configuration possible de un à huit processeurs, tout d'abord par le système le plus efficace, puis par le système de mesure. Les résultats sont ensuite traités pour produire un tableau de mesures détaillées par programme de test ainsi qu'un tableau général résumant les temps d'exécution et facteurs d'accélération de l'ensemble des programmes. L'ensemble de ces tableaux récapitulatifs est donné dans [Chassin 89].

7.2.2 Discussion: utilisation de l'*Instant replay* pour les mesures

Il semblerait logique d'utiliser le système d'*Instant Replay* pour pratiquer les mesures où le temps n'intervient pas. Celles-ci se feraient au cours de la phase de réexécution (*Replay*). Cette solution n'a pas été retenue pour plusieurs raisons. La première est que les résultats obtenus par cette méthode sont peu différents de ceux que produit la version instrumentée du système. Cette vérification n'a pas été systématique mais tous les tests faits ont montrés des différences entres les résultats obtenus par les deux systèmes inférieurs aux marges d'incertitude. La faiblesse de ces écarts tient sans doute au faible surcoût introduit par les mesures : moins de 10 % à comparer aux 2 % introduits par l'enregistrement des synchronisations dans l'*Instant Replay*. La seconde raison est la complication que cette technique de mesure aurait introduite. Quatre séries de tests seraient devenues nécessaires ; les deux premières étant les mêmes que celles pratiquées actuellement et utilisées pour les mesures de temps d'exécution et de durées de vie des processus et oisiveté. Les deux dernières auraient été l'enregistrement et la réexécution pour les autres mesures. La dernière raison pour ne pas utiliser l'*Instant Replay* pour les mesures provient de la façon dont ce système a été implémenté. Nous avons vu qu'il s'agissait d'une implémentation très efficace mais très liée à l'ordonnanceur originel. Il aurait donc été nécessaire de l'adapter à chacun des ordonnanceurs testé, ce qui n'a pas été possible.

7.3 Conditions expérimentales

7.3.1 Influence du système d'exploitation

Le système parallèle PEPSys est implémenté sur une variante du système UNIX. Lorsque l'on procède aux mesures, il faut garantir qu'à chacun des processus UNIX, représentant un processeur virtuel, est alloué un processeur réel. Le système d'exploitation offre la possibilité d'allouer statiquement les processus aux processeurs, pour un programme exécuté en mode privilégié. Cependant, cette assignation statique provoque parfois des dégradations de performances considérables, lorsque le test est perturbé par l'exécution d'autres processus du système d'exploitation.

Les mesures de cette thèse ont été effectuées en utilisant l'utilitaire *team*, qui permet aux processus UNIX non-privilégiés de s'exécuter dans des conditions proches de la correspondance un-un avec les processeurs réels. *Team* donne la priorité maximale aux processus UNIX qu'il exécute, tout en prohibant leur préordonnancement et l'échange de pages (*swapping*). L'exécution de PEPSys sous *team*, en utilisant 8 processeurs virtuels, stoppe l'ensemble des processus utilisateurs et donne un temps de réponse proche du temps utilisateur des processus UNIX mis en œuvre. Pour rendre l'association entre processus UNIX et processeurs réels plus réelle, les mesures ont été pratiquées en utilisant le système d'exploitation en mode mono-utilisateur, ce qui élimine les processus du système gérant le courrier, le réseau, etc.

7.3.2 Exécutions répétées

Chacun des programmes de test a été exécuté cinq fois sur chaque nombre possible de processeurs, de un à huit. La première de chacune des séries de cinq mesures n'a pas été prise en compte, ceci afin de limiter l'influence des caches. Les mesures suivantes sont en effet le plus souvent meilleures en raison de la présence initiale dans les caches des processeurs d'une partie du code et des données du système parallèle. Les quatre résultats à chacune des mesures ont permis de calculer leurs moyennes et écarts types. Les écarts types sont habituellement faibles, un écart type important est assimilé à une perturbation due au système d'expoitation durant le test. De telles perturbations sont exceptionnelles, moins de une par série de 1120 exécutions utilisées pour les mesures, et elles ont été éliminées en réexécutant le programme posant problème.

En dépit de la faiblesse des écarts types, deux séries de mesures peuvent présenter des moyennes dont l'écart est supérieur à la somme des écarts types de chaque série de mesures. Pour prendre en compte ce phénomène, il aurait été nécessaire de pratiquer chaque mesure au cours d'une session distincte au lieu de procéder à des exécutions consécutives. Une telle modification du système de mesure n'a pas été considérée comme nécessaire en raison de la limitation des écarts entre séries de mesures, moins de 1 %, du caractère non systématique de ce phénomène et enfin de l'accroissement de la durée des tests qui en aurait résulté.

7.3.3 Influence du compilateur

Le compilateur C du système Dynix de Sequent a été utilisé tout au long du développement du système parallèle car il supporte une librairie parallèle écrite en C. Par la suite, le compilateur GNU ayant été installé sur la machine, le système a été adapté pour l'utiliser. Ce compilateur est en effet réputé générer un code objet 40 % plus efficace en moyenne que le code objet généré par le compilateur Dynix. Les appels à la librairie parallèle ont été regroupés dans un fichier unique, compilé à l'aide du compilateur Dynix. La plupart des autres fichiers ont été compilés avec le compilateur GNU en mode optimisé. L'optimiseur n'a toutefois pu être utilisé pour la boucle principale de l'émulateur en raison de la taille de cette boucle et vraisemblablement d'une erreur dans le compilateur. En dépit de ce problème, la version du système compilée avec GNU est 10 % plus efficace que celle compilée avec le compilateur Dynix en mode optimiseur.

7.3.4 Paramètres du système

Les paramètres dont il est question ici permettent d'ajuster manuellement ce qui devrait l'être automatiquement sur un système "réel". Il s'agit tout d'abord de la taille des *hash-windows* qui devrait être petite par défaut, de nouvelles *hash-windows*, connectées aux précédentes, étant affectées au processus en cas de saturation. La solution simple retenue ici est de choisir une taille suffisante pour suporter l'ensemble des programmes de test. Alors que la plupart des programmes se contentent de moins de seize éléments, il a été nécessaire, à cause du *puzzle de Baskett* (voir section 7.4.2) d'utiliser des *hash-windows* de 128 éléments, soit 2K octets. Le second paramètre est la taille des zones de données allouées sur la pile globale pour stocker le résultat des *bagof*, qui a été fixée à 5000 éléments de pile soit 40 K-octets. Il s'agit cette fois ci d'un palliatif à l'absence de tas permettant d'allouer de la mémoire dynamiquement.

7.4 Programmes de test

Il n'y a pas eu de réelle sélection des programmes de test. Lorsque les mesures ont été faites, nous avons utilisé tous les programmes alors disponibles en langage PEPSys. Ces programmes sont soit des programmes écrits spécialement dans le langage, soit des programmes Prolog parallélisés. Comme nous le verrons, la variété de ces programmes permet de tester sérieusement l'implémentation parallèle. On trouve aussi bien des programmes jouets de quelques prédicats, très utiles pour mettre au point l'implémentation, que des programmes de plusieurs milliers de clauses. Les temps d'exécution de ces programmes séquentiellement varient entre quelques centaines de millisecondes et quelques centaines de secondes. On peut classer ces programmes en trois catégories : tout d'abord les programmes OU-parallèles "toutes solutions" dans lesquels on recherche l'ensemble des solutions,

ensuite les programmes OU-parallèles "une-solution" où l'utilisateur ne désire que la première solution et enfin les programmes ET-parallèles[7].

Les programmes les plus courts en temps calcul sont exécutés plusieurs fois lors de chaque mesure. La raison est l'incertitude introduite dans la mesure du temps d'exécution d'un programme parallèle par la durée nécessaire à arrêter une exécution parallèle à la fin du programme. A cela s'ajoute l'incertitude provenant du caractère non reproductible des exécutions qui introduit des variations relatives importantes entre des mesures différentes de programmes courts. Enfin la granularité de la mesure du temps du système d'exploitation est relativement plus importante pour ce type de programme.

7.4.1 Programmes OU-parallèles toutes solutions

Farmer : Le programme *farmer* donne une solution au problème du fermier, du loup, de la chèvre et du chou. Il s'agit du plus petit problème (seulement 300 inférences) de test exécuté. Pour cette raison, la solution est calculée 100 fois à chaque exécution du programme. Ce programme a été très utile pour mettre au point l'implémentation. Depuis il a conservé son utilité en tant que pire exemple possible pour tester une nouvelle version du système, en raison de ses mauvaises performances en parallèle.

Hamilton : Le problème consiste à trouver un chemin fermé à travers un graphe, de telle sorte que tous les nœuds du graphe ne soient visités qu'une seule fois. Cet algorithme est utilisé dans la solution au problème du "voyageur de commerce". Le programme est petit, - vingt clauses "base de données" décrivant le graphe et neuf clauses seulement pour calculer le chemin -, mais ce problème présente, avec 460.000 inférences, le plus grand nombre d'inférences des programmes de mesure utilisés dans cette thèse.

Houses : Ce programme résoud un puzzle dans lequel cinq personnes de pays différents habitent dans cinq maisons différentes d'une même rue. La connaissance d'un certain nombre de relations permet de résoudre ce puzzle. Le programme utilise une recherche dirigée par contrainte. Le problème étant petit, - moins de 2500 inférences -, il est exécuté vingt fois.

Mandel : Ce programme calcule un ensemble de Mandelbrot (problème de géométrie fractale) de trois cent points. Le programme comporte près de 74.000 inférences et n'est exécuté qu'une seule fois.

Map : Ce programme calcule toutes les solutions (cent vingt) à un petit problème de coloration de graphe. Il comporte 22.800 inférences et n'est exécuté qu'une seule fois.

[7]Rappelons que seul le parallélisme ET déterministe est implémenté et les programmes testés appartiennent à cette catégorie.

Queens1 : *Queens1* donne une solution au problème des huit reines. Afin de restreindre l'espace de recherche, une liste de valeurs possibles est passée à la procédure de génération. Si, pour une colonne donnée, une ligne est sélectionnée, alors cette ligne est enlevée de la liste. Ainsi il n'est pas nécessaire de vérifier les contraintes horizontales et verticales. Les mesures sont faites lors du calcul de l'ensemble des solutions au problème des huit reines (92 solutions), qui n'est exécuté qu'une fois (103.000 inférences).

Queens2 : *Queens2* donne la solution "naturelle" au problème des huit reines. Pour chacune des colonnes, toutes les valeurs de ligne possibles sont testées. Toutes les solutions au problème sont calculées une fois lors des mesures (233.000 inférences).

Salt : Le programme *salt* donne une solution au puzzle "sel et moutarde" de Lewis Carroll [Disz 87]. Bien que le problème n'ait qu'une seule solution, l'ensemble de l'espace de recherche est exploré. Le programme comporte moins de 23.000 inférences et est exécuté une seule fois.

TInA1, TInA2 et TInA3 : Le but de *TInA (Tourist Information Adviser)* est de proposer des circuits touristiques à des touristes en fonction de leurs souhaits. Comme il est expliqué dans [Ing 87a], trois algorithmes différents donnent une solution au même problème. L'algorithme de *TInA1* s'inspire de la compilation du parallélisme ET en parallélisme OU et utilise un nombre important de *bagofs*. *TInA2* utilise la combinaison complète des parallélismes ET et OU. *TInA3* n'utilise que le parallélisme OU. Selon l'analyse "abstraite" du parallélisme présenté par ces trois algorithmes, c'est le programme *TInA2* qui offre les meilleures perspectives de parallélisme et d'efficacité sur un multiprocesseur disposant de ressources infinies [Ing 87a]. Sur le système PEPSys, qui n'implémente pas la combinaison des parallélismes ET et OU, seul le parallélisme OU est exploité et la réexécution répétée des branches prévues pour s'exécuter en parallélisme ET fait de *TInA2* le plus gros des trois algorithmes (259.000 inférences contre 152.000 pour *TInA1* et 222.000 pour *TInA3*).

Les programmes *TInA*s présentent un certain nombre de caractéristiques intéressantes. Tout d'abord, contrairement aux autres programmes de test originellement écrits en Prolog puis traduits en PEPSys, ils ont été développés à ECRC en PEPSys dans le but de tester le langage sur une application réelle et de fournir des programmes de test pour le simulateur et le système parallèle PEPSys. Ce ne sont pas des programmes jouets puisqu'ils résolvent une application réelle. Ils comportent plusieurs centaines de clauses divisées en modules séquentiels qui présentent une interface utilisateur confortable, auxquelles s'ajoute une base de données décrivant les attractions touristiques de la Bavière et comportant plusieurs milliers de clauses.

Pour les mesures présentées dans cette thèse, seul le module parallèle et un sous-ensemble de la base de données restreint à la région de Munich ont été

utilisés. Des expériences d'utilisation de l'ensemble de *TInA* sont décrites dans [Rapp 88].

Chat80 : *Chat80* est un programme Prolog de grande taille, écrit par D.H.D. Warren, qui a été converti en PEPSys. Il comporte une base de données géographique ainsi qu'un analyseur de requêtes à la base de données, exprimées en langue naturelle.[8] Sept mesures sont faites en utilisant *Chat80* : cinq d'entre elles utilisent l'analyseur de requêtes tandis que les deux autres accèdent la base de données en utilisant le résultat de l'analyse des deux phrases les plus complexes. L'ensemble de l'espace de recherche est exploré même si la requête n'a qu'une seule solution. Les mesures ne portent que sur le module séquentiel de *Chat80*. Des résultats portant sur l'utilisation du programme complet sont donnés dans [Rapp 88]. Les requêtes utilisées pour les mesures sont les suivantes :

Parse1 : analyse de la phrase : " *Is London in the United-Kingdom?*". La solution n'exécutant que 920 inférences, le programme est exécuté 20 fois à chaque mesure.

Parse2 : analyse de la phrase : " *Which European country borders a country in Asia?*". La solution n'exécutant que 3400 inférences, le programme est exécuté 20 fois à chaque mesure.

Parse3 : analyse de la phrase : " *Where is China?*". La solution n'exécutant que 760 inférences, le programme est exécuté 20 fois à chaque mesure.

Parse4 : analyse de la phrase : " *Which European countries bordering a country in Asia contain a city the population of which is more than one million?*". La solution exécute 12.400 inférences et le programme est exécuté 5 fois à chaque mesure.

Parse5 : analyse de la phrase : " *Which European countries that contain a city the population of which is more than one million and that border a country in Asia containing a city the population of which is more than 3 million border a country in Western-Europe containing a city the population of which is more than 1 million?*". La solution exécute 44.000 inférences et n'est exécutée qu'une seule fois à chaque mesure.

db4 et db5 : il s'agit des recherches dans la base de données utilisant comme requêtes d'appel les résultats produits par *parse4* et *parse5*. Les solutions font 6200 et 4400 inférences et sont exécutées dix fois à chaque mesure.

7.4.2 Programmes OU-parallèles une solution

La plupart des programmes de la catégorie précédente peuvent aussi être classés dans cette catégorie si au lieu de calculer l'ensemble des solutions ou d'explorer

[8] Dans ce contexte, il s'agit évidement de l'Anglais...

l'ensemble de l'arbre de recherche on ne s'intéresse qu'à une seule solution. On trouve dans cette catégorie les programmes *queens* et *hamilton*. Deux programmes, le *master-mind* ainsi que le puzzle de *Baskett* ont par contre été conçus pour ne produire qu'une seule solution et ne termineraient pas si l'on tentait d'explorer l'ensemble de leur arbre de recherche. Ces programmes utilisent le prédicat prédéfini *toponeof* (voir section 6.7.2) qui utilise les interruptions du système d'exploitation pour arrêter l'exécution aussitôt qu'une solution a été produite. Les nombres d'inférences exécutées dépendent du nombre de processeurs utilisés et des conditions d'exécution. Elles dépendent également de la technique d'implémentation du *oneof*, étant habituellement moins nombreuses avec un *killing* qu'avec un *backbranching*.

Master-mind (mm) : Ce programme est l'un des seuls programmes à utiliser, en plus du *toponeof* déjà mentionné, un *oneof* s'exécutant en parallèle (et qui ne peut donc être compilé par un *cut*). Cette particularité explique son utilisation intensive commme programme de test du *killing* dans le simulateur PEPSys [Baron 88]. Ici, il s'agit du seul programme permettant de tester le *backbranching* (voir section 6.4.6).

Le puzzle de Baskett : ce programme résoud le fameux *puzzle de Baskett* [Tick 88]. Ce problème a donné naissance à des solutions en C, Fortran, Pascal et Lisp. La solution Prolog a été écrite par Evan Tick puis parallélisée ultérieurement par Seif Haridi pour le système Aurora. Ces deux dernières versions ont été publiées dans le *newsgroup* Prolog du réseau *usenet*. La version utilisée ici est la dernière, convertie en PEPSys. Alors que l'ensemble des autres programmes peut s'exécuter avec des *hash-windows* de moins de 32 éléments, ce programme impose d'offrir près de 128 éléments.

7.4.3 Programmes ET parallèles déterministes

La majorité du travail effectué pour implémenter et tester PEPSys a été consacrée au support du parallélisme OU. Nous avons vu que seul le parallélisme ET déterministe avait été implémenté et que cette implémentation ayant été calquée sur celle du parallélisme OU n'était en rien optimale. Seuls deux programmes ont été utilisés pour mesurer l'implémentation du parallélisme ET déterministe : il s'agit des programmes "jouets" *fibonacci* et *quick-sort*. Les mesures du parallélisme ET ont été comparées à une autre série de mesures des mêmes programmes pour lesquels le parallélisme ET a été transformé en parallélisme OU, en utilisant la méthode de [Carlsson 88a] décrite dans la section 3.4.7.

Fibonacci (fib) : il s'agit bien sûr de l'implémentation naive du calcul des suites de Fibonacci en PEPSys. Les mesures sont faites pour le calcul de *fibonacci(20)*, qui exécute près de 66.000 inférences dans la version ET parallèle et plus de 99.000 dans la version OU-parallèle.

Quick-sort (qsort) : ce programme est l'implémentation de l'algorithme *quick-sort* en PEPSys. Les mesures enregistrent le tri d'une liste de 500 entiers tirés au hasard, qui exécute 11.500 inférences dans la version ET parallèle et 16.000 inférences dans la version transformée.

7.5 Résumé du chapitre

Les principaux paramètres permettant d'évaluer le système parallèle PEPSys sont l'efficacité et le facteur d'accélération en exécution parallèle mais aussi le comportement du modèle de gestion des liaisons : nombre de références non-locales et dans les *hash-windows*, longueur des chaînes de *hash-windows*, surcoût du au déréférencement non-local et à la recherche dans les *hash-windows*. Les autres paramètres importants à mesurer sont l'activité des processeurs, la consommation mémoire, ainsi que toutes les sources possibles de surcoût. Deux versions du système permettent les mesures : la première est réservée aux mesures de vitesse, l'autre, instrumentée, effectue toutes les autres mesures. Le surcoût provenant des mesures demeure limité et l'influence de l'environnement, principalement due à l'utilisation du système d'exploitation de la machine, a été limitée. Tous les programmes PEPSys disponibles ont été utilisés pour les tests, leur nombre et leur variété assurant une évaluation sérieuse du système parallèle.

Chapitre 8

Résultats expérimentaux

8.1 Introduction

Un grand nombre de mesures ont été collectées en utilisant les systèmes décrits au chapitre précédent. Une partie importante des résultats obtenus est présentée dans [Chassin 89] ; un échantillon des résultats produits pour chaque programme de test est donné dans l'annexe B. La suite de ce chapitre se concentre sur les plus importants d'entre eux et essaie d'en tirer une interprétation.

8.2 Performances

8.2.1 Exécution séquentielle

L'efficacité du système PEPSys exécutant des programmes Prolog séquentiellement est bonne comparée aux autres systèmes Prolog. Le fameux programme de test *naive-reverse* appliqué à une liste de 30 éléments est exécuté 10 fois en 1,57 secondes, correspondant à une "vitesse" de 3,16 Klips. Cette mesure peut sembler peu impressionante comparée aux vitesses de plus de 100 Klips annoncées pour certains systèmes Prolog commerciaux sur des stations de travail. On peut expliquer cet écart en faisant intervenir deux facteurs : l'efficacité du processeur utilisé ainsi que la technique d'implémentation utilisée. Comme nous l'avons vu le système PEPSys est basé sur un émulateur séquentiel de la machine abstraite PEPSys écrit en langage C. C'est également la technique utilisée par le système SICStus Prolog [Carlsson 88b]. Une traduction soigneuse de la boucle d'émulation en assembleur augmente les performances d'un système Prolog d'un facteur compris entre deux et trois. Cette technique est utilisée par le système Quintus Prolog [Quintus 85]. L'utilisation d'un compilateur optimisé générant du code natif, technique employée par les systèmes Prolog les plus efficaces, multiplie les performances par un autre facteur compris entre deux et trois. Cette technique d'implémentation est utilisée par les systèmes ALS Prolog et BIM Prolog. Appliquée au système PEPSys, elle permettrait d'atteindre une vitesse de 12 Klips par processeur sur le MX500.

Plus sérieusement, la comparaison du systèmes PEPSys avec le système Prolog SICStus indique que le système PEPSys est 30 % plus lent en exécutant les mêmes programmes sur le même processeur (voir table 8.2). Ce genre de comparaison portant sur un ensemble de programmes est d'ailleurs plus sérieux qu'une mesure brute sur un programme peu significatif comme *naive-reverse*. Cet écart de 30 % provient essentiellement des surcoûts difficilement mesurables qu'introduit le modèle de calcul de PEPSys (voir section 7.1.2). Il provient vraisemblablement aussi de simplifications adoptées pour le codage de l'émulateur PEPSys (par exemple l'unification est récursive) et pourrait sans doute être réduit.

8.2.2 Programmes OU-parallèles toutes solutions

La table 8.1 donne les résultats obtenus par l'exécution des programmes OU-parallèles toutes solutions. Ces résultats appellent quelques remarques :

- tous les programmes utilisés montrent des facteurs d'accélération lors de l'exécution parallèle, comparés à l'exécution séquentielle sur le système PEPSys. Nous verrons ultérieurement (voir table 8.2 et figure 8.2) qu'il en est de même si l'on compare les temps d'exécution à ceux du système Prolog SICStus s'exécutant sur le même processeur. Cela prouve que l'exécution parallèle d'un programme PEPSys n'est pas nuisible à l'efficacité, même si le programme considéré a un comportement médiocre en parallèle.

- les "petits" problèmes procurent des facteurs d'accélération limités. La définition de petit est évidement assez arbitraire. Pour les programmes OU-parallèles toutes solutions, nous avons classé dans cette catégorie les problèmes dont la solution calcule moins de 20,000 inférences. Tombent dans cette catégorie les programmes *farmer, houses, parse1, parse2, parse3, parse4, db4* et *db5*. Les résultats à l'intérieur de cette catégorie sont loin d'être uniformes puisqu'on note des écarts de performances importants entre le plus petit, *farmer* et les plus gros *parse4, db4* et *db5*. On peut noter que le temps de réponse d'un système Prolog séquentiel est déjà satisfaisant pour les problèmes de cette classe.

- les "gros" problèmes, la limite arbitraire définie ici étant de 50.000 inférences, procurent de bons facteurs d'accélération. Les programmes de cette catégorie sont : *hamilton, mandel, queens1, queens2, TInA1, TInA2 et TInA3*. Ce résultat est très encourageant puisque le temps de réponse de ces programmes est (relativement) long séquentiellement et qu'il est rendu beaucoup plus supportable par le parallélisme. De plus l'ensemble du projet PEPSys a pour but de rendre possible la solution de problèmes de grande taille en programmation logique.

- les facteurs d'accélération des problèmes de taille "intermédiaire", comprise entre 20.000 et 50.000 inférences, sont proches de la moyenne des facteurs d'accélération de l'ensemble des programmes OU-parallèles.

Table 8.1: Performances des programmes OU-parallèles toutes solutions
Pour chaque programme, la première ligne donne le temps d'exécution en secondes, et la seconde ligne le facteur d'accélération obtenu par rapport à l'exécution séquentielle sur le système PEPSys. Les écarts types, voisins de 1 %, ont été omis pour plus de lisibilité. La ligne du bas contient les moyennes des facteurs d'accélération.

Programme	1	2	4	6	8
farmer *100	23,05	14,70	11,62	11,02	11,45
	1	1,57	1,99	2,09	2,02
hamilton	255,71	128,96	65,45	44,02	34,31
	1	1,98	3,91	5,81	7,45
houses *20	33,80	27,03	14,62	10,23	8,37
	1	1,25	2,31	3,30	4,04
mandel	36,58	19,89	9,98	6,70	5,12
	1	1,84	3,67	5,46	7,15
map	15,52	8,92	5,46	4,32	3,53
	1	1,74	2,84	3,59	4,41
queens1	57,72	29,03	15,54	10,53	7,97
	1	1,99	3,72	5,49	7,24
queens2	125,74	64,25	33,13	22,96	17,71
	1	1,96	3,80	5,48	7,10
salt	6,67	4,42	2,39	1,62	1,25
	1	1,51	2,79	4,12	5,37
tina1	70,25	37,92	20,21	14,04	11,07
	1	1,85	3,48	5,01	6,35
tina2	280,97	167,37	98,41	66,91	50,95
	1	1,68	2,86	4,20	5,51
tina3	97,67	51,65	26,72	18,20	14,41
	1	1,89	3,66	5,37	6,80
parse1 *20	11,09	6,85	3,99	3,72	3,69
	1	1,62	2,78	2,98	3,00
parse2 *20	42,31	22,61	14,01	11,29	10,97
	1	1,87	3,02	3,75	3,86
parse3 *20	9,58	6,24	4,00	3,60	3,50
	1	1,53	2,40	2,66	2,73
parse4 *5	39,26	21,24	12,06	8,69	8,02
	1	1,85	3,26	4,52	4,91
parse5	27,71	15,51	9,32	6,21	5,57
	1	1,79	2,98	4,47	5,03
db4 *10	14,15	9,06	5,51	3,94	3,10
	1	1,56	2,57	3,59	4,57
db5 *10	17,16	10,87	6,55	4,81	3,80
	1	1,58	2,62	3,57	4,52
Moyennes	1	1,73	3,03	4,19	5,11

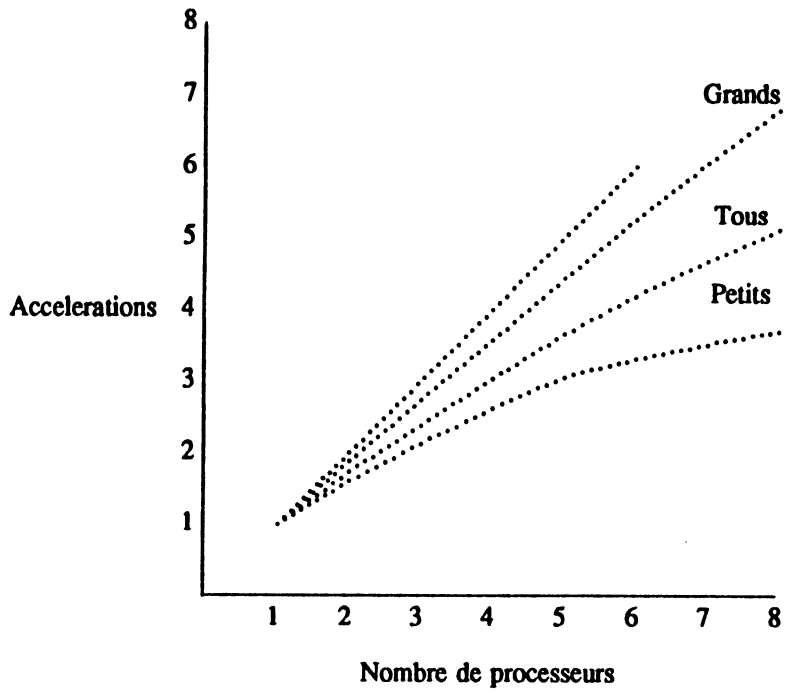

Figure 8.1: Facteurs d'accélération des programmes OU-parallèles toutes solutions

Cette figure résume les résultats de la table 8.1. Les trois différentes courbes représentent la moyenne des facteurs d'accélération sur l'ensemble des programmes OU-parallèles toutes solutions (Tous) telle qu'elle apparait dans la table 8.1, ainsi que les moyennes pour les petits et les grands problèmes.

La figure 8.1 résume ces résultats. On retrouve pour la programmation logique parallèle des résultats communément admis par ailleurs pour la programmation parallèle. La figure 8.1 est très similaire à la figure publiée en page 60 de [Quinn 87] pour illustrer l'influence de la taille du problème sur le facteur d'accélération.

Comparaison avec un système Prolog séquentiel efficace

Le système Prolog séquentiel avec lequel nous avons comparé l'implémentation de PEPSys est le système SICStus [Carlsson 88b]. Ce système est l'un des représentants de ce qu'on appelle "l'état de l'art" en matière d'implémentation Prolog. La technique utilisée est analogue à celle de PEPSys : il s'agit d'un émulateur de WAM écrit en langage C. Ce choix rend le système SICStus très facilement portable au détriment de l'efficacité, qui est deux fois moindre que celle de Quintus Prolog. En outre, contrairement à SICStus, le système Quintus Prolog n'a pas été porté sur Sequent.

La comparaison a été effectuée sur un sous-ensemble des programmes OU-parallèles toutes solutions pour lequel les résultats de SICStus Prolog étaient diponibles (voir table 8.2 et figure 8.2). Un grand nombre de ces programmes appartiennent

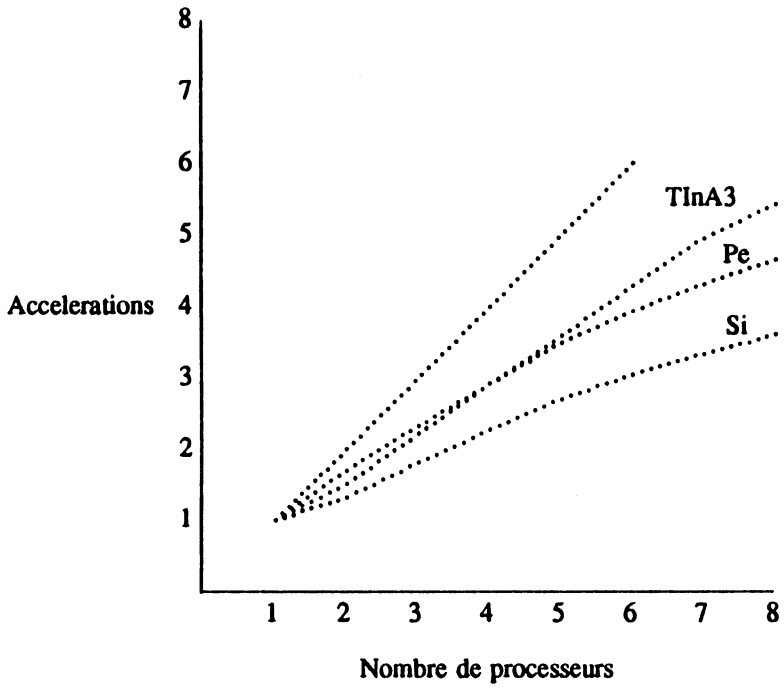

Figure 8.2: Facteurs d'accélération des programmes OU-parallèles toutes solutions relativement à un système Prolog efficace

Cette figure résume les résultats de la table 8.2. Les deux courbes basses représentent les moyennes des facteurs d'accélération relatifs à SICStus Prolog (Si)) et PEPSys (Pe) pour le sous-ensemble des programmes de la table 8.2. Comme il a été mentionné, il ne s'agit pas des mêmes programmes que ceux utilisés dans la figure 8.1. Les chiffres utilisés sont proches des "cas les plus défavorables" ainsi qu'on peut le constater en comparant les courbes de moyennes avec celle du facteur d'accélération obtenu pour *TInA3*.

à la catégorie des "petits" problèmes, pour lesquels nous avons déjà remarqué que PEPSys n'est pas bien adapté et procure de médiocres facteurs d'accélération. Les temps d'exécution par le système SICStus des programmes procurant de bons facteurs d'accélération, tels que *hamilton, mandel* et *TInA1*, n'étaient pas disponibles lors de la rédaction de cette thèse. En dépit de ce "handicap", il est à remarquer que l'exécution des programmes en parallèle par PEPSys procure toujours un facteur d'accélération. Il y a certes une exception, le programme *houses* qui est légèrement plus lent exécuté par deux processeurs que par le système SICStus. La raison est que ce programme crée un très grand nombre de *branch-points* lesquels, comme nous l'avons déjà constaté ne sont pas implémentés aussi efficacement qu'ils le devraient. Néanmoins des facteurs d'accélération apparaissent aussi à l'exécution de ce programme, dès que l'on dépasse deux processeurs.

Table 8.2: Comparaison entre SICStus et PEPSys exécuté en parallèle

La première colonne donne les résultats obtenus par SICStus Prolog sur Sequent Balance 8000 et les autres colonnes par PEPSys sur MX500. Pour ces dernières, la première ligne donne le temps d'exécution en secondes et la seconde ligne le facteur d'accélération relativement au temps d'exécution séquentiellement par SICStus. La dernière ligne contient la moyenne des facteurs d'accélération sur l'ensemble des programmes et est utilisée dans la figure 8.2.

Programme	SICStus	2	4	8
farmer	17,06	14,70	11,62	11,45
	1	1,16	1,47	1,49
houses	24,84	27,03	14,62	8,37
	1	0,92	1,70	2,97
queens1	37,25	29,03	15,54	7,97
	1	1,28	2,40	4,67
queens2	97,56	64,25	33,13	17,71
	1	1,52	2,94	5,51
tina3	78,1	51,65	26,72	14,41
	1	1,51	2,92	5,43
parse1	8,76	6,85	3,99	3,69
	1	1,28	2,20	2,37
parse2	32,97	22,61	14,01	10,97
	1	1,46	2,35	3,01
parse3	7,50	6,24	4,00	3,50
	1	1,20	1,88	2,14
parse4	30,48	21,24	12,06	8,02
	1	1,44	2,53	3,81
parse5	21,43	15,51	9,32	5,57
	1	1,38	2,30	3,89
db4	12,22	9,06	5,51	3,10
	1	1,35	2,22	3,95
db5	14,97	10,87	6,55	3,80
	1	1,38	2,28	3,94
Moyennes	1	1,32	2,27	3,60

Comparaison avec le système Aurora

Les projets PEPSys et GIGALIPS ont établi une collaboration informelle qui a permis entre autres le partage de programmes de tests et l'échange de résultats. Comme le remarque [Lusk et al. 88], les performances des systèmes PEPSys et Aurora, mesurées sur les mêmes matériels et en exécutant les mêmes programmes de test, sont étonnament similaires. La comparaison entre les tables 3.1 et 8.1 donne une idée de cette similarité. Celle-ci est la même pour l'ensemble des programmes OU-parallèles dont les résultats sont donnés dans la table 8.1. Ce résultat est assez remarquable si l'on considère les différences entre ces projets (voir la section 4.4.3) et entre les deux systèmes: PEPSys est une maquette développée rapidement pour valider les idées du projet alors que Aurora est basé sur SICStus Prolog qui est un système confirmé. Il semble également que beaucoup plus d'efforts aient été investis dans la mise au point d'ordonnanceurs sophistiqués pour Aurora [Brand 88a,Calderwood 89,Butler et al. 88]. Il serait intéressant de voir si cette similarité persiste lorsque que l'on approche des limites des systèmes en utilisant un plus grand nombre de processeurs sur des architectures extensibles. Il semble que les performances dépendront alors plus de la qualité des ordonnanceurs mis en œuvre que des modèles de gestion des variables logiques en parallélisme OU.

Comparaison avec les résultats du simulateur et de l'analyse abstraite

Les résultats du système parallèle PEPSys ont été comparé avec ceux des deux autres outils d'évaluation développés dans le cadre du projet [Chassin, Baron et al. 89]. Afin de valider le simulateur, il a été configuré comme une seule grappe, simulant ainsi l'architecture du MX500. Les résultats obtenus sont étonnament proches de ceux obtenus par l'implémentation. Lorsque le nombre de processeurs se rapproche de 30, limite de ce type d'architecture, les performances s'applatissent, même pour les plus "gros" problèmes. La principale raison est l'oisiveté des processeurs contre laquelle on s'efforce de mettre au point de bons ordonnanceurs.

Les résultats de l'implémentation ont également été comparés avec ceux de l'analyse abstraite pour essayer de découvrir des corrélations. Outre la petite taille de certains problèmes que met en évidence l'analyse abstraite, on a pu constater que les graphes d'analyse abstraite très hachés ne donnaient pas de bonnes performances. D'autre part, on a pu constater que le système parallèle permettait d'obtenir sur les petits programmes tels que *farmer* et *houses* près de la moitié du facteur d'accélération idéal prévu par l'analyse abstraite.

La comparaison est plus intéressante avec les programmes *TInAs* qui ont été développés en utilisant l'analyse abstraite. Selon [Ing 87a], si on adopte les hypothèses de l'analyse abstraite du parallélisme (voir la section 4.7), le programme *TInA2* devrait être légèrement plus efficace que *TInA3*, les deux battant plus nettement *TInA1*. Ceci provient de ce que la combinaison des parallélisme ET-OU dans *TInA2* évite des répétitions de calculs inutiles dans les branches OU-parallèles. Quant à l'algorithme *TInA1*, il présente moins de parallélisme que les deux autres.

Nous constatons que, exécuté en parallélisme OU seulement et sur un nombre limité de processeurs, l'ordre d'efficacité est inversé par rapport à l'analyse abstraite. La première raison réside dans les nombres d'inférences exécutées par chacunes des versions de *TInA* : respectivement 152.000, 259.000 et 222.000 pour *TInA1*, *TInA2* et *TInA3*. Le programme *TInA2* est non seulement moins rapide que les deux autres mais il procure un facteur d'accélération inférieur en parallèle. Comme nous le verrons dans la section 8.3.2, cela provient en grande partie du plus grand nombre de références non-locales manipulées dans le cas de *TInA2*. On peut expliquer ce résultat en remarquant que *TInA3* et surtout *TInA1* "simulent" le parallélisme ET en utilisant des *bagofs*[1]. L'effet du *bagof* que nous constatons ici est de recopier localement des variables non-locales. Le nombre de déréfécement non-locaux exécutés par les *TInA1*, *TInA2* et *TInA3* sur huit processeurs sont respectivement de 65.000, 662.000 et 49.000.

Les résultats de simulation, obtenus sur un plus grand nombre de processeurs, montrent que la croissance de *TInA2* reste régulière, le surcoût du aux variables non-locales restant limité (voir section 8.3.2). La croissance des facteurs d'accélération de *TInA3* en fonction du nombre de processeurs est plus rapide que celle de *TInA1*, les deux algorithmes étant équivalents pour une quinzaine de processeurs. Ceci s'explique par le grand nombre de *bagofs* utilisés par *TInA1*, chacun d'entre eux mettant en œuvre des mécanismes complexes en dépit de l'implémentation efficace qui en est faite dans PEPSys. Ce dernier résultat est somme toutes logique : sur un plus grand nombre de processeurs, l'algorithme *TInA3*, conçu pour tirer parti du parallélisme OU, serait le plus efficace des trois.

8.2.3 Programmes une-solution

Résultats

Les résultats des programmes OU-parallèles une solution sont donnés dans la table 8.3. A la différence des des programmes toutes solutions, on ne peut pas attendre de l'exécution de programmes une-solution sur un système idéal des facteurs d'accélération linéaires. Le facteur d'accélération d'un programme une-solution dépend de l'ordre suivant lequel l'arbre de réfutation est traversé. Cet ordre dépend lui-même essentiellement de deux paramètres qui sont la forme de l'arbre et l'ordonnanceur utilisé pour la recherche de travail.

La lecture des résultats appelle les remarques suivantes :

- les deux programmes spécifiquement une-solution, *master-mind* et *puzzle*, ont des résultats assez différents. Ainsi qu'il a été mentionné au préalable, le *master-mind* utilise le prédicat *oneof*, dont l'implémentation sur PEPSys n'est pas optimale (voir sections 6.4.6 et 6.7.2). Ceci explique que le programme de *master-mind* ne produise pratiquement aucun facteur d'accélération

[1] Il s'agit ici d'une simulation au niveau de l'algorithme et non d'une implémentation du parallélisme Et comme dans la section 8.2.4.

Table 8.3: Résultats des programmes OU-parallèles une-solution

Pour chacun des programmes, la première ligne donne le temps d'exécution en secondes et la seconde ligne le facteur d'accélération relativement au temps d'exécution séquentiel par PEPSys. La ligne *ham_one* donne le temps de calcul de la première solution du programme *hamilton*. Les lignes *queensj_i* correspondent au calcul de la première solution au problème des *i* reines par le programme*queensj*. La dernière ligne contient la moyenne des facteurs d'accélération sur l'ensemble des programmes, la moyenne de la colonne 6 étant faussée par l'absence de résultat pour *queensj_16*.

Programme	1	2	4	6	8
ham_one	2,67	2,36	2,37	1,55	1,40
	1	1,13	1,13	1,78	1,91
queens1_8	3,29	1,25	0,56	0,22	0,23
	1	2,63	5,90	14,79	14,18
queens2_8	6,57	2,39	0,97	0,36	0,35
	1	2,75	6,80	18,35	18,96
queens1_12	15,59	14,87	1,49	1,48	1,56
	1	1,05	10,50	10,57	9,99
queens1_16	902	89	46		44
	1	10	19,6		20,5
queens2_16	2466	217	127		129
	1	11,4	19,4		19,1
mm	60,30	60,48	49,72	11,91	7,66
	1	1,00	1,27	5,23	8,41
puzzle	30,07	4,68	4,68	4,68	4,76
	1	6,43	6,42	6,43	6,32
Moyennes	1	4,5	8,8	9,52	12,42

pour moins de 5 processeurs (les facteurs d'accélération pour 5,6 et 7 processeurs étant respectivement 3,9 5,2 et 7,3). En raison de son implémentation, les performances du *oneof* dépendent également beaucoup des conditions à l'exécution, ce qui explique les écarts types importants observés pour les résultats.

- le calcul de la première solution d'un problème qui est gros lorsque l'on calcule l'ensemble de ses solutions est parfois un petit problème. Dans ce cas la détermination de la branche la plus rapide peut dépendre des conditions à l'exécution et ne pas être déterministe, et donc occasionner des écarts types plus importants que pour les problèmes toutes solutions. Les écarts types redeviennent plus faibles lorsque le problème une solution est de grande taille.

Facteurs d'accélération superlinéaires?

Peut-on parler de facteurs d'accélération superlinéaires? Ce phénomène semble apparaitre dans les problèmes OU-parallèles une-solution (voir table 8.3). La raison est alors la présence d'une solution à proximité de la racine de l'arbre de réfutation, dans une branche explorée rapidement en parallèle, mais qui ne serait explorée que tardivement par un système séquentiel. Des facteurs d'accélération superlinéaires sont également susceptibles d'apparaitre dans les programmes combinant les parallélismes OU et ET. En effet le parallélisme permet d'éviter de recalculer une branche ET-parallèle autant de fois que l'autre branche ET-parallèle a de solutions, ce que fait un système Prolog séquentiel avec le mécanisme du retour-arrière.

Selon certains auteurs, les facteurs d'accélération superlinéaires n'existent pas : l'implémentation séquentielle la plus efficace explore tout de suite la bonne branche de l'arbre de recherche ou encore, elle ne fait aucun calcul inutile. Nous ne suivrons pas cette voie. En effet, la découverte de "l'algorithme optimal" dans un langage tel que Prolog impliquerait d'abandonner l'aspect déclaratif de ce langage pour se préoccuper étroitement du contrôle. De plus, la mise en œuvre de l'algorithme optimal au sens qui vient d'être donné ne semble pas toujours possible au codage, mais uniquement après un calcul du chemin optimal.

8.2.4 Programmes ET-parallèles

Ainsi que nous l'avons vu, seul le parallélisme ET déterministe du modèle de calcul PEPSys a été implémenté. Peu de temps a été consacré à le mettre au point et seuls deux programmes "jouets" ont servi à le mesurer. Les mesures de l'implémentation du parallélisme ET déterministe ont été comparées à celles de la compilation du parallélisme ET en parallélisme OU : le prédicat prédéfini *and_par* défini en section 3.4.7 a été rajouté au prédicats prédéfinis du système et on l'a substitué à l'opérateur d'indépendance dans les programmes ET-parallèles. Cette compilation du parallélisme ET en parallélisme OU a permis de tester la combinaison des deux

Table 8.4: Temps d'exécution des programmes ET-parallèles

Pour chacun des programmes, la première ligne donne le temps d'exécution en secondes et la seconde ligne le facteur d'accélération relatif au temps d'exécution séquentiellement. *fibor* et *qsortor* sont les versions OU-parallèles des programmes. L'avant-dernière ligne contient la moyenne des facteurs d'accélération sur l'ensemble des programmes, *TInA2* exclus. La dernière ligne donne les quelques mesures faites pour *TInA2*. Celles-ci ont été peu nombreuses en raison du manque d'efficacité de la méthode.

Programme	1	2	4	6	8
fib(20)	18,59	12,29	6,66	4,55	3,56
	1	1,50	2,77	4,05	5,18
fibor(20)	33,58	16,88	8,50	5,71	4,37
	1	1,99	3,95	5,88	7,68
qsort(500)	5,66	3,27	2,85	2,38	2,15
	1	1,74	2,00	2,40	2,64
qsortor(500)	13,84	7,70	5,08	4,08	2,45
	1	1,81	2,74	3,41	6,11
Moyennes	1	1,76	2,86	3,94	5,40
TInA2	426		198	133	
	1		2,1	3,2	

types de parallélisme dans le cas du programme *TInA2*. L'ensemble des résultats obtenu est donné dans la la figure 8.3 et la table 8.4.

Les temps d'exécution séquentiels des versions OU-parallèles des programmes précédents sont très supérieurs aux temps d'exécution des versions ET-parallèles pour deux raisons :

- les versions OU-parallèles exécutent plus d'inférences. L'exécution de *fib(20)* déroule 66.000 inférences, alors que celle de *fibor(20)* en déroule 99.000. Les nombres pour *qsort(500)* et *qsortor(500)* sont respectivement de 11.500 et 16.000.
- l'exécution d'un grand nombre de *bagofs* est consommatrice de temps calcul. Dans le cas des programmes déterministes, ces *bagofs* ne rassemblent chacun qu'une seule solution et pourraient être optimisés. Néanmoins ils sont nécessaires pour "recopier" la solution de chaque branche dans une zone de données dont la durée de vie dépasse celle des branches qui la produise.

Les facteurs d'accélération des versions OU-parallèles étant très supérieurs à ceux des versions ET-parallèles, les temps d'exécution des deux types de programmes sur huit processeurs sont très proches l'un de l'autre. Si cette tendance

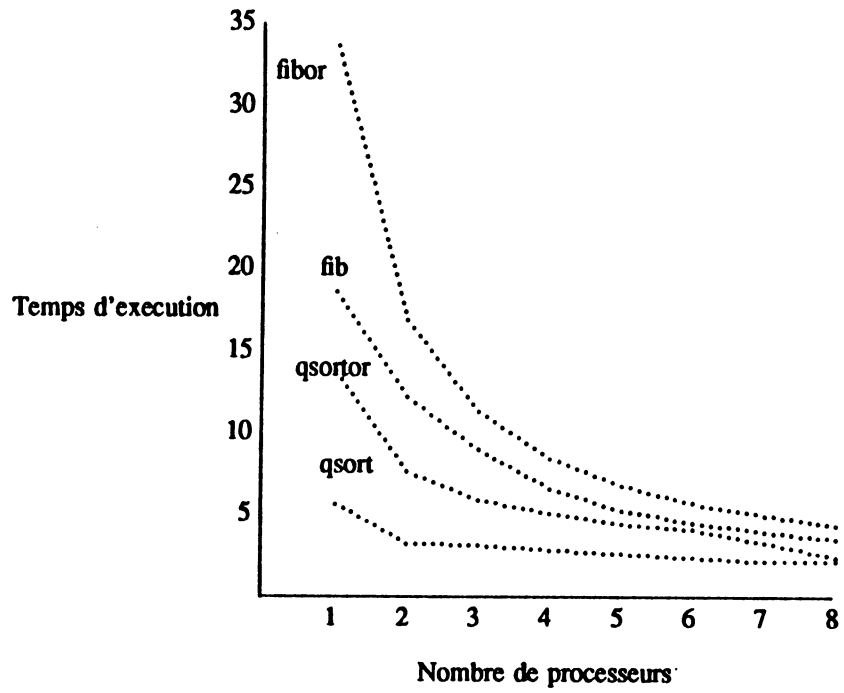

Figure 8.3: Comparaison entre les temps d'exécution des programmes ET-parallèles déterministes

Les courbes ci-dessus proviennent de la table 8.4. Les temps d'exécution sont mesurés en secondes.

se confirmait, la compilation du parallélisme ET en parallélisme OU se révèlerait plus efficace que l'implémentation directe du parallélisme ET, pour un plus grand nombre de processeurs. Ce genre de conclusion est cependant prématuré en raison du manque de représentativité des programmes mesurés et de l'état expérimental du système PEPSys. D'une part, il est possible d'optimiser notablement l'implémentation du parallélisme ET déterministe dans le modèle. D'autre part une implémentation réelle du *bagof* en utilisant un tas[2] en diminuerait vraisemblablement l'efficacité. La seule conclusion qu'il est possible de tirer au stade actuel de l'étude est que les deux méthodes d'exécution du parallélisme ET sont comparables et devront continuer à être évaluées dans les systèmes parallèles à venir.

8.3 Evaluation du modèle de calcul

L'une des caractéristiques essentielles du modèle de calcul PEPSys est la gestion des liaisons de variables dans un environnement OU-parallèle. Les mesures per-

[2]L'implémentation actuelle du *bagof* réserve une zone de taille fixe sur la pile Prolog. La taille par défaut est de 5000 éléments de pile. Cette taille doit être adaptée à chaque programme dans le cas de la compilation du parallélisme ET en parallélisme OU.

mettant de l'évaluer sont donc les plus importantes, après les mesures de performances. Parmi ces mesures, la plus importante concerne la longueur des chaines de références dans les *hash-windows*, non bornée dans le modèle. Le nombre de références non-locales est également d'une grande importance étant donné que chacune d'elle introduit un surcoût non négligeable par rapport à une référence locale.

8.3.1 Déréférencements non-locaux

Le déréférencement d'une variable non locale (voir section 5.3.3 pour plus de détails) implique le test systématique de la validité des liaisons et le changement éventuel de la "présentation" du résultat. Ces opérations supplémentaires par rapport au déréférencement local sont coûteuses en temps. Il est donc souhaitable qu'elles demeurent peu fréquentes et le pourcentage de déréférencements non locaux, par rapport au nombre total de déréférencements, a donc été mesuré. Cette mesure a permis d'évaluer le surcoût associé à ces opérations.

Pourcentage de déréférencements non locaux

Le pourcentage de déréférencements non locaux varie entre 10 % pour le programme *hamilton* et plus de 50 % pour les programmes *farmer* et *houses*. L'origine de ces différences tient aux programmes, - le programme *houses* par exemple utilise un grand nombre de variables et n'a que des *branch-points* et aucun point de choix - , ainsi qu'aux conditions d'exécution puisque les petits programmes génèrent des branches courtes, propices à l'utilisation de variables non locales. Une moyenne, calculée à partir de l'ensemble des résultats, des pourcentages de déréférencements non locaux est donnée à la figure 8.4. *TInA2* a été traité séparément étant donné que son nombre total de déréférencements non locaux (660.000 sur huit processeurs) est proche du total des déréférencements non locaux des autres programmes dans les mêmes conditions (1.110.000). La figure 8.4 indique également que les pourcentages se stabilisent lorsque le nombre de processeurs augmente. Ce résultat est encourageant mais devrait être confirmé sur un multiprocesseur comportant un plus grand nombre de processeurs.

Surcoût du aux déréférencements non locaux

Le surcoût du au déréférencement non local par rapport au déréférencement local, mesuré par la méthode décrite au paragraphe 7.1.2, est de 50 micro-secondes par déréférencement, soit le double de la durée de l'opération de déréférencement. Le pourcentage du surcoût du au déréférencement non local par rapport au temps d'exécution des programmes varie entre 15 % pour le programme *houses* et 1,7 % pour le programme *hamilton*, sur huit processeurs dans les deux cas. Ce pourcentage augmente avec le nombre de processeurs. En effet, comme le montre la figure 8.4, le pourcentage de déréférencements non locaux augmente ou reste con-

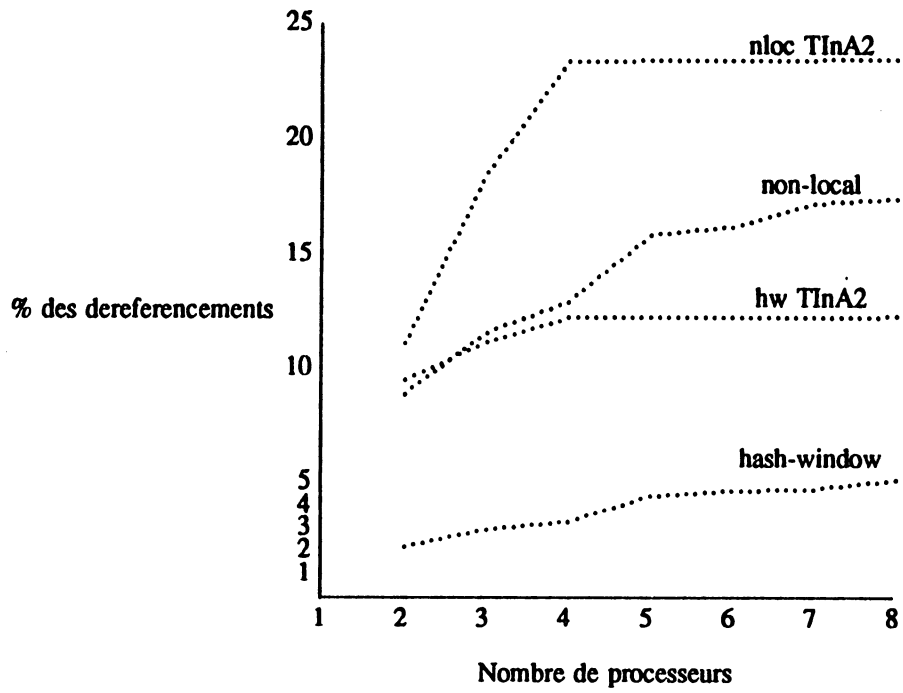

Figure 8.4: Pourcentages de déréférencements non locaux et dans les *hash-windows*

Cette figure montre les pourcentages de déréférencement non locaux et dans les *hash-windows*. Les courbes *nloc TInA2* et *non-local* sont les pourcentages de déréférencements non locaux, tandis que les courbes *hw TInA2* et *hash-window* correspondent aux pourcentages de déréférencements impliquant une recherche dans une *hash-window*, pour *TInA2* et pour la moyenne des autres programmes.

stant lorsque le nombre de processeurs augmente, alors que le temps d'exécution diminue.

8.3.2 Recherche dans les *hash-windows*

Les paramètres permettant d'évaluer le modèle de calcul PEPSys ont été mesurés ainsi qu'il a été fait mention dans le paragraphe 7.1.2.

Pourcentage de déréférencements dans les *hash-windows*

Lorsqu'une opération de déréférencement implique une recherche dans les *hash-windows*, le nombre de recherches dans les *hash-windows* est incrémenté de 1, quelque soit la longueur de la chaine visitée. On en déduit le pourcentage de recherches dans les *hash-windows* par rapport au nombre total de déréférencements. Ce pourcentage dépend du programme considéré, mais il demeure généralement faible, compris entre 1 % et 25 %. Les moyennes des valeurs des déréférencements dans les *hash-windows* sont données dans la figure 8.4. Une nouvelle fois, le programme *TInA2* est traité séparément car il effectue plus de recherches dans

les *hash-windows* - 346.000 pour huit processeurs - que l'ensemble des autres programmes- 329.000 dans les mêmes conditions. De même que les pourcentages de déréférencements non locaux, ces pourcentages augmentent avec le nombre de processeur pour atteindre rapidement un palier, résultat très encourageant qui devrait être confirmé par des mesures sur un multiprocesseur de plus grande taille.

Répartition des chaines d'accès aux *hash-windows*

La longueur des chaines de *hash-windows* est enregistrée dans des compteurs différents pour des longueurs de un à dix neuf et un compteur pour les longueurs de vingt et au delà. Une longueur de un signifie que la référence est trouvée dans la *hash-window* locale, une longueur de deux qu'elle se trouve dans la *hash-window* du processus père, etc.

Les résultats de ces mesures sont résumés dans la figure 8.5. Pour tous les programmes, la plupart des références sont trouvées dans la *hash-window* locale. Ce pourcentage décroit avec le nombre de processeurs et dépend du programme. Il varie entre 57 % sur huit processeurs pour le programme *houses* et 99 % dans les mêmes conditions pour les programmes *queens*. Dans les pires des cas, - programmes *farmer* et *houses* -, plus de 95 % des déréférencements dans les *hash-windows* impliquent moins de trois *hash-windows*. Ce pourcentage atteint 99 % pour moins de cinq *hash-windows*.

Longueur maximale de la chaine de *hash-windows*

La longueur de la plus longue chaine de *hash-windows* explorée se situe aux environs de cinq pour la majorité des programmes. La plus grande valeur obtenue à ce jour est de vingt pour les programmes *TInAs*, cette situation se produisant dans moins de 0,01 % des déréférencements dans les *hash-windows*, c'est à dire 0,0003 % du total des opérations de déréférencement.

Pourcentage de *hash-windows* utilisées. Taux de remplissage

Le pourcentage de *hash-windows* utilisées dépend du programme mais demeure le plus souvent supérieur à 90 %, la valeur la plus basse étant de 79 % pour la requête *db5* de *Chat80*.[3]

Le taux de remplissage des *hash-windows* reste par contre très faible, moins de 5 % des positions étant utilisées en moyenne, même si l'on cache les références des *hash-windows* non-locales.[4] Il serait donc souhaitable de n'allouer que de petites *hash-windows* d'une taille à définir entre 4 et 16 positions, le débordement de capacité étant traité par la création d'une nouvelle *hash-window* de taille supérieure, chainée à la précédente et allouée au même processus. La mise en œuvre de cette

[3] On trouve des pourcentages inférieurs pour les programmes ET-parallèles mais l'utilisation de *hash-windows* dans ce cas n'est justifiée que pour des raisons de simplicité d'implémentation.

[4] Le programme *Baskett* fait exception à cette règle puisque chacune des positions de *hash-window* est utilisée en moyenne plus de vingt fois durant l'exécution du programme.

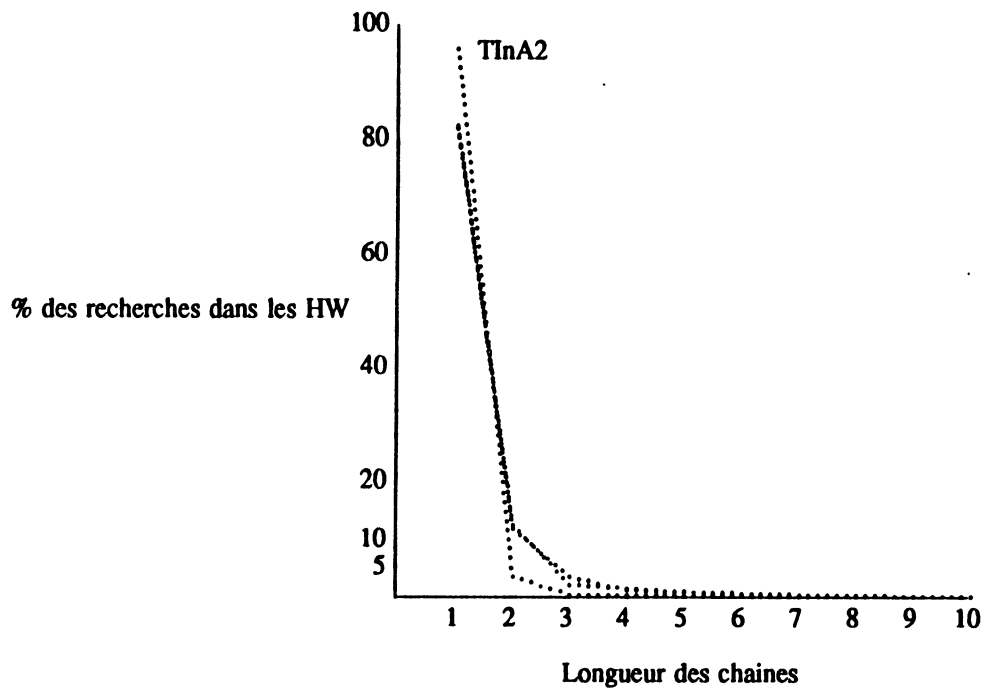

Figure 8.5: Pourcentage des longueurs des chaines de *hash-windows*

Cette figure donne les pourcentages des opérations de déréférencement dans les *hash-windows* de longueur 1 à 10 par rapport à l'ensemble des opérations de recherche dans les *hash-windows*. Les deux courbes qui sont presque partout superposées donnent les moyennes de tous les programmes, sauf *TInA2*, pour quatre et huit processeurs. L'autre courbe qui intersecte les précédentes et converge plus rapidement donne les pourcentages pour *TInA2* sur huit processeurs. *TInA2* est traité séparément car le nombre de ses opérations de recherche dans les *hash-windows*, - 346.000 sur huit processeurs -, est à lui seul supérieur à la somme des opérations correspondantes pour les autres programmes, - 329.000 sur huit processeurs -.

solution devrait se faire soigneusement afin d'éviter de trop ralonger les chaines de références et de trop augmenter le nombre de collisions (très faible avec des tailles de *hash-windows* de 128 éléments). Il s'agit là d'une illustration du compromis classique entre espace utilisé et efficacité.

Surcoût provenant des recherches dans les *hash-windows*

En utilisant les résultats du paragraphe 7.1.2, et le nombre de recherches dans les chaines de *hash-windows* de longueurs un à vingt, on calcule une estimation du temps passé dans les opérations de recherche dans les *hash-windows*. Le rapport entre cette valeur et le temps d'exécution du programme augmente avec le nombre de processeurs. Il varie autour de 1 % pour la plupart des programmes. Il est habituellement plus grand pour les petits programmes. Sa valeur maximale est d'environ 10 % pour les programmes *houses* et *TInA2*. Il est habituellement plus

grand pour les petits programmes.

Influence du *caching* dans les *hash-windows*

L'optimisation consistant à "cacher" dans la *hash-window* locale les références valides trouvées dans les *hash-windows* distantes accroit très notablement la localité de référence dans les *hash-windows*, particulièrement pour le programme *TInA2* pour lequel celle-ci était médiocre. Pour ce dernier programme, les résultats sont particulièrement spectaculaires. Le temps d'exécution sur sept processeurs a diminué de 100 à 73 secondes. Les surcoûts dus à la recherche dans les *hash-windows* et au déréférencement non-local, mesurés suivant les techniques du paragraphe 7.1.2, sont passées respectivement de 37 à 7 secondes et de 11 à 7 secondes. Le nombre d'opérations de déréférencement impliquant une recherche dans les *hash-windows* a lui diminué de 910.000 à 360.000. Par contre le nombre de liaisons profondes dans les *hash-windows* a légèrement augmenté de 580.000 à 615.000. Cette optimisation diminue également la longueur maximale des chaines de *hash-windows* pour l'ensemble des programmes de mesure. La plus longue chaîne enregistrée a une longueur de 17 unités contre 70 au préalable. Dans tous les cas, même pour les programmes tels que *queens* qui ne font que très peu de liaisons profondes, il est bénéfique de cacher dans la *hash-window* locale les liaisons faites dans une *hash-window* d'un processus ancêtre.

8.3.3 Evaluation du modèle de calcul

Les mesures précédentes valident les choix faits pour le modèle de calcul. Le programmes PEPSys testés présentent une bonne localité de référence et les opérations de déréférencement non local ou d'accès à des *hash-windows* non locales demeurent peu fréquentes. La longueur des chaines de références dans les *hash-windows* demeure très limitée, la plupart des accès restant locaux. Le modèle de calcul ne grève donc pas les performances du système PEPSys. Le nombre de déréférencements non locaux étant malgré tout important, il n'est pas évident que les performances du système resteraient bonnes sur une machine à mémoire distribuée. Les résultats préliminaires obtenus en simulant le modèle PEPSys sur ce type d'architecture sont peu encourageants [Baron, Chassin et al. 88]. Certaines optimisations permettraient sans doute d'améliorer les performances. On pourrait par exemple cacher dans la *hash-window* locale toutes les références non locales, et pas seulement celles des *hash-windows* non locales ainsi qu'il est fait dans l'implémentation. On pourrait également recopier dans un processus nouvellement créé le dernier environnement du processus père.

8.4 Activité des processeurs

8.4.1 Nombre de processus

Le nombre de processus créés durant l'exécution des programmes est très variable selon les programmes. Le minimum est atteint avec le programme *queens1* pour lequel environ[5] 50 processus sont créés lorsqu'il est exécuté sur huit processeurs. Le maximum est obtenu pour le test *parse2* de *Chat80* avec 3900 processus.

8.4.2 Granularité des processus

Bien que le langage favorise les processus de granularité importante, les mesures de durées de vie des processus montrent que, même pour les programmes obtenant de bons résultats, la majorité des processus a une durée de vie très courte. Ainsi le programme *hamilton*, qui procure un excellent facteur d'accélération, crée plus de 700 processus durant son exécution sur huit processeurs. Parmi ceux-ci plus de 600 ne dépassent pas dix inférences et la plus grande partie du calcul est faite par douze d'entre eux. Le nombre de processus créés est encore plus grand pour les programmes obtenant des résultats médiocres. Ainsi le programme *farmer*, tout en exécutant nettement moins d'inférences, crée plus de 4600 processus dont une écrasante majorité exécute moins de cinq inférences.

L'exécution des programmes présentant de bons facteurs d'accélération crée quelques processus de granularité importante (plus de dix milles inférences) et on constate que ces processus en petit nombre effectuent la plus grande partie du calcul. Ce phénomène n'apparait pas pour les programmes montrant des performances médiocres pour lesquels on trouve, en plus des processus de durées de vie faibles, un nombre relativement important de processus de durée de vie moyenne (quelques centaines d'inférences).

L'ordonnanceur utilisé influence le nombre de processus créés. Le passage de l'ordonnanceur originel à l'ordonnanceur "Robin des Bois" a eu des effets inverses sur les programmes longs et les programmes courts. Le nombre de processus créés par les programmes longs a diminué (passant de 1500 environ à un peu plus de 700 pour *hamilton* sur huit processeurs par exemple), alors qu'il a augmenté pour les programmes courts (de moins de 4000 à plus de 4600 pour *farmer* sur 8 processeurs). Deux raisons différentes expliquent probablement ce phénomène. D'une part, "Robin des Bois" permet aux processeurs exécutant les gros programmes de sélectionner du travail à partir des plus longues branches actives, ce qui augmente la granularité moyenne des processus créés par les gros programmes. D'autre part l'abandon de la stratégie "aider ou attendre" réduit l'oisiveté des petits programmes en permettant la création d'un plus grand nombre de processus. Ceux-ci restent de petite taille puisque, pour ces programmes, presque tout le parallélisme possible est déjà utilisé.

[5] Cette mesure n'est pas déterministe et présente habituellement un écart type important.

No. de Procs.	2		4		8	
Programme	Pourc.	Acc.	Pourc.	Acc.	Pourc.	Acc.
Hamilton	0,2 %	1,98	0,5 %	3,91	1,1 %	7,45
TInA3	0,1 %	1,89	0,9 %	3,66	1,8 %	6,80
Farmer	6,7 %	1,57	97,5 %	1,99	97,5 %	2,02
Houses	0,3 %	1,11	10,6 %	1,7	31 %	2,93
parse3	25,5 %	1,53	72,3 %	2,40	98,9 %	2,73
db5	0,9 %	1,58	4,3 %	2,62	8,4 %	4,52

Table 8.5: Pourcentage du calcul effectué par les processus courts

La corrélation entre le pourcentage et le facteur d'accélération apparait nettement pour un sous ensemble représentatif des programmes OU-parallèles. Il semble qu'on puisse attendre de meilleures performances de *db5*, pénalisé par le surcoût du à la gestion des *branch-points*.

Les mesures de granularité des processus confirment certaines hypothèses faites dans le projet PEPSys. Tout d'abord, la nécessité de donner au programmeur les moyens d'exprimer les sources de parallélisme de ses algorithmes, faute de quoi le nombre de processus de faible taille serait encore plus important, accroissant le surcoût du à la création et à la terminaison des processus. Une autre hypothèse se trouve confirmée : celle de différer le surcoût associé à la gestion des variables pour le parallélisme OU, contrairement à d'autres modèles qui "paient" ce surcoût à la création des processus (voir section 4.4.3).

8.4.3 Pourcentage du calcul effectué par les processus courts

On constate une forte corrélation entre la durée de vie moyenne des processus créés lors de l'exécution parallèle d'un programme et les performances de ce programme. Cependant, ce genre de moyenne a peu de sens en raison des disparités entre les durées de vie des processus mentionnées ci-dessus. Une autre corrélation, ne posant pas ces problémes, a été établie entre le pourcentage de l'exécution des programmes effectué par les processus de courte durée de vie et les performances des programmes. Un seuil arbitraire a été fixé à la valeur de 100 inférences et les nombres d'inférences effectué par les processus d'une durée de vie inférieure au seuil ont été sommés. Cette nouvelle corrélation se révèle plus forte que la précédente, ce qui permet de constater que les nombreux processus créés par les programmes ayant les meilleurs facteurs d'accélération contribuent beaucoup moins au calcul des solutions que les quelques processus de grande taille. Les principaux résultats montrant cette corrélation sont donnés dans la table 8.5.

No. de Procs.	2		4		8	
Programme	Pourc.	Acc.	Pourc.	Acc.	Pourc.	Acc.
Hamilton	0,5 %	1,98	0,5 %	3,91	1,0 %	7,45
TInA3	0,2 %	1,89	0,9 %	3,66	2,2 %	6,80
Farmer	10,6 %	1,57	30 %	1,99	56 %	2,02
Houses	1,2 %	1,11	6,1 %	1,7	15,6 %	2,93
parse3	7,9 %	1,53	22,4 %	2,40	50,5 %	2,73
db5	1,4 %	1,58	4,2 %	2,62	9,3 %	4,52

Table 8.6: Pourcentage d'oisiveté durant l'exécution

L'oisiveté est la principale source de surcoût pour les petits programmes, ce qui explique la corrélation importante entre le pourcentage d'oisiveté et le facteur d'accélération pour ce sous ensemble représentatif des programmes OU-parallèles.

8.4.4 Oisiveté des processeurs

Nous avons défini l'**oisiveté** comme le temps passé par les processeurs à chercher du travail, créer et initialiser des processus. L'oisiveté est déduite de la mesure du temps d'activité des processus en effectuant la différence entre la durée d'exécution des programmes et la somme des durées d'activité des processus. En raison de l'imprécision de la mesure du temps dans le système UNIX, la mesure de l'oisiveté est peu précise lorsque sa valeur est faible. On peut en attendre plutôt une estimation d'un ordre de grandeur. Cela n'est pas le cas dans les programmes pour lesquels elle représente une source importante de surcoût.

La table 8.6 donne les mesures d'oisiveté pour le même sous ensemble de programmes OU-parallèles que ceux utilisés dans la table 8.5. Les résultats dépendent encore une fois beaucoup des programmes et du nombre de processeurs utilisés. L'oisiveté est la source principale de surcoût pour les petits programmes qui présentent de mauvaises performances et pour lesquels le pourcentage d'oisiveté croît très rapidement avec le nombre de processeurs utilisés. La présence d'une oisiveté importante ne correspond pas nécessairement à l'utilisation de toutes les possibilités de parallélisation (voir section 8.5.1). En effet, l'exécution peut se diviser en phases durant lesquelles du travail potentiellement parallèle est exécuté séquentiellement, faute de processeur disponible, et en phases où les processeurs actifs n'ont plus de travail à offrir aux processeurs oisifs.

8.5 Autres mesures liées au parallélisme

8.5.1 Nœuds parallèles utilisés séquentiellement

Les pourcentages mesurés ici sont les pourcentages de nœuds, - *branch-points* et *fork-points* -, dont le travail offert est utilisé séquentiellement. Pour les programmes OU-parallèles dont chaque *branch-point* n'offre qu'une possibilité de travail ainsi que pour les programmes ET-parallèles déterministes, ce pourcentage représente le pourcentage de travail parallèle potentiel inutilisé. Pour les autres programmes OU-parallèles, *farmer*, *TInAs* et *Chat80*, le pourcentage de possibilités inexploitées est supérieur à cette mesure.

Très logiquement, ce pourcentage décroît avec le nombre de processeurs utilisés et croît avec la taille du programme. Néanmoins, dans presque tous les cas il demeure très important, la seule exception étant le programme *farmer* pour lequel chacun des *branch-points* (1700) créés durant l'exécution du programme sur huit processeurs est utilisé pour générer du parallélisme[6]. Pour les autres programmes, le pourcentage de nœuds parallèles utilisés séquentiellement reste toujours très élevé, les pourcentages variant entre 48,66 % pour *parse1* et 99,16 % pour *TInA2*. Dans la plupart des cas, les chiffres sont supérieurs à 90 %.

8.5.2 Surcoût provoqué par l'utilisation séquentielle des nœuds parallèles

La surcoût associé à l'utilisation séquentielle d'un nœud parallèle a été mesurée suivant la technique donnée au paragraphe 7.1.4. Elle est de 400 micro-secondes pour les *branch-points* n'offrant qu'une seule alternative ainsi que pour les *fork-points*. Pour les autres *branch-points*, elle dépend du nombre d'alternatives offertes, variant de 700 micro-secondes pour ceux de *farmer* qui en offrent trois, à 1200 micro-secondes en moyenne pour *TInA* dont les *branch-points* sont de taille variées, le nombre d'alternatives offertes allant de une à plusieurs dizaines. La mesure n'a pas été faite pour *Chat80* mais il est vraisemblable que le surcoût moyen par *branch-point* y est élevé, en raison de la taille importante de ceux-ci.

Etant donnée la disparité des mesures, la valeur de 400 micro-secondes par nœud utilisé séquentiellement a été retenue dans les tableaux de résultats de [Chassin 89] comme surcoût du à l'exécution séquentielle d'un nœud parallèle. L'estimation obtenue est réaliste pour un grand nombre de programmes alors que pour les *TInAs* et *Chat80* il s'agit d'un minorant. Les valeurs obtenues dépendent bien sûr du nombre de programmes. Les pourcentages du temps de calcul des programmes OU-parallèles sont souvent faibles mais pas négligeables (3 % pour *queens1*, 9 % pour *TInA1* sur huit processeurs). Pour certains petits programmes, le surcoût du à ce facteur est supérieure à toutes les autres (22 % dans

[6]Cela ne veut pas dire que toutes les possibilités de création de processus ont été utilisées. Chaque *branch-point* de *farmer* offrant trois alternatives, 4600 des 5100 processus possibles sont créés, sur huit processeurs.

le cas de *map*, sans doute très supérieure au minorant de 13 % pour *db5* sur huit processeurs).

Les mesures des surcoûts liés à l'utilisation séquentielle des nœuds parallèles confirment la nécessité d'améliorer l'implémentation du parallélisme OU en utilisant un jeu d'instructions et des structures de données plus proches des instructions *try* et des points de choix qui sont utilisés séquentiellement (voir section 5.2.2). Le manque de mesures faites sur les programmes ET parallèles ne permet pas encore d'estimer l'importance du surcoût du à l'utilisation séquentielle des *fork-points* (17 % pour *fib(20)* mais seulement 1 % pour *quicksort(500)* sur huit processeurs).

Remarque : les instructions *test_or_par* et *test_and_par* ont été utilisées pour tenter de limiter le nombre de nœuds parallèles créés inutilement, sans résultat probant, les limitations valables pour un programme produisant des effets négatifs pour les autres.

8.5.3 Valeur maximale de l'*OBL*

La mesure de la valeur maximale atteinte par l'*OBL* varie beaucoup suivant le programme considéré. Sur huit processeurs le maximum varie entre 2 pour le programme *mandel* et 95 pour *hamilton*. L'espace de huit bits, réservé dans chaque élément de pile pour enregistrer l'*OBL*, suffit donc largement. En considérant que la valeur maximale atteinte dans la plupart des programmes est inférieure à 30 et que la plupart des *branch-points* sont inutilisés, il semble que 5 bits suffiraient à encoder l'*OBL*.[7] A supposer que la valeur limite soit atteinte, deux politiques sont possibles :

- ne plus créer de *branch-points* mais seulement des points de choix. C'est la solution qui a été implémentée mais ce cas ne s'est encore pas produit car la limite actuelle de 255 n'a pas encore été atteinte. Le risque, surtout avec une valeur maximale admissible plus faible pour l'*OBL* , est de ne pas offrir assez de travail pour occuper tous les processeurs.
- créer un nouveau processus et une nouvelle *hash-window* liée à la précédente et réinitialiser la valeur de l'*OBL* à 0. L'inconvénient est alors d'augmenter le nombre de références non locales et d'allonger les chaines de *hash-windows*.

8.5.4 Compétition

Le premier résultat de ces mesures est que la compétition est très peu fréquente : plus de 99 % des verrous peuvent être réservés au premier accès. Quant au temps total passé à attendre la libération des verrous, il demeure extrêmement faible, le plus souvent de l'ordre de un pour mille, atteignant parfois un pour cent. La

[7] Ce genre de considération devrait être fait pour porter l'implémentation sur une machine à 32 bits d'adresse car l'étiquette devrait alors être placée sur le second mot des éléments de pile. Il faudrait alors "rogner" sur les tailles des autres champs dont l'*OBL*.

compétition est donc de loin la plus faible des sources de surcoût mesurés lors de l'exécution parallèle.

Le temps passé à acquérir et libérer les verrous n'a pas été mesuré mais il est certainement très supérieur à la mesure faite ici puisque la durée d'une prise et libération d'un verrou est de 50 micro-secondes, à comparer aux 8 micro-secondes de chaque boucle d'attente. Ce temps est susceptible d'incorporer un surcoût inutile si l'ordonnanceur réserve des verrous inutilement, ce qui reste à établir. De plus les perspectives de gains d'efficacité en réduisant l'utilisation des verrous ne sont pas évidentes puisque le temps de prise et relâchement d'un verrou a été réduit à 2 micro-secondes sur les Sequent Symmetry (successeurs du Balance).

8.6 Consommation mémoire

Tous les choix faits à l'implémentation du système PEPSys ont eu pour but de maximiser l'efficacité du système parallèle, le plus souvent sans se soucier de l'occupation mémoire. En dépit de ces choix, le manque de mémoire n'a jamais posé de problème. Environ 20 % des 16 méga-octets de mémoire du MX500 ont été réservés pour les piles locales et globales et même lorsque l'ordonnanceur a permis la création de trous noirs, aucun problème n'est apparu. Durant l'exécution séquentielle des programmes OU-parallèles toutes solutions, l'une des sources importantes de consommation mémoire est l'espace de 40 Kilo-octets réservé pour stocker les solutions collectées par le *bagof* de plus haut niveau. Contrairement à [Chassin 89], cet espace a été soustrait aux mesures d'occupation mémoire, afin de mieux mettre en évidence la surconsommation due au parallélisme. Les résultats des mesures qui en résultent sont résumés dans la figure 8.6 et indiquent une surconsommation mémoire très supérieure à [Chassin 89]. L'importante surconsommation mémoire due au parallélisme est vraisemblablement surévaluée puisque l'on calcule la somme des maximums pour chaque processeur et que ceux-ci ne sont vraisemblablement pas tous atteints au même moment (voir section 7.1.5).

Pour tenter d'évaluer les sources principales de surconsommation en exécution parallèle, les influences des *hash-windows* et des trous noirs sur l'espace de pile consommé ont été evaluées.

Le système actuel utilise l'ordonnanceur "Robin des Bois" qui créée des trous noirs sur la pile. Une autre campagne de mesures a été faite pour les mêmes programmes, en utilisant l'ordonnanceur originel dont la stratégie "aider ou attendre" évite la création de trous noirs sur la pile. Bien que la différence entre les rapports de surconsommation mémoire soit importante, les pentes des deux courbes restent parallèles.

Le nombre d'éléments de chaque *hash-window* de l'implémentation a été progressivement porté de 16 à 32 puis à 128 éléments, soit une taille de 2 Koctets, afin de pouvoir exécuter l'ensemble des programmes disponibles. Des mesures ont été faites lors du passage de 32 a 128 éléments sur les onze programmes OU-parallèles toutes solutions qui étaient alors disponibles. La figure 8.7 compare les moyennes

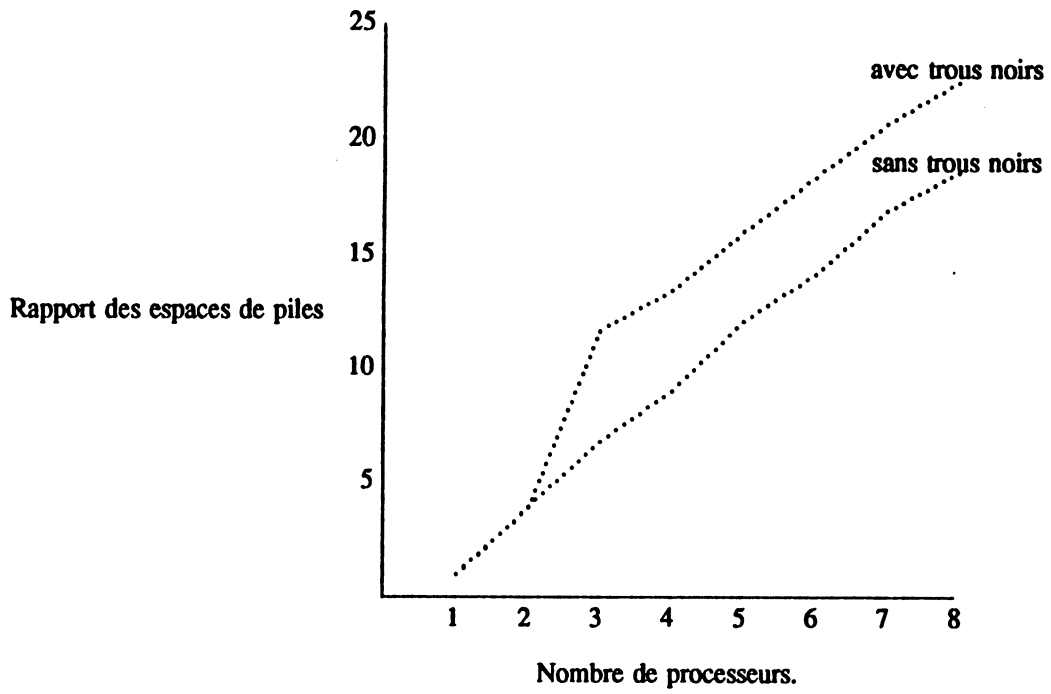

Figure 8.6: Rapports entre l'espace de pile consommé en parallèle et séquentiellement. Influence des trous noirs

La courbe supérieure représente la moyenne des rapports entre l'espace de pile consommé en parallèle et séquentiellement, faite pour l'ensemble des programmes utilisés pour les mesures, exception faite des *queens(16)*. L'autre courbe représente la même moyenne, faite pour les mêmes programmes en utilisant le scheduler originel, donc sans créer de trous noirs dans les piles.

des rapports d'espace de pile occupé, faites pour ces onze programmes, pour les deux tailles de *hash-windows* de 32 et de 128 éléments. Cette figure montre qu'il est possible de limiter nettement la surconsommation d'espace de piles en limitant la taille des *hash-windows*. Cette taille pourrait être nettement inférieure à 32 éléments étant donné le faible taux d'occupation des *hash-windows* (voir paragraphe 8.3.2). Cependant, l'incidence des réductions de taille de *hash-windows* sur la consommation mémoire des piles, pour des tailles de *hash-windows* inférieures à 32 éléments, ne serait peut-être pas aussi spectaculaire.

D'autres sources de consommation mémoire potentiellement importantes ne sont pas estimées ici. Il s'agit de la place occupée par les *branch-points* des gros prédicats parallèles dans les programmes *TInAs* et *Chat80* dont la taille pourrait être réduite. Leur influence a été surestimée dans [Chassin 89] où l'on a négligé l'importance des espaces de taille fixes réservés pour les *bagof*. Ces espaces, en augmentant artificiellement de manière importante la consommation mémoire en exécution séquentielle, réduisent le pourcentage de surconsommation en parallèle.

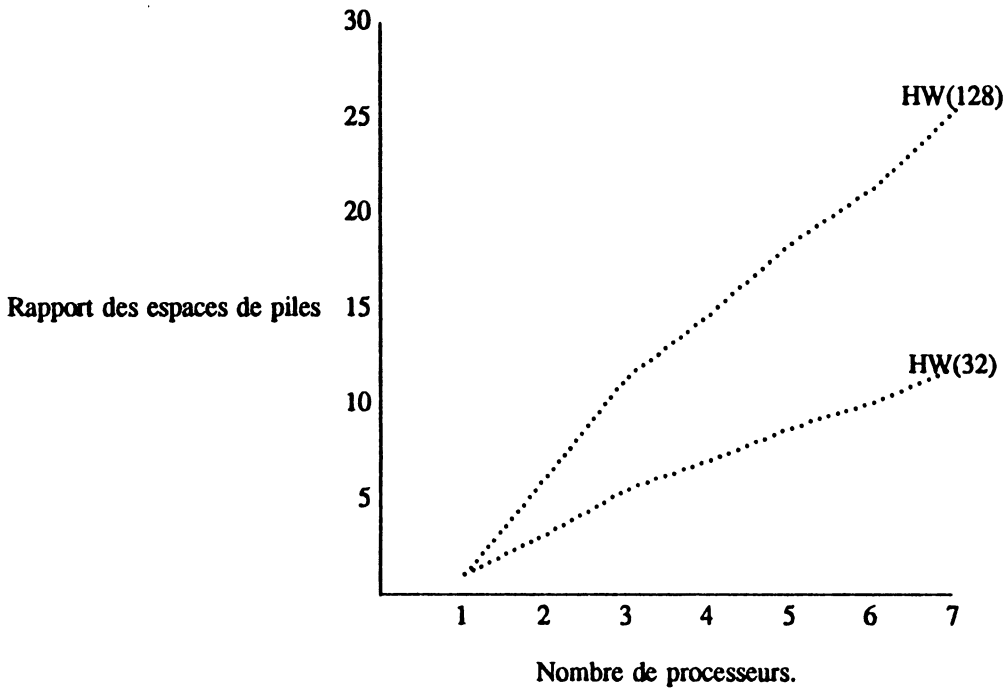

Figure 8.7: Influence de la taille des *hash-windows* sur la consommation mémoire

Les courbes représentent les moyennes des surconsommation mémoire dans les piles pour des sous-ensembles des programmes OU-parallèles toutes solutions, en utilisant des *hash-windows* de 128 éléments (HW(128)) dans un cas et de 32 éléments (HW(32)) dans l'autre. Ces deux séries de mesures ont été faites en utilisant le scheduler originel.

Comme nous l'avons signalé, la consommation mémoire n'a pas posé de problème, même lors de l'exécution de programmes de très grande taille (plusieurs heures de temps calcul) par la version parallèle de CHIP [Dincbas et al. 88] implémentée à partir du système PEPSys (Résultats non publiés dans [Van Hentenryck 89]). Sur un système "réel" disposant d'un mécanisme permettant de gérer le débordement de *hash-window* et utilisant des *branch-points* de plus petite taille, la consommation mémoire pourrait être notablement réduite. Si malgré tout l'occupation mémoire posait un problème il faudrait alors envisager un ramasse-miettes complet en parallèle.

8.7 Conclusion du chapitre

Un grand nombre de mesures ont permis d'évaluer le système PEPSys. Le principal résultat est bien sûr l'efficacité du système qui procure de réels facteurs d'accélération comparé à un système Prolog efficace. Certes, la courbe des performances moyennes s'aplatit lorsque l'on augmente le nombre de processeurs, mais on constate que ce phénomène est principalement dû aux "petits" programmes, dont

le temps de réponse séquentiellement est déjà satisfaisant. Les "gros" programmes ne présentent pas un facteur d'accélération idéal non plus mais les origines des écarts avec un système idéal ont été identifiées et évaluées. Elles pourraient être réduites dans une version ultérieure du système.

Le second résultat important des campagnes de mesure est la viabilité du modèle défini par PEPSys pour la gestion des résolvantes en parallélisme OU. Les mesures de déréférencement non local et de recherche dans les *hash-windows* ont montré que ces opérations étaient peu fréquentes, même lorsqu'on augmente le nombre de processeurs. La longueur des chaines de références dans les *hash-windows*, potentiellement illimitée, est en moyenne très courte, la plupart des références étant trouvées dans la *hash-window* locale.

Pour confirmer ces premiers résultats encourageants, il serait nécessaire d'améliorer le prototype de l'implémentation PEPSys et de pouvoir l'exécuter sur un plus grand nombre de processeurs.

Chapitre 9

Conclusion

9.1 Bilan

La recherche présentée dans cette thèse s'est déroulée dans le cadre du projet PEPSys. Le projet PEPSys intègre l'ensemble des aspects liés à l'exploitation efficace du parallélisme en programmation logique, à savoir un langage de programmation, un modèle de calcul et des applications. L'approche suivie a en outre été validée par une implémentation parallèle sur un multiprocesseur ainsi que par une simulation d'une architecture plus extensible. Le langage PEPSys, très proche de Prolog, permet au programmeur d'exprimer le parallélisme de son application. Le modèle de calcul incorpore les parallélismes OU et ET ainsi que la combinaison des deux avec l'exécution séquentielle et le retour-arrière. Ce modèle résoud en outre le problème de la gestion des variables logiques dans un environnement OU-parallèle, en utilisant une étiquette pour dater toutes les liaisons, ainsi que des *hash-windows* pour enregistrer les liaisons profondes. L'écriture d'applications parallèles a été menée simultanément pour valider le langage et fournir des programmes parallèles pour tester le simulateur et l'implémentation.

Le travail qui est décrit dans cette thèse est l'implémentation du langage PEPSys sur un multiprocesseur, conformément au modèle de calcul ainsi que l'évaluation du système parallèle réalisé. Le but de ce travail était de valider les choix du projet, d'acquérir une expérience de mise en œuvre effective du paralléllisme et enfin d'obtenir un système permettant d'exécuter de grandes applications et de collecter des mesures. L'architecture du multiprocesseur choisi pour réaliser cette implémentation est celle dont la mise en œuvre pose le moins de problèmes, à savoir bus et mémoire communs. Bien que ce type d'architecture soit généralement considéré comme peu extensible, c'est le seul à offrir à l'heure actuelle des multiprocesseurs facilement utilisables. L'implémentation parallèle utilise les techniques les plus efficaces développées pour l'implémentation séquentielle de Prolog. La *WAM* a été adaptée au modèle et au langage PEPSys et un compilateur du langage dans cette nouvelle machine abstraite réalisé. Un interprète de cette machine abstraite a ensuite été développé en langage C. Il est complété par un ordonnanceur également écrit en langage C. Seuls les parallélismes OU et ET

déterministe ont pu être réalisés dans la version existante du système PEPSys parallèle.

La thèse décrit également l'expérimentation du système parallèle. L'efficacité de ce système sur un seul processeur est proche de celle des meilleurs systèmes Prolog implémentés selon les mêmes techniques. L'exécution du système PEPSys en parallèle procure des gains de performances parfois importants, relativement à l'exécution séquentielle de PEPSys mais aussi des systèmes Prolog efficaces. L'implémentation paralléle de PEPSys est l'une des premières implémentations efficaces du parallélisme OU en programmation logique. Le parallélisme est simple à mettre en œuvre puisque les programmes PEPSys ne diffèrent des programmes Prolog équivalents que par de simples pragmas. En particulier, il n'est pas nécessaire au programmeur de spécifier la granularité du parallélisme et donc de changer le programme pour des données différentes comme dans d'autres approches en programmation logique [Ramkumar et Kalé 89] ou dans d'autres langages parallèles [Odijk 87]. Un grand nombre de mesures ont été faites pour valider le modèle de calcul et estimer les sources de surcoût. Elles ont confirmé la validité des hypothèses faites lors des définitions du langage et du modèle de calcul. Elles ont également mis en évidence des optimisations possibles dans l'implémentation du parallélisme OU et de la gestion mémoire. Enfin si le parallélisme s'est révélé compatible avec l'utilisation des techniques les plus efficaces d'implémentation, le système parallèle compilé s'est également montré compatible avec l'implémentation des techniques de programmation logique avec contraintes, offrant ainsi de nouvelles perspectives pour la solution de problèmes de très grande taille.

9.2 Comparaison avec d'autres approches

La comparaison la plus naturelle est avec les autres systèmes multiséquentiels et particulièrement ceux qui ont été implémenté sur le même matériel.

Le système PEPSys est environ trois fois plus rapide que l'ANLWAM [Butler 86] qui fut le premier système logique parallèle efficace à être réalisé sur multiprocesseur. Ses performances sont très comparables à celles du système Aurora qui implémente le modèle SRI [Lusk et al. 88]. PEPSys et Aurora ont tous deux été implémentés au même moment, sur le même multiprocesseur, et ont utilisé les mêmes programmes de test. Le faible écart de performances constaté parfois en faveur de Aurora pourrait sans doute être corrigé en effectuant les améliorations suggérées dans la thèse.

La mise en œuvre du modèle SRI semble nettement plus simple que celle de PEPSys, bien que l'ordonnancement des processus soit plus complexe dans le modèle SRI. En outre le modèle SRI se prête mieux à l'implémentation des techniques de programmation avec contraintes du système CHIP [Van Hentenryck 89]. Cependant le modèle SRI ne semble pas implémentable sur les architectures extensibles à communication par message alors qu'une implémentation de PEPSys sur ce ce type d'architecture a été simulée [Baron et al 88]. Les médiocres performances

enregistrées par le simulateur sont susceptibles d'améliorations importantes en raison des progrès en cours dans l'architecture de ce type de multiprocesseurs. Signalons enfin que les techniques utilisées pour implémenter une partie de CHIP sur PEPSys seraient également applicables sur les multiprocesseurs à communication par message.

Aucune comparaison entre le système PEPSys et un système gardé n'ayant été faite, nous ne pouvons ici que mentionner l'étude comparative entre Aurora et l'implémentation de KL1 sur Sequent [Tick 89]. Ses principales conclusions sont qu'une amélioration des performances du système KL1 par un facteur compris entre 2 et 9 serait nécessaire pour égaler Aurora. En outre, Aurora bénéficie d'un meilleur comportement mémoire. Par contre, il semble qu'un système supportant le parallélisme ET dépendant permette d'exploiter du parallélisme de grain fin, ce qui n'est pas le cas des systèmes OU-parallèles.

9.3 Perspectives

Le travail décrit dans cette thèse offre de nombreux prolongements possibles dont certains sont en cours actuellement. Le système décrit a en effet été porté par Andy Cheese sur un Sequent Symmetry - utilisant des processeurs Intel 80386 qui sont trois à quatre fois plus efficaces que les processeurs NS32032 du MX500 - comportant 12 processeurs. Pierre Heuzé a porté sur ce système la partie de CHIP qui avait été implémentée sur le MX500. Ce système doit également être connecté à une base de données, la conjonction du parallélisme, des contraintes et des bases de données le rendant susceptible de traiter de très grosses applications. Il est actuellement utilisé pour développer des applications de biologie moléculaire en collaboration avec le projet américain de "cartographie génétique" [Overbeek 89], obtenant des facteurs d'accélération de l'ordre de 10 sur 12 processeurs pour des programmes de plusieurs millions d'inférences.

Comme nous l'avons signalé, l'implémentation complète de la combinaison du parallélisme ET avec le parallélisme OU reste à faire. Cette combinaison est nécessaire pour offrir un système parallèle capable de tirer parti de tout le parallélisme potentiel des applications. La recherche devra établir si le parallélisme ET indépendant de PEPSys sera suffisant ou s'il sera nécessaire de tirer parti d'autres formes de parallélisme comme le parallélisme ET dépendant [Brand 88b]. La combinaison des parallélismes OU et ET est très complexe et semble difficile à implémenter efficacement. Dans tous les cas, de nombreuses expérimentations seront nécessaires pour ajuster les paramètres intervenant dans la combinaison des parallélismes.

Une autre extension au travail de la thèse serait vers un nombre plus important de processeurs. Les multiprocesseurs UMA existants[1] tels que le MX500 peuvent être étendus jusqu'à une trentaine de processeurs. Selon les résultats de simulation des programmes PEPSys existants, la courbe des performances s'infléchit à partir

[1] Voir la section 6.2.1 pour une définition des termes UMA, NUMA et NORMA.

d'une quinzaine de processeurs, et ceci même pour les programmes les plus efficaces. L'expérimentation sur une machine à trente processeurs devrait permettre de tester si cet infléchissement demeure semblable quand on exécute des problèmes d'une taille supérieure, trop grande pour pouvoir être testés actuellement par le simulateur. Elle devrait également permettre d'expérimenter des stratégies permettant d'éliminer ou de retarder cet infléchissement.

Il n'est possible de dépasser la centaine de processeurs qu'avec des architectures NUMA. Il est vraisemblablement possible d'implémenter efficacement PEPSYs sur une telle architecture. En effet le coût matériel d'accès à une mémoire non locale est très inférieur au coût logiciel du déréférencement non local ou de l'accès à une *hash-window* (voir section 7.1.2 pour une estimation de ces coûts). Il est cependant difficile de prévoir l'ensemble des problèmes que peut poser à l'implémenteur ce type d'architecture ainsi que l'évolution des performances lorsqu'on augmente de façon importante le nombre de processeurs. Les performances resteraient-elles bonnes à condition d'augmenter encore la taille des problèmes? Le système parallèle pourrait alors être utilisé efficacement sur des problèmes de très grande taille.

Pour étudier le comportement du modèle sur un nombre de plusieurs milliers de processeurs, il serait nécessaire de passer à une architecture de type NORMA. Comme le montrent les résultats de simulation d'une architecture NORMA de quelques processeurs[2], une telle architecture ne permet pas à l'heure actuelle d'implémentation efficace de PEPSys, même si chaque processeur de traitement est couplé à un processeur de communication. La raison est la lenteur de la communication par message, la plus grande partie du temps étant passée dans le logiciel de fabrication et de lecture de message. Ce type de communication semble devoir être beaucoup plus efficace dans le futur, si l'on intègre les primitives de communication au code d'ordre du processeur de traitement [Dally et Wills 89]. Dans cette perspective, on pourrait alors envisager une implémentation de PEPSys sur une architecture de ce type. Il est vraisemblable que seuls des problèmes d'une taille inconnue actuellement en Prolog pourraient tirer parti de la puissance d'une telle machine de plus de mille processeurs.

9.4 Le parallélisme en programmation logique est-il rentable?

Les bons résultats obtenus permettent-ils d'affirmer que le parallélisme en programmation logique est rentable? Paradoxalement, la réponse à cette question n'a rien d'évident. [Lusk et al. 88] fait remarquer que les résultats obtenus par Aurora sur un Sequent Balance 8000 de huit processeurs (et donc aussi par PEPSys) sont plus lents que ceux obtenus en utilisant le système Quintus Prolog sur Sun 3/50, alors que le multiprocesseur est nettement plus coûteux que la station de travail.

[2]Obtenus en configurant l'architecture simulée avec un seul processeur par grappe

Certes l'efficacité de Aurora (et aussi de PEPSys) pourrait être multipliée par un facteur deux ou trois en réécrivant en assembleur l'interpréteur de *WAM* comme le fait Quintus, permettant aux systèmes parallèles d'égaler les performances de Quintus sur Sun. La principale explication des performances limitées du système parallèle réside dans la faible puissance des processeurs de traitement du Sequent (MX500), nettement inférieure à celle du processeur du Sun. Cette différence de puissance a deux origines : d'abord les processeurs du multiprocesseur ne sont pas aussi récents que ceux de la station de travail, ce décalage semblant dû au surcroît de temps nécessaire pour mettre au point un multiprocesseur par rapport à une station de travail ; de plus les processeurs du multiprocesseur ne peuvent pas être employés à pleine puissance en raison des problèmes d'accès à la mémoire. Il semble donc que ce type de décalage entre les processeurs des multiprocesseurs et des stations de travail soit appelé à se poursuivre tant que des progrès très rapides dans l'architecture des microprocesseurs permettront aux stations de travail monoprocesseur d'augmenter leur puissance de traitement aussi rapidement.

Au contraire, si un infléchissement des progrès des microprocesseurs se produisait, les stations de travail devraient augmenter leur puissance en ayant recours au parallélisme. Une autre évolution possible de l'architecture des multiprocesseurs assurerait la rentabilité du parallélisme. Ce serait le cas si les microprocesseurs étaient conçus dès l'origine pour être connectés dans une architecture multiprocesseur, réduisant ainsi le délai entre l'apparition d'un nouveau microprocesseur et celle des multiprocesseurs l'utilisant. C'est partiellement le cas du Motorola 88000 dont l'unité de gestion mémoire assure la cohérence de cache et qui peut être livré sur une carte comportant quatre processeurs de traitement. Ce type d'évolution est de nature à généraliser les architectures multiprocesseurs pour les stations de travail. A supposer que l'on assiste à l'une ou l'autre de ces évolutions possibles, les systèmes logiques parallèles se montreraient alors plus efficaces que les systèmes logiques séquentiels, du moins pour les problèmes se prêtant à la parallélisation.

La rentabilité du parallélisme en programmation logique est également susceptible d'être assurée si elle permet l'utilisation d'architectures massivement parallèles, de type NUMA ou NORMA. Comme nous l'avons signalé, il n'est pas encore établi que les bons résultats obtenus se généralisent à ces architectures. Si c'est le cas, il semble que seuls des problèmes de très grande taille pourront tirer réellement parti du parallélisme massif offert. Au lieu de rendre intantanée la solution de problèmes de taille petite ou moyenne, le parallélisme massif permettrait de résoudre des problèmes de taille trop importante pour se prêter à une solution sur les architectures actuelles.

Quelle que soit l'évolution de l'architecture des machines dans les années à venir, suivant l'un des schémas ci-dessus ou de façon encore complètement différente, il est vraisemblable que l'importance des multiprocesseurs continuera à croître. Les résultats obtenus montrent que la programmation logique parallèle se présente comme l'un des meilleurs candidats pour tirer parti de la puissance offerte, tant en raison de sa simplicité de mise en œuvre que de l'efficacité des ses implémentations.

Annexe A

Lexique et abbréviations

Lexique Français-Anglais

Dans l'ensemble de cette thèse, des termes techniques en Français ont été employés autant que possible. Certains termes Anglais ont été conservés lorsqu'il n'exite pas de terme Français satisfaisant équivalent. Un certain nombre de termes techniques sont mieux connus en Anglais qu'en Français ou alors il n'existe pas de traduction reconnue universellement en Français. C'est pour cette raison que le petit lexique qui suit est présenté dans le sens Francais vers Anglais.

but inclus	*trapped goal*
clos	*ground*
compétition	*contention*
déréférencement	*dereferencing*
environnement	*environment*
échange	*swapping*
équitable	*fair*
facteur d'accélération	*speedup*
famine	*starvation*
flot	*stream*
grappe	*cluster*
largeur d'abord	*breadth first*
liaison	*binding*
liaison profonde	*deep binding*
liaison superficielle	*shallow binding*
ordonnancement	*scheduling*
ordonnanceur	*scheduler*
pile	*stack*
pile de restauration, trace	*trail*
point de choix	*choice-point*
pointeur fantôme	*dangling pointer*
profondeur d'abord	*depth first*
ramasse miettes	*garbage collection*
retour arrière	*backtracking*
surcoût	*overhead*
tableau de liaison	*binding array*
tas	*heap*
taux de succès	*hit ratio*
trous noirs	*garbage slots*
verrou	*lock*

ABREVIATIONS COURANTES

OBL	*Or Branch Level*
MIMD	*Multiple Instructions Multiple Data-streams*
SIMD	*Single Instruction Multiple Data-streams*
NORMA	*No Remote Memory Access*
NUMA	*Non Uniform Memory Access*
UMA	*Uniform Memory Access*

Annexe B

Tableaux de mesures détaillés

Cette section donne deux échantillons des tableaux de mesures détaillés produits par le système de mesure et dont l'interprétation étendue à l'ensemble des programmes de test a été donnée dans le chapitre 8.

La signification des abréviations étiquetant les champs de ces tableaux est la suivante :

PEs : nombre de processeurs virtuels.

Tests : nombre de tests effectués. Tous les résultats sont les moyennes et écarts types calculées à partir de ce nombre de résultats moins un, la première série de mesures étant écartée pour tenir compte de l'effet cache.

Time : temps passé. La version du sytème qui effectue les mesures détaillées est environ 10 % plus lente que la version la plus rapide du système.

Klips : même remarque que précédement.

Infs : nombre d'inférences du programme.

Procs : nombre de processus légers créés durant l'exécution du programme.

AvLif : Durée de vie moyenne des processus légers.

%Small : Pourcentage du calcul effectué par les processus courts. Ici, les processus courts font moins de 100 inférences.

Idle : oisiveté ou temps passé à chercher du travail et à initialiser un processus léger. Cette mesure est peu précise à cause de la granularité de la mesure du temps du système Unix.

%Idle : pourcentage d'oisiveté dans le temps de calcul.

Stack : taille de l'espace utilisé par les piles . Il s'agit en fait de la somme des maximums utilisés par chacun des processeurs virtuels, sachant que la taille de chaque *hash-window* est de 2 Koctets et que chaque *bagof* réserve 40 Koctets sur la pile.

Local : espace utilisé par les piles locales (Koctets).

Globa : espace utilisé par les piles globales (Koctets).

Trail : espace utilisé par les piles traces (Koctets).

OBL : valeur maximum atteinte par les registres OBL.

BPs : nombre de *branch-points* créés durant l'exécution. Aucun n'est créé si le programme est exécuté sur un seul processeur.

SeqBP : nombre de *branch-points* utilisés séquentiellement.

%SeqB : pourcentage des *branch-points* utilisés séquentiellement.

OvBPs : Surcoût introduit par l'utilisation séquentielle de *branch-points* au lieu de points de choix. Cette mesure est un minorant qui assigne à chaque *branch-point* inutile un surcoût de 400 micro-secondes.

OvBlo : surcoût dû à la contention sur les verrous. Il s'agit d'un minorant.

Deref : nombre de déréférencements effectués durant l'exécution.

NlocD : nombre de déréférencements non-locaux effectués durant l'exécution.

%Nloc : pourcentage de déréférencements non-locaux relativement à l'ensemble des déréférencements.

OvNlo : Surcoût provoqué par les déréférencements non-locaux.

HWBnd : nombre de liaisons faites dans les *hash-windows* durant l'exécution.

%HWent : pourcentage des emplacements disponibles dans les *hash-windows* utilisés durant l'ex'ecution (taille des *hash-windows*: 128 éléments).

HWCol : nombre de collisions provoquées par les liaisons dans les *hash-windows*.

HWused : nombre de *hash-windows* utilisées durant l'exécution.

%HWused : pourcentage de *hash-windows* utilisées durant l'exécution.

HWdfs : nombre de déréférencements impliquant une recherche dans les *hash-windows*.

%HWdf : pourcentage de déréférencements impliquant une recherche dans les *hash-windows*.

HWacc : nombre d'accès aux *hash-windows* effectués durant l'exécution. L'accès à la *hash-window* locale compte pour 1 et l'exploration d'une chaîne de longueur i compte pour i accès.

OvHW : surcoût provenant du déréférencement dans les *hash-windows*. Il s'agit d'un minorant.

%HW1, %HW2, %HW3 : pourcentage des recherches dans les *hash-windows* imposant l'exploration de 1, 2, 3 *hash-windows* relativement à l'ensemble des recherches dans les *hash-windows.*

HWMax : longueur de la plus grande chaîne de *hash-windows* explorée durant l'exécution.

OvGLO : somme de tous les surcoûts mesurés : oisiveté, contention, déréférencements non-locaux et dans les *hash-windows.*

%OvGL : pourcentage du temps d'exécution provenant des surcoûts précédents.

Table B.1: Résultats détaillés du programme *farmer*

Il s'agit du plus petit programme de test utilisé. Pour la mesure, il est exécuté 100 fois.

PE's	1	2	3	4	5	6	7	8
Tests	5	5	5	5	5	5	5	5
Time	24.68+-0.01	16.08+-0.03	13.24+-0.46	13.20+-0.19	12.20+-0.17	11.48+-0.18	12.07+-0.08	12.27+-0.10
KLips	1.34	2.06	2.50	2.51	2.71	2.88	2.74	2.70
Infs	33102	33102	33102	33102	33102	33102	33102	33102
Procs	1+-0	1001+-0	1789+-12	2336+-34	2855+-10	3637+-3	4009+-15	4644+-12
AvLif	33102+-0	33+-0	18+-0	14+-1	11+-0	9+-0	8+-0	7+-0
%Small	0.00	6.65	57.94	97.58	97.58	97.58	97.58	97.58
Idle	0.01+-0.00	1.70+-0.03	2.21+-0.31	4.01+-0.24	4.59+-0.16	4.98+-0.21	6.17+-0.09	6.92+-0.09
%Idle	0.04	10.54	16.69	30.41	37.65	43.35	51.15	56.38
Stack	41.91+-0.00	47.42+-0.00	62.98+-1.10	82.29+-0.35	87.61+-1.91	95.55+-2.95	113.30+-1.23	123.40+-1.13
Local	1.11+-0.00	2.32+-0.00	3.91+-0.00	6.23+-0.36	7.04+-0.09	8.64+-0.35	10.30+-0.16	11.20+-0.10
Globa	40.77+-0.00	45.05+-0.00	58.98+-1.10	75.92+-0.02	80.41+-1.89	86.70+-2.66	102.76+-1.09	111.93+-1.17
Trail	0.03+-0.00	0.05+-0.00	0.09+-0.00	0.14+-0.01	0.16+-0.00	0.21+-0.01	0.24+-0.01	0.28+-0.00
OBL	0+-0	7+-0	7+-0	5+-0	4+-1	4+-0	3+-0	3+-0
BPs	0+-0	1500+-0	1700+-0	1700+-0	1700+-0	1700+-0	1700+-0	1700+-0
SeqBP	0+-0	1000+-0	580+-6	462+-13	220+-20	33+-7	11+-1	0+-0
%SeqB	0.00	66.67	34.09	27.18	12.93	1.96	0.65	0.00
OvBPs	0.00+-0.00	0.20+-0.00	0.08+-0.00	0.05+-0.00	0.02+-0.00	0.00+-0.00	0.00+-0.00	0.00+-0.00
OvBlo	0.00+-0.00	0.03+-0.00	0.02+-0.00	0.02+-0.00	0.02+-0.00	0.01+-0.00	0.01+-0.00	0.01+-0.00
Deref	189501+-0	191901+-0	197616+-106	201646+-91	203097+-123	202949+-113	205433+-44	205840+-61
NlocD	0+-0	29200+-0	71811+-761	95082+-438	102457+-494	107527+-461	117413+-169	122487+-321
%Nloc	0.00	15.22	36.34	47.15	50.45	52.98	57.15	59.51
OvNlo	0.00+-0.00	0.73+-0.00	1.20+-0.01	1.19+-0.01	1.02+-0.00	0.90+-0.00	0.84+-0.00	0.77+-0.00
HWBnd	0+-0	900+-0	2505+-36	4890+-100	6077+-108	6774+-74	9431+-45	10677+-87
%HWent	0.00	1.17	1.38	1.78	1.88	1.74	2.07	2.01
HWCol	0+-0	0+-0	3+-1	161+-18	1360+-63	32+-4	98+-8	227+-8
HWused	0+-0	600+-0	1419+-18	2147+-28	2530+-15	3038+-13	3552+-12	4158+-19
%HWused	0.00	59.94	79.29	91.92	88.63	83.53	88.60	89.54
HWdfs	0+-0	4100+-0	12764+-158	20040+-213	23081+-247	24185+-213	30031+-98	32108+-171
%HWdf	0.00	2.14	6.46	9.94	11.36	11.92	14.62	15.60
HWacc	0+-0	4300+-0	14124+-193	23444+-292	27450+-354	28871+-292	37183+-141	40271+-247
%HW1	0.00	95.12	89.92	84.12	82.37	81.87	77.90	76.38
%HW2	0.00	4.88	9.60	14.93	16.76	17.30	21.44	23.00
%HW3	0.00	0.00	0.40	0.79	0.44	0.49	0.00	0.00
HWMax	0+-0	2+-0	4+-0	4+-0	5+-0	5+-0	5+-0	5+-0
OvHW	0.00+-0.00	0.20+-0.00	0.43+-0.01	0.52+-0.01	0.48+-0.01	0.42+-0.00	0.46+-0.00	0.43+-0.00
OvGLO	0.01+-0.00	2.85+-0.03	3.94+-0.33	5.79+-0.25	6.13+-0.17	6.31+-0.22	7.48+-0.10	8.12+-0.10
%OvGL	0.05+-0.00	17.75+-0.20	29.73+-2.51	43.85+-1.92	50.28+-1.40	54.97+-1.90	62.01+-0.81	66.22+-0.78

Table B.2: Résultats détaillés du programme *hamilton*

Il s'agit du plus gros programme de test utilisé. Il procure des facteurs d'accélération supérieurs à 7 sur huit processeurs.

PE's	1	2	3	4	5	6	7	8
Tests	5	5	5	5	5	5	5	5
Time	267.37+-0.01	135.91+-0.05	91.17+-0.07	68.92+-0.17	55.49+-0.16	46.48+-0.33	40.58+-0.11	36.47+-0.16
KLips	1.75	3.45	5.14	6.80	8.44	10.08	11.55	12.85
Infs	468489	468489	468489	468489	468489	468489	468489	468489
Procs	1+-0	178+-3	173+-8	348+-75	353+-78	392+-169	632+-61	737+-68
AvLif	468489+-0	2632+-44	2720+-121	1399+-319	1369+-257	1468+-884	746+-71	640+-62
%Small	0.00	0.23	0.25	0.49	0.45	0.51	0.92	1.08
Idle	0.01+-0.00	0.70+-0.04	0.32+-0.05	0.37+-0.14	0.21+-0.03	0.23+-0.05	0.39+-0.06	0.36+-0.12
%Idle	0.00	0.52	0.36	0.53	0.39	0.50	0.97	0.99
Stack	55.05+-0.00	75.08+-0.00	98.90+-2.96	128.67+-7.96	150.95+-5.19	160.41+-17.40	189.32+-3.33	213.15+-6.66
Local	10.22+-0.00	19.99+-0.00	30.05+-0.00	41.63+-1.80	51.01+-2.34	62.89+-4.17	68.55+-0.59	83.52+-2.17
Globa	44.20+-0.00	53.89+-0.00	67.11+-2.96	84.75+-6.28	97.22+-3.17	94.24+-13.90	117.18+-3.08	125.54+-7.49
Trail	0.63+-0.00	1.20+-0.00	1.74+-0.00	2.29+-0.07	2.72+-0.09	3.28+-0.14	3.59+-0.02	4.09+-0.11
OBL	0+-0	37+-0	40+-0	49+-7	62+-2	68+-8	83+-6	94+-14
BPs	0+-0	593+-9	1204+-35	3782+-323	5297+-737	6231+-2271	10104+-903	16669+-991
SeqBP	0+-0	416+-7	1032+-42	3435+-310	4945+-678	5840+-2109	9474+-868	15934+-956
%SeqB	0.00	70.14	85.75	90.84	93.36	93.73	93.76	95.59
OvBPs	0.00+-0.00	0.08+-0.00	0.14+-0.01	0.34+-0.03	0.40+-0.05	0.39+-0.14	0.54+-0.05	0.80+-0.05
OvBlo	0.00+-0.00	0.01+-0.00	0.01+-0.00	0.02+-0.00	0.02+-0.00	0.02+-0.01	0.02+-0.00	0.03+-0.00
Deref	1621664+-0	1622492+-8	1622531+-15	1623287+-252	1623340+-423	1623395+-593	1624265+-193	1624629+-225
NlocD	0+-0	50487+-24	61524+-29	79077+-994	79195+-1617	82186+-5314	96435+-603	99034+-1270
%Nloc	0.00	3.11	3.79	4.87	4.88	5.06	5.94	6.10
OvNlo	0.00+-0.00	1.26+-0.00	1.03+-0.00	0.99+-0.01	0.79+-0.02	0.68+-0.04	0.69+-0.00	0.62+-0.01
HWBnd	0+-0	579+-7	638+-8	1163+-195	1170+-182	1242+-429	1944+-141	2147+-175
%HWent	0.00	2.55	2.90	2.62	2.60	2.48	2.41	2.28
HWCol	0+-0	1+-0	2+-1	5+-3	22+-8	13+-9	6+-3	11+-7
HWused	0+-0	177+-3	172+-8	347+-75	352+-78	391+-169	630+-63	735+-67
%HWused	0.00	99.44	99.42	99.71	99.72	99.74	99.68	99.73
HWdfs	0+-0	1477+-15	1574+-13	2929+-486	2963+-627	3081+-1071	4774+-327	5348+-421
%HWdf	0.00	0.09	0.10	0.18	0.18	0.19	0.29	0.33
HWacc	0+-0	1533+-16	1759+-5	3404+-529	3415+-642	3516+-1224	5488+-338	6104+-480
%HW1	0.00	96.17	88.28	88.50	89.03	89.88	88.45	89.43
%HW2	0.00	3.83	11.67	8.08	8.03	7.64	9.42	8.41
%HW3	0.00	0.00	0.05	2.41	2.00	1.34	1.29	1.23
HWMax	0+-0	2+-0	3+-1	5+-0	6+-1	5+-2	7+-1	7+-1
OvHW	0.00+-0.00	0.07+-0.00	0.05+-0.00	0.08+-0.01	0.06+-0.01	0.05+-0.02	0.07+-0.01	0.07+-0.01
OvGLO	0.01+-0.00	2.13+-0.04	1.55+-0.06	1.79+-0.20	1.48+-0.12	1.38+-0.26	1.72+-0.12	1.88+-0.18
%OvGL	0.00+-0.00	1.57+-0.03	1.70+-0.06	2.60+-0.29	2.67+-0.21	2.96+-0.57	4.24+-0.30	5.15+-0.50

Bibliographie

[Abbott 87] R.J. Abbott. Knowledge Abstraction. *Communications of the ACM*, 30(8), August 1987.

[Ali 88] K. Ali. OR-Parallel execution of Prolog on bc-machine. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545, Seattle, August 1988.

[Alshawi 88] H. Alshawi and D. Moran. The delphi model and some preliminary experiments. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1578–1589, Seattle, August 1988.

[Appleby 88] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on the WAM. *Communications of the ACM*, 31(6), June 1988.

[Baldwin 87] Douglas Baldwin. *Why we can't Program Multiprocessors the Way We Are Trying to Do It Now.* Technical Report Technical Report 224, Department of Computer Science, University of Rochester, August 1987.

[Baron 88] Uri Baron. *A Scheme for Aborting Useless Processes in PEPSys.* Internal Report PEPSys-27, ECRC, June 1988.

[Baron et al 88] U.C. Baron, B. Ing, M. Ratcliffe, and P. Robert. A distributed architecture for the PEPSys Parallel Logic Programming System. In *Proceedings ICPP'88*, International Conference on Parallel Processing, Chicago, August 1988.

[Baron, Chassin et al. 88] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: an overview and evaluation results. In *Proceedings*

FGCS'88, International Conference on Fifth Generation Computer Systems, Tokyo, Nov-Dec 1988.

[Baron 89] Uri Baron. *Scheduling Experiments in PEPSys: Results and Analysis.* Internal Report PEPSys-35, ECRC, January 1989.

[Bekkers 86] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. Mali: a memory with a real-time garbage collector for implementing Logic Programming. In *Proc. - 3rd Symposium on Logic Programming*, Salt Lake City, 1986.

[Benker et al. 89] H. Benker, J.M. Beacco, S.Bescos, M. Dorotchevsky, Th. Jeffre, A. Pöhlman, J. Noye, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, and G. Watzlawik. KCM: a Knowledge Crunching Machine. In *Proc. of the* 15^{th} *Symposium on Computer Architecture*, IEEE, June 1989.

[Bocca et al. 86] J. Bocca, H. Decker, J.-M. Nicolas, L. Vieille, and M. Wallace. Some steps towards a DBMS based KBMS. In *Information Processing 86*, pages 1061–1067, 1986.

[Boizumault 88] P. Boizumault. *Prolog l'implantation.* Masson, 1988.

[Borgwardt 84] Peter Borgwardt. Parallel Prolog using stack segments on shared memory multiprocessors. In *84 Int. Symposium on Logic Programming*, pages 2–11, IEEE, February 1984.

[Brand 88a] Per Brand. *Wavefront scheduling.* Internal Report, Gigalips Project, 1988.

[Brand 88b] Per Brand, Seif Haridi, and David H.D. Warren. Andorra Prolog: the language and its application in distributed simulation. In *Proceedings FGCS'88*, International Conference on Fifth Generation Computer Systems, Tokyo, Nov-Dec 1988.

[Butler et al. 88] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling OR-Parallelism: an Argonne perspective. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seattle, August 1988.

[Butler 86] R. Butler, E.L. Lusk, R. Olson, and R.A. Overbeek. *ANLWAM - A Parallel Implementation of the Warren*

Abstract Machine. Internal Report, Argonne National Laboratory, 1986.

[Calderwood 89] A. Calderwood and P. Szeridi. Scheduling OR-Parallelism in Aurora. In *Proceedings of the 6th International Conference on Logic Programming*, Lisboa, June 1989.

[Carlsson 87] Mats Carlsson. *Internals of Sicstus Prolog version 0.6.* Internal Report, Gigalips Project, November 1987.

[Carlsson 88a] M. Carlsson, K. Danhof, and R. Overbeek. A simplified approach to the implementation of AND-Parallelism in an OR-Parallel environment. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577, Seattle, August 1988.

[Carlsson 88b] M. Carlsson and J. Widen. *SICStus Prolog User Manual.* Research Report, SICS, 1988.

[Chassin 86] J. Chassin de Kergommeaux. Machines abstraites Prolog. In AFCET-GROPLAN, editor, *Bigre + Globule No 50*, September 1986.

[Chassin et al. 87] J. Chassin de Kergommeaux, P. Robert, and H. Westphal. *An Abstract Machine for the Implementation of the PEPSys model.* Technical Report CA-26, ECRC, February 1987.

[Chassin et al 88] J. Chassin de Kergommeaux, D. Peterson, W. Rapp, and H. Westphal. *The Implementation of PEPSys on an MX-500 Multiprocessor.* Technical Report CA-38, ECRC, March 1988.

[Chassin et Robert 88] J. Chassin de Kergommeaux and P. Robert. An Abstract Machine to Implement Efficiently OR-AND Parallel Prolog. *Supplement to the Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, 1988. To appear in the Journal of Logic Programming.

[Chassin et al. 88] J. Chassin, J.C. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *Proceedings*, European Conference on Artificial Intelligence, Munich, August 1988.

[Chassin 89] J. Chassin de Kergommeaux. *Measures of the PEPSys Implementation on the MX500.* Technical Report CA-44, ECRC, January 1989.

[Chassin, Baron et al. 89] J. Chassin de Kergommeaux, U. C. Baron, W. Rapp, and M. Ratcliffe. Performance analysis of a Parallel Prolog: a correlated approach. In *Parallel Architectures and Languages Europe PARLE'89*, page 151–163, Eindhoven, June 1989.

[Chassin, Codognet et al. 89] J. Chassin de Kergommeaux, P. Codognet, P. Robert, and J-C. Syre. Une revue des modèles de Programmation Logique Parallèle. *Technique et Science Informatiques (TSI)*, 8(3 et 4), 1989.

[Chikayama 87] T. Chikayama. Parallel Inference System Researches in the FGCS Project. In 4^{th} *Symposium on Logic Programming*, pages 274–276, San Francisco, Sept. 1987.

[Clark 81a] K.L. Clark and F. McCabe. *The Control Facilities of IC-PROLOG*, pages 122–149. Edinburgh University Press, 1981.

[Clark 81b] K. Clark and S. Gregory. A relational language for Parallel Programming. In *proc. Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, ACM, 1981.

[Clark 86] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[Clocksin et Mellish 81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag Berlin Heidelberg New York, 1981.

[Clocksin 87] W. F. Clocksin and H. Alshawi. *A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors.* Technical Note CCSRC-3, SRI International, 1987.

[Codognet 88] C. Codognet, P. Codognet, and G. File. Yet another intelligent backtracking method. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 447–465, Seattle, August 1988.

[Codognet 86] C. Codognet, M. Corsini, and G. File. *Optimisation of Logic Programs based on their static analysis.* Technical

Report 86-24, Université de Bordeaux-1, 1986. et proc. GULP 88, Roma 1988.

[Cohen 87] J. Cohen and T. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, April 1987.

[Colmerauer 87] A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine* 12(9), August 1987.

[Conery 83] J. S. Conery. *The AND/OR Process Model for Parallel Execution of Logic Programs.* PhD thesis, Univ. of California, Irvine, 1983. et Tech. Report 204, Dept. of Computer and Information Science, UCI.

[Crammond 85] J. Crammond. A comparative study of unification algorithms for OR-Parallel execution of Logic languages. In DeGroot, editor, *Int. Conf. on Parallel Processing*, pages 131–138, IEEE, St. Charles, Ill., August 1985.

[Crammond 86] J. Crammond. An execution model for commited-choice non-deterministic languages. In *Proc. - 3rd Symposium on Logic Programming*, Salt Lake City, 1986.

[Dally et Wills 89] W. J. Dally et D. S. Wills. Universal Mechanisms for Concurrency. In *PARLE'89 Parallel Architectures and Languages Europe* pages 19–33, Eindhoven, June 1989.

[DeGroot 85] D. DeGroot and J.H. Chang. A comparison of two AND-Parallel execution models. *Congres AFCET Informatique*, March 1985.

[DeGroot 84] Doug DeGroot. Restricted AND-Parallelism. In *Proc. of FGCS'84*, pages 471–478, ICOT, November 1984.

[Dincbas et al. 88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T.Graf, and F. Berthier. The constraint Logic Programming language CHIP. In *Proc. of FGCS'88*, ICOT, December 1988.

[Disz 87] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In 4th *Int. Conf. on Logic Programming*, pages 576,599, Melbourne, May 1987.

[Dwork 84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[van Emden 84] M. H. van Emden. An Interpreting Algorithm for Prolog Programs. *Implementations of Prolog*, T.A. Campbell (ed.), Ellis Horwood Series in Artificial Intelligence, 1984.

[Foster 86] Ian Foster, Steve Gregory, Graem Ringwood, and Ken Sayoh. A sequential implementation of Parlog. In 3rd *Int. Conf. on Logic Programming*, pages 149–156, London, July 1986.

[Hailperin 86] M. Hailperin and H. Westphal. *A Computational Model for PEPSys.* Technical Report CA-16, ECRC, April 1986.

[Hailperin 85] M. Hailperin and H. Westphal. *An Empirical Study of Locality of References in Prolog.* Technical Report CA-15, ECRC, 1985.

[Haridi 89] S. Haridi and E. Hagersten. The cache coherence protocol of the data diffusion machine. In *PARLE'89 Parallel Architectures and Languages Europe*, pages 1–18, Eindhoven, June 1989.

[Hausman 88] A. Calderwood B. Hausman, A. Ciepielewski. Cut and side-effects in OR-Parallel Prolog. In *Proceedings FGCS'88*, International Conference on Fifth Generation Computer Systems, Tokyo, Nov-Dec 1988.

[Hermenegildo 86a] M. V. Hermenegildo. An abstract machine for restricted AND-Parallel execution of Logic Programs. In 3rd *Int. Conf. on Logic Programming*, pages 25–39, London, July 1986.

[Hermenegildo 87] M. V. Hermenegildo. Relating goal scheduling, precedence, and memory management in AND-Parallel execution of Logic Programs. In 4th *Int. Conf. on Logic Programming*, pages 556–575, Melbourne, May 1987.

[Hermenegildo 86] Manuel V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Computer Science, The University of Texas at Austin, August 1986. TR-86-20.

[Hermenegildo 86b] M. V. Hermenegildo and R. I. Nasr. Efficient management of backtracking in AND-Parallelism. In 3rd *Int. Conf. on Logic Programming*, pages 40–54, London, July 1986.

[Houri 86] A. Houri and E. Shapiro. *A sequential abstract machine for Flat Concurrent Prolog.* internal report CS-86-20, Weizmann Institute, July 1986.

[Ing 87a] Bounthara Ing. *Tourist Information Advisor: A Case Study of an Application in PEPSys, Final Report.* Internal Report PEPSys-32, ECRC, December 1987.

[Ing 87b] Bounthara Ing. *COKE: An Analysis Tool for PEPSys Programmes.* Internal Report PEPSys-23, ECRC, October 1987.

[Jaffar 87] J. Jaffar and J-L Lassez. Constraint Logic Programming. In *POPL-87*, Munich, January 1987.

[Kalé 87a] L. V. Kalé. The Reduce-OR process model for Parallel evaluation of Logic Programs. In *Proc. of the Fourth International Conference of Logic Programming*, pages 616–632, Melbourne, May 1987.

[Kalé 87b] L. V. Kalé. Completeness and full Parallelism of Parallel Programming schemes. In 4^{th} *Symposium on Logic Programming*, pages 125–133, San Francisco, Sept. 1987.

[Kimura 87] Y. Kimura and T. Chikayama. An abstract KL1 machine and its instruction set. In 4^{th} *Symposium on Logic Programming*, pages 468–477, San Fransisco, Sept. 1987.

[Kharoune 88] Mouloud Kharoune. *MIPAP Un modèle d'interprétation Parallèle pour Prolog.* Thèse de Troisième Cycle, Université de Rennes I, Janvier 1988.

[Kowalski 79] R.A. Kowalski. *Logic for Problem Solving.* Elsevier Science Publishing, 1979.

[Kliger 88] S Kliger and E. Shapiro. A decision tree compilation algorithm for FCP. In MIT Press, editor, 5^{th} *Int'l Conference/Symposium on Logic Programming*, pages 1315–1336, Seattle, August 1988.

[LeBlanc 87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[LeBlanc 88] T. J. LeBlanc, M. Scott, and C. Brown. Large-scale Parallel Programming : experience with the BBN Butterfly Parallel processor. *SIGPLAN Notices*, 23(9), September 1988.

[Levy 86a] J. Levy. Shared memory execution of committed-choice languages. In 3rd *Int. Conf. on Logic Programming*, pages 298–312, London, July 1986.

[Levy 86b] Jacob Levy. A GHC abstract machine and instruction set. In 3rd *Int. Conf. on Logic Programming*, pages 157–171, London, July 1986.

[Lusk et al. 88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwodd, P. Szeridi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, and B. Hausman. The Aurora OR-Parallel Prolog System. In *Proceedings FGCS'88*, International Conference on Fifth Generation Computer Systems, Tokyo, Nov-Dec 1988.

[Lin 88] Y. J. Lin and V. Kumar. AND-Parallel execution of Logic Programs on a shared memory multiprocessor: a summary of results. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141, Seattle, August 1988.

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New-York, London, Paris, Tokyo, 1987.

[Mannila 87] H. Mannila and E. Ukkonen. Flow analysis of Prolog Programs. In 4th *Symposium on Logic Programming*, pages 205–214, San Francisco, Sept. 1987.

[Masuzawa 86] H. Masuzawa et al. Kabu Wake Parallel inference mechanism and its evaluation. In *1986 FJCC*, pages 955–962, IEEE, November 1986.

[Meier 89] M. Meier, P. Dufresne, D. de Villeneuve, et al. Sepia: an extendable Prolog System. In *IFIP'89*, San Francisco, August 1989.

[Mellish 86] C. S. Mellish. Abstract interpretation of Prolog Programs. In 3rd *Int. Conf. on Logic Programming*, pages 463–474, London, July 1986.

[Noye 87] J. Noyé, J-C Syre, and et al. ICM3: design and evaluation of an inference crunching machine. In *Proc.* 5th *Int. Workshop on Database Machines*, pages 1–14, Japan, October 1987.

[Odijk 87] E.A.M Odijk. The DOOM System and its applications: A survey of ESPRIT 415 subproject A, Philips Research Laboratories. In *Parallel ARchitectures and Languages Europe*, page 461–479, Eindhoven, March 1987.

[Overbeek 89] Ross Overbeek. Molecular Genetics Project - Applications of Logic Programming in Genome Sequencing *Tutorial T5, North American Conference on Logic Programming*, Cleveland, october 1989.

[Paaki 88] J. Paakki. A note on the speed of Prolog. *SIGPLAN Notices*, 23(8), 1988.

[Percebois et al. 86] C. Percebois, I. Futo, I. Durand, C. Simon, and B. Bonhoure. Résolution Parallèle de sous-buts indépendants dans le graphe de R. Kowalski. In *Actes du 5eme Seminaire sur la Programmation Logique*, pages 553,570, CNET, Tregastel, May 1986.

[Pfister 85] G. F. Pfister, W.C. Brantley et al. The IBM research Parallel processor prototype (RP3): introduction and architecture. In *Proc. of the International Conference on Parallel Processing*, pages 764–771, August 1985.

[Quinn 87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill International Editions, 1987.

[Quintus 85] *Quintus Prolog Reference Manual.* Quintus Computer Systems Ltd, 1985.

[Ramkumar et Kalé 89] B. Ramkumar and L. V. Kalé *Compiled Execution of the REDUCE-OR Process Model on Multiprocessors.* Technical Report UIUC-DCS-R-84-151Z, University of Illinois at Urbana-Champaign, 1989.

[Rapp 88] Wolfgang Rapp. *The Implementation and Evaluation of the PEPSys Module Concept.* Technical Report CA-40, ECRC, October 1988.

[Ratcliffe et Robert 85] Michael Ratcliffe and Philippe Robert. *The Static Analysis of Prolog Programs.* Technical Report CA-11, ECRC, October 1985.

[Ratcliffe et Robert 86] Michael Ratcliffe and Philippe Robert. *PEPSy: A Prolog for Parallel Processing.* Technical Report CA-17, ECRC, April 1986.

[Ratcliffe et Syre 87] J.C. Syre M. Ratcliffe. The PEPSys Parallel Logic Programming language. In *IJCAI87*, pages 48–55, Milano, Italy, August 1987.

[Reintjes 88] P. B. Reintjes. A VLSI Design Environment in Prolog. In *Proc. of the 5th International Conference/Symposium on Logic Programming*, pages 70–81, August 1988.

[Robert 88] Philippe Robert. *Une Machine Abstraite pour la Mise en Oeuvre du Parallélisme OU/ET en Programmation Logique.* Thèse, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, Juin 1988.

[Robinson 65] J.A. Robinson. A machine-oriented Logic based on the resolution principle. *Journal of the ACM*, 12(1), January 1965.

[Saraswat 86] V. A. Saraswat. *Problems with Concurrent Prolog.* Technical Report CMU-CS-86-100, Carnegie-Mellon University, January 1986.

[Sato 87] M. Sato and et al. Kl1 execution model for pim cluster with shared memory. In *4th Int. Conf. on Logic Programming*, pages 339–355, Melbourne, May 1987.

[Sato 88] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a shared memory multiprocessor. In *Proc. - IFIP Working Conf. on Parallel Processing*, North Holland, May 1988.

[Scott 1819] Walter Scott. *Ivanhoé.*

[Shapiro 83b] Ehud Shapiro. Object oriented Programming in Concurrent Prolog. *New Generation Computing*, 1(1):25–48, 1983.

[Shapiro 83a] Ehud Y. Shapiro. *A subset of Concurrent Prolog and its interpreter.* Technical Report, Weizmann Institute, Rehovot, February 1983.

[Silvermann 86] W. Silvermann, M. Hirsch, A. Houri, and E. Shapiro. *The Logix System User manual, version 1.21.* TR CS-21, Weizmann Institute of Science, July 1986.

[Simonet 81] M. Simonet. *W.Grammaires et logique du premier ordre pour la définition et l'implantation des langages.* Thèse d'Etat, USMG Grenoble, Juillet 1981.

[Sohma et al. 85] Yukio Sohma, Ken Satoh, Koichi Kumon, Hideo Masuzawa, and Akihiro Itashiki. A new Parallel inference mechanism based on sequential processing. In *IFIP TC-10 Working Conf. on Fifth Generation Computer Architecture*, Manchester, July 15-18 1985.

[Sterling et Shapiro 86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press Series in Logic Programming, 1986.

[Syre et al. 88] J.C. Syre, P. Robert, and J. Chassin. Le Système Logique Parallèle PEPSys In M. Dincbas S. Bourgault, editor, *Programmation en Logique*, pages 425–454, 7ème Séminaire, Trégastel, May 1988.

[Szeridi 89] Peter Szeridi. Performance Analysis of the Aurora OR-Parallel Prolog System. *North American Conference on Logic Programming* Cleveland, October 1989.

[Tick 88] E. Shapiro and E. Tick. Baskett's puzzle- a challenge and response. *Logic Programming Newsletter*, 2, October-November 1988.

[Tick 89] E. Tick. A performance comparison of AND- and OR-Parallel Logic Programming architectures. In *Proceedings of the 6th International Conference and Symposium on Logic Programming*, Lisbonne, June 1989.

[Thom 86] J. A. Thom and J.A. Zobel. *NU-Prolog 1.0 Reference Manual.* Technical Report 86/10, Department of Computer Science, University of Melbourne, 1986.

[Ueda 85] Kazunori Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, June 1985.

[Van Hentenryck 89] P. Van Hentenryck. Parallel constraint satisfaction in Logic Programming: preliminary results of CHIP within PEPSys. In *Proc. of the 6th International Conference on Logic Programming*, June 1989.

[Warren 77] D. H. D. Warren. *Implementing Prolog - Compiling Logic Programs.* Research Reports No. 39,40, University of Edinburg, 1977.

[Warren 80] D. H. D. Warren. An improved Prolog implementation which optimises tail recursion. In *Logic Programming Workshop*, Debrecen, 1980.

[Warren 83] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report tn309, SRI, October 1983.

[Warren 87a] D.H.D Warren. OR-Parallel execution models of Prolog. In *TAPSOFT 87, Joint Conference on Theory and Practice of Software Development*, Pisa, March 1987.

[Warren 87b] D.H.D. Warren. The SRI model for OR-Parallel execution of Prolog. abstract design and implementation issues. In *4th Symposium on Logic Programming*, pages 46–53, San Fransisco, Sept. 1987.

[Westphal 87] H. Westphal and D. Peterson. An efficient implementation of instant replay. Internal Note, Computer Architecture Group, ECRC, 1987.

[Westphal et al. 87] H. Westphal, P. Robert, J. Chassin, and J.-C. Syre. The PEPSys model: combining backtracking, and- and OR-Parallelism. In *4th Symposium on Logic Programming*, pages 436–448, San Fransisco, Sept. 1987.

[Wilson 87] A. W. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. 14th Annual Int. Symposium on Computer Architecture*, pages 244–252, Pittsburg, June 1987.

[Winsborough 88] W. Winsborough and A. Waern. Transparent AND-Parallelism in the presence of shared free variables. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle, August 1988.

[Yang 88] R. Yang and D. H. D. Warren . *The Andorra Execution Model of Prolog.* Internal Report, Gigalips Project, 1988.

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT INGENIEUR,
DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du **16 avril 1974**,

Vu les dispositions de l'Arrêté du **5 juillet 1984**,

Vu les rapports de M. D. H. D. WARREN, Professeur U. Bristol.

M. L. TRILLING, Professeur U.J.F

Monsieur CHASSIN de KERGOMMEAUX Jacques est autorisé(X)
à présenter une thèse en vue de l'obtention du Doctorat de
l'Université Joseph Fourier, Grenoble 1, Spécialité Informatique

Grenoble, le 15 NOV. 1989

Le Président de l'Université
Joseph Fourier - Grenoble 1

A. NEMOZ
Po / La Vice-Présidente Formation,

N. LONGEQUEUE

Résumé

Cette thèse est consacrée à l'implémentation de PEPSys (Parallel ECRC Prolog System) sur un multiprocesseur à mémoire partagée et à l'évaluation de cette implémentation. Le projet PEPSys vise à exploiter le parallélisme en programmation logique pour obtenir, sur les multiprocesseurs existants actuellement, des gains de performances relativement aux systèmes Prolog les plus efficaces. Un langage, extension de Prolog, un modèle de calcul et une machine abstraite basée sur la *WAM* ont été définis et validés par une implémentation sur multiprocesseur et une simulation d'architecture parallèle extensible. Le modèle de calcul supporte les parallélismes OU et ET indépendant ainsi que leur combinaison avec l'exécution séquentielle et le retour-arrière. L'implémentation de PEPSys qui fait l'objet de cette thèse constitue l'un des premiers systèmes logiques OU-parallèles à procurer des gains de performances, relativement aux systèmes Prolog séquentiels efficaces. Les nombreuses mesures présentées dans la thèse permettent de valider cette implémentation ainsi que les principaux mécanismes du modèle de calcul, tout en suggérant des optimisations.
Mots clés : PEPSys : Parallel ECRC Prolog System, Parallélismes OU et ET, Implémentation sur multiprocesseur, Prolog Parallèle basé sur la WAM, Mesures d'exécution, Gains de performances en Parallèle.

Abstract

This thesis is dedicated to the implementation of PEPSys (Parallel ECRC Prolog System) on a shared memory multiprocessor and to the evaluation of this implementation. The PEPSys project aims at exploiting the parallelism in logic programming to obtain, on existing multiprocessors, performance improvements compared to efficient sequential Prolog implementations. A language, extension of Prolog, a computational model and an abstract machine based on the WAM have been defined and validated by a multiprocessor implementation and a simulator of extensible architectures. The computational model supports OR and independent AND parallelisms and the combination of both with sequential execution and backtracking. The implementation of PEPSys which is the main topic of this thesis, is one of the first OR-parallel logic systems to provide efficiency improvements, compared to state of the art Prolog systems. The numerous measures provided in this thesis allow to validate this implementation and the main design decisions of the computational model, while suggesting improvements.
Keywords: PEPSys: Parallel ECRC Prolog System, OR-AND parallelism, Multiprocessor Implementation, WAM-based Parallel Prolog, Execution Measures, Parallel Speedups.