Une contribution à l'étude du parallélisme ou en Prolog sur des machines sans mémoire commune

Claudio Fernando Resin Geyer

▶ **To cite this version:**

Claudio Fernando Resin Geyer. Une contribution à l'étude du parallélisme ou en Prolog sur des machines sans mémoire commune. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1991. Français. NNT : . tel-00340398

HAL Id: tel-00340398

https://theses.hal.science/tel-00340398

Submitted on 20 Nov 2008

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.

THESE

présentée par

Cláudio Fernando RESIN GEYER

pour obtenir le titre de **DOCTEUR**

de l'UNIVERSITE JOSEPH FOURIER - GRENOBLE I

(arrêté ministériel du 5 juillet 1984)

Spécialité: **INFORMATIQUE**

UNE CONTRIBUTION A L'ETUDE DU PARALLELISME OU EN PROLOG SUR DES MACHINES SANS MEMOIRE COMMUNE

Date de soutenance: 29 octobre 1991.

Composition du Jury:

Yves CHIARAMELLA	*Président*
Yves BEKKERS	*Rapporteurs*
Gilles BERGER-SABBATEL	
Phillipe JORRAND	*Examinateurs*
Laurent TRILLING	
Jacques BRIAT	

Thèse préparée au sein du Laboratoire de Génie Informatique

Remerciements

Je tiens à remercier

Yves Chiaramella, directeur du Laboratoire de Génie Informatique (LGI), de me faire l'honneur de présider le jury de cette thèse.

Jacques Briat, maître de conférences à l'Université Joseph Fourier, qui m'a accepté dans son équipe (Flop) et qui a bien dirigé mon travail. Qu'il soit également remercié pour ses conseils et ses critiques lors de la rédaction de ce document.

Yves Bekkers, directeur de recherche INRIA, et Gilles Berger-Sabbatel, chargé de recherches CNRS, d'avoir bien voulu juger ce travail.

Philippe Jorrand, directeur du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle, et Laurent Trilling, professeur à l'Université Joseph Fourier, qui ont accepté de participer du jury de cette thèse.

Jacques Mossière, directeur de l'ENSIMAG, de m'avoir invité à travailler au sein de son laboratoire.

Mon camarade et collègue Michel Favre, dont la coopération lors du développement du projet Opera a toujours été amicale et fructueuse.

Tous les membres de l'équipe Flop et de l'équipe Sympa, au LGI; en particulier Philippe Waille qui m'a initié à la connaissance de la machine Tnode, et Jacques Eudes et Miguel Santana qui ont participé au développement des logiciels de base de cette machine.

Thadeu Botteri-Corso, pour m'avoir lancé dans le domaine du parallélisme et aux langages déclaratifs.

Résumé

Cette thèse est consacrée à l'étude de l'implantation du parallélisme OU en Prolog sur des machines sans mémoire commune. Nous présentons le modèle multi-séquentiel OU Opera, implanté par compilation (machine abstraite de Warren - WAM), en préservant la sémantique de Prolog. Les deux problèmes principaux d'un tel système, la gestion de contextes multiples et l'ordonnancement, sont détaillés. La gestion des contextes multiples s'effectue par copie incrémentale, en parallèle au calcul. Pour que ceci reste efficace et cohérent, le traitement des variables conditionnelles a été inclus dans la WAM. Notre méthode introduit une nouvelle pile pour ces variables dont l'initialisation, la liaison et la déliaison ont été modifiées. Le coût des opérations séquentielles de la WAM est constant et indépendant du nombre de processus. Nous proposons encore une méthode simple et efficace pour la réalisation de la coupure.

Un prototype Opera a été implanté sur un réseau de Transputers. Dans ce prototype, l'ordonnancement a été résolu par une méthode basée sur des heuristiques d'évaluation de charge. Cet ordonnancement est mis en œuvre par une architecture centralisée où un processus ordonnanceur unique régule la charge des autres processus. L'ordonnanceur utilise une représentation approximative de l'état du système. La partie séquentielle du prototype Opera constitue l'un des systèmes Prolog les plus efficaces existant actuellement sur le Transputer. Ses gains de performance en parallèle sont aussi effectifs.

Mots-clés: Opera: Ou Parallélisme Et Régulation Adaptative, Prolog, parallélisme OU multi-séquentiel, Prolog parallèle basé sur la WAM, implantation sur machine parallèle sans mémoire commune, gestion des contextes multiples, copie incrémentale, ordonnancement, coupure en parallèle.

Abstract

This thesis is dedicated to the study of the implementation of Or-parallel Prolog over distributed memory machines. The Opera Or multi-sequential model is presented. It uses compiling techniques (Warren Abstract Machine) and preserves the Prolog semantics. Multi-environment management and scheduling, the two major problems of Opera, are described. Multi-environment management is realized by incremental copying, in parallel to the computation. The treatment of conditional variables is included in the WAM, in order to allow an efficient and coherent cooperation. Our method introduces a new pile for these variables, initialization, binding and unbinding of which are adapted. The cost of WAM sequential operations is constant and independent of the number of processes. We also propose a simple and efficient method for implementing cut in parallel.

An Opera prototype has been implemented over a Transputer array. In the current prototype, scheduling is resolved by heuristics of load evaluation. This scheduling is centralized, a unique process balancing the load of the other Prolog workers, and using an approximate representation of the state of the system. The Opera prototype is one of the most efficient Prolog implementations on the Transputer, and reaches effective speed-ups in parallel.

Key-words: Opera: Or Parallelism and Adaptable Balancing, Prolog, Or multi-sequential parallelism, based-WAM parallel Prolog, implementation over distributed parallel machine, multi-environment management, incremental copy, scheduling, parallel cut.

Table des Matières

1. Chapitre 1 15
   1.1. Le Thème 15
   1.2. Justification 15
   1.3. Contexte Mondial 15
   1.4. Le Contexte Local 16
   1.5. Contribution de l'Auteur 16
   1.6. La Structure de la Thèse 17
2. Chapitre 2 19
   2.1. Prolog en un Exemple 19
   2.2. Programmation Logique 21
      2.2.1. Syntaxe 21
      2.2.2. La sémantique déclarative 23
      2.2.3. La résolution SLD 24
      2.2.4. Interprétation Procédurale 28
   2.3. Prolog 29
      2.3.1. Prolog Pur 29
      2.3.2. La Coupure 30
      2.3.3. La Négation 31
      2.3.4. Les Prédicats à Effet de Bord 32
   2.4. Les Machines Abstraites pour Prolog 32
      2.4.1. L'organisation de la Mémoire et le Retour-Arrière 32
      2.4.2. Les Variables: Représentation et Etats 33
      2.4.3. La Pile Traînée 34
   2.5. Parallélisme en Programmation Logique 34
      2.5.1. Les Sources de Parallélisme 34
      2.5.2. Parallélisme Explicite et Implicite 35
      2.5.3. Modèles Théoriques et Multi-Séquentiels 36
      2.5.4. Le Parallélisme OU 37
      2.5.5. Le Parallélisme ET 37
      2.5.6. Le Parallélisme ET/OU 38
      2.5.7. Les Langages Logiques Gardés 39
   2.6. Conclusions 39
3. Chapitre 3 41
   3.1. Les Principes 41
   3.2. La Condition d'Existence du Parallélisme OU 42
   3.3. La Gestion des Contextes Multiples 43
      3.3.1. La Gestion de la Mémoire 43
      3.3.2. Le Maintien de la Cohérence des Sections Communes 45
   3.4. La Copie des Sections Communes 47
      3.4.1. La Copie Simple de Piles 47
      3.4.2. La Copie avec Datation 47
   3.5. Le Partage des Sections Communes 47
      3.5.1. Le Tableau de Liaisons 47
      3.5.2. Les Liaisons Privilégiées 49
      3.5.3. Les Tables d'Adressage Associatif 49
      3.5.4. La Liste de Liaisons et l'Historique 50
      3.5.5. La Fermeture d'Environnements 51
   3.6. L'Ordonnancement 51
      3.6.1. Les Conditions Minimales du Parallélisme Efficace 51
      3.6.2. La Régulation de la Charge 54
      3.6.3. Quelques Méthodes de Régulation de Charge 55
   3.7. Les Opérateurs Séquentiels de Prolog 57

3.7.1. La Coupure 57
3.7.2. Les Effets de Bord 59
3.8. Les Résultats 60
3.9. Conclusions 61
4. Chapitre 4 63
4.1. Le Projet 63
4.1.1. Objectifs et Choix Initiaux 63
4.1.2. Hypothèses de Travail 63
4.1.3. Décisions Techniques Importantes 65
4.1.4. Résumé des Caractéristiques Principales 65
4.2. Les Constituants Séquentiel et Parallèle 66
4.2.1. La partie Séquentielle 66
4.2.1.1. Le Langage 66
4.2.1.2. Compilation et Machine Abstraite 66
4.2.1.3. Les Optimisations d'Opera (TWAM) 67
4.2.2. Le Modèle Opera 67
4.2.2.1. Schéma de Coopération Parallèle 67
4.2.2.2. Le Problème de la Gestion des Contextes Multiples 67
4.3. La Gestion de Contextes Multiples 68
4.3.1. Principes 68
4.3.1.1. Méthode Naïve 68
4.3.1.2. Copie Incrémentale 69
4.3.1.3. Objectif d'une Implantation 69
4.3.2. Problèmes 71
4.3.2.1. Liaison et Copie Incrémentale 71
4.3.2.2. Gestion de la Mémoire et la Pile de Variables 72
4.3.2.3. Liaison Superficielle/Liaison Profonde 73
4.3.2.4. Variable Conditionnelle/Inconditionnelle 73
4.3.2.5. Gestion des Points de Choix et la Pile Locale 74
4.3.2.6. Bilan 74
4.3.3. La Méthode Pile de Variables Plus Datation (VD) 75
4.3.3.1. Organisation de la Mémoire 75
4.3.3.2. Allocation et Initialisation de Variables 76
4.3.3.3. Liaison et Déréférencement 78
4.3.3.4. Déliaison Séquentielle 79
4.3.3.5. Récupération de l'Espace dans les Piles de Variables 79
4.3.3.6. Datation 79
4.3.3.7. Installation de Tâches 79
4.3.4. La Méthode Pile de Variables et Traînée plus Valeur (VV) 81
4.3.4.1. Organisation de la Mémoire 82
4.3.4.2. Traitement des Variables 82
4.3.4.3. Installation de Tâches 82
4.3.5. Bilan Datation ou Traînée plus Valeur (VD/VV) 84
4.3.6. Contrôle des Points de Choix Communs 85
4.3.6.1. Enregistrement des Liaisons de la Dernière Branche 86
4.3.6.2. Maintien des Points de Choix Communs 87
4.3.7. Cohérence des Données 88
4.3.7.1. Les Sources d'Incohérence 88
4.3.7.2. Les Remèdes 90
4.4. La Coupure 92
4.4.1. Le Problème 92
4.4.2. Les Principes de la Méthode Opera 94
4.4.3. Registres de la TWAM 94
4.4.4. Instructions de la TWAM et Compilation 94
4.4.5. La Synchronisation avec l'Exportation 95
4.4.6. Autres Méthodes 96
4.5. L'ordonnancement des Tâches 97

4.5.1. Classification selon la Charge 98
4.5.2. Evaluation de la Charge 98
4.5.3. Conditions Minimales du Parallélisme 99
4.5.4. Régulation de Charge 99
4.6. L'architecture du Logiciel: Processus et Protocole 100
4.6.1. Organisation en Processus 100
4.6.2. L'Ordonnanceur 102
4.6.3. L'Espion 103
4.6.4. Le Calculateur 104
4.6.5. L'Exportateur 104
4.7. Conclusions 104
5. Chapitre 5 107
5.1. Le Supernode 107
5.2. Le Placement de l'Architecture Opera sur le Supernode 110
5.3. L'adaptation du Modèle Opera au Supernode 111
5.3.1. La Gestion des Contextes Multiples 112
5.3.2. L'ordonnancement 113
5.3.3. Le Format des Messages inter Processus 113
5.4. Les Programmes de Test 114
5.5. Les Résultats en Séquentiel 115
5.6. Les Résultats en Parallèle 115
5.7. Le Logiciel de Développement 116
5.8. Conclusions 117
6. Chapitre 6 119
6.1. Résumé 119
6.1.1. Prolog et Parallélisme 119
6.1.2. Le Modèle OU Multi-séquentiel 119
6.1.3. Gain d'Efficacité 120
6.1.4. Le Modèle Opera 120
6.1.5. Le Prototype Opera 122
6.2. Le Problème de la Mémoire et de la Complétude 122
6.3. Opera: Critiques et Perspectives 123
A. Annexe A 125
A.1. Modules 125
A.2. Les Prédicats Prédéfinis 125
A.3. La Coupure 127
B. Annexe B 129
B.1. Les Eléments de Base de la Machine Abstraite (WAM) 129
B.1.1. Les Termes de la WAM 129
B.1.2. L'organisation de la Mémoire 130
B.1.3. Les Registres 130
B.1.4. Les Instructions 131
B.2. La Compilation vers la WAM 131
B.2.1. La Compilation de Prédicats 131
B.2.2. La Compilation de Clauses 134
B.2.3. Résumé de l'Exécution d'un Prédicat 134
B.3. Les Optimisations de la WAM 135
B.3.1. Occurrences de Variables 135
B.3.2. Optimisation d'Appel Terminal 135
B.3.3. Allocation de Variables 136
B.3.4. Globalisation de Variables Locales 136
B.4. Un Exemple 136
B.5. Les Optimisations d'Opera (TWAM) 137
B.5.1. Les Termes 137
B.5.2. Listes 138
B.5.3. La Disjonction 138
B.5.4. Mise à Jour de l'Etat sur Echec 139

B.5.5. La Coupure....139
B.5.6. Indexation....139
C. Annexe C....141
C.1. Introduction....141
C.2. Implantation....143
C.2.1. Le mécanisme de coupure de la WAM....143
C.2.2. La Compilation de la Coupure....144
C.2.2.1. Principe de Compilation....144
C.2.2.2. Coupure Locale....144
C.2.2.3. Coupure Globale....145
C.2.2.4. Coexistence de la Double Sémantique de la Coupure....146
C.3. Extensions....147
C.3.1. Le Snip (Arity_Prolog)....147
C.3.2. La Coupure Faible....147
C.3.2.1. Objectif....147
C.3.2.2. Réalisation....148
C.3.2.3. Coupure Faible Locale/Globale....150
C.3.2.4. Coexistence des Coupures Fortes/Faibles....150
C.4. Conclusion....150
D. Annexe D....151
D.1. Nouvelles Instructions....151
D.2. Instructions Modifiées....153
E. Annexe E....155
E.1. Les Langages et les Logiciels de Développement....155
E.2. Les Modules....155
E.3. La Structure de la Mémoire....155
E.4. Les Registres....156
E.5. Les Données....157
E.6. Les Instructions de la TWAM....158
E.7. Le Module d'Initialisation (init)....158
E.8. Les Prédicats Pré-Définis (builtin)....158
E.9. Les Exceptions et la Mise au Point (debug)....158
E.10. L'appel des Routines....159
Bibliographie:....161

Liste des Figures

Fig. 2.1 - Programme exemple de la résolution SLD 24
Fig. 2.2 - Arbre OU fini 27
Fig. 2.3 - Arbre OU infini 28
Fig. 2.4 - Mémoire simplifiée d'une MAP 33
Fig. 2.5 - Graphe d'états d'une variable 34
Fig. 2.6 - Liaisons inconditionnelle et conditionnelle 35
Fig. 3.1 - Récupération d'environnement commun 44
Fig. 3.2 - Récupération totale de la section commune 45
Fig. 3.3 - Liaisons multiples (a) 46
Fig. 3.4 - Liaisons multiples (b) 46
Fig. 3.5 - Datation 48
Fig. 3.6 - Tableau de liaisons 48
Fig. 3.7 - Installation avec tableau de liaisons 49
Fig. 3.8 - Liaison privilégiée 50
Fig. 3.9 - Cas des conditions minimales 53
Fig. 3.10 - Coupure conditionnée 58
Fig. 4.1 - Sections communes 70
Fig. 4.2 - Liaison avec pile de variables et datation 78
Fig. 4.3 - Schéma de l'installation VD 80
Fig. 4.4 - Liaison conditionnelle dans la méthode VV 82
Fig. 4.5 - Points de choix communs 83
Fig. 4.6 - Schéma de la méthode VV 84
Fig. 4.7 - Destruction de section commune 85
Fig. 4.8 - Règles d'ordonnancement 86
Fig. 4.9 - Transfert de liaison incohérente 90
Fig. 4.10 - Transfert de liaison cohérente 91
Fig. 4.11 - Inhibition totale et inhibition partielle 93
Fig. 4.12 - Coupure globale dans une disjonction 94
Fig. 4.13 - Exécution de la coupure en parallèle 96
Fig. 4.14 - Organisation en processus d'Opera 102
Fig. 5.1 - Configuration du Tnode à 16 Transputers 108
Fig. B.1 - Code TWAM pour sous-but et clause 137
Fig. B.2 - Etat des piles et des registres 138
Fig. E.1 - Organisation de la mémoire de l'émulation de la TWAM 156

Liste des Tables

Table 3.1 - Résultats de Aurora, Muse et PEPSys 60
Table 5.1 - Résultats en séquentiel 116
Table 5.2 - Les résultats sur le Tnode 117

1. Chapitre 1
Introduction

1.1. Le Thème

Le thème général de cette thèse est l'étude et l'évaluation de l'implantation du langage Prolog sur des architectures parallèles. Pendant ce travail nous avons restreint notre recherche au parallélisme OU implicite, implanté sur des machines avec un haut degré de parallélisme et sans mémoire commune.

1.2. Justification

Parmi les principaux buts de la recherche en Génie Informatique se trouvent:

- la recherche de puissance;
- la facilité de programmation.

a) La difficile maîtrise du parallélisme

Dans la recherche de puissance, le parallélisme a toujours été considéré comme une bonne voie de recherche. Cet intérêt a produit de nouvelles architectures de machines offrant un excellent rapport prix/performance, comme par exemple: Sequent, Alliant, Butterfly, Encore, Intel Hypercube et autres ([HER 86c], [DUN 90]).

Ces architectures revendiquent deux avantages sur l'architecture séquentielle:

- une façon de contourner les limites technologiques actuelles;
- une augmentation de performance pour une augmentation correspondante de prix.

Ce dernier point est dû à la régularité de l'architecture, permettant l'ajout aisé de processeurs.

L'utilisation effective de ces machines est cependant limitée du fait de l'absence de logiciel de base et de langage de programmation appropriés. Cette inadéquation provient de ce que systèmes ou langages sont de simples adaptations de ceux utilisés sur les machines classiques. L'utilisation "à bon escient" du parallélisme est donc intégralement à la charge du programmeur. Cette tâche délicate entraîne des coûts de développement trop importants.

b) Le problème de l'efficacité de la programmation logique

Si l'on s'intéresse à la facilité de programmation, le langage Prolog est un bon candidat dans un certain nombre de domaines:

- traitement des langages formels ou naturels;
- expression de gros problèmes combinatoires;
- systèmes experts;
- prototypage rapide.

Son utilisation comme langage "à tout faire" en informatique bute sur un problème d'efficacité. En [REI 88] et [PAA 88] des études montrent un facteur de 1 à 10 entre les temps d'exécution de Prolog et ceux des langages C ou Pascal. Plusieurs techniques d'optimisation ont contribué à diminuer ce facteur, mais la différence persiste. La source de cette inefficacité provient de sa nature "applicative" ou "non-séquentielle" qui rend son exécution (et sa compilation) délicate.

c) La programmation logique parallèle

Ainsi, nous voici face à un double problème:

- les machines parallèles existent et sont viables, mais leur emploi est limité par l'absence de langages de programmation parallèle;
- Prolog est un bon langage de programmation, mais son utilisation est limitée par son inefficacité séquentielle.

Toutefois, la nature "non-séquentielle" de Prolog se traduit par la présence de multiples opportunités de parallélisation. En conséquence un programme Prolog pourrait être exécuté de façon parallèle faisant ainsi:

- de Prolog, une solution à la maîtrise du parallélisme;
- du parallélisme, un remède à l'inefficacité séquentielle de Prolog.

C'est l'objectif de notre travail, comme de celui de beaucoup d'autres équipes.

1.3. Contexte Mondial

Divers centres de recherche travaillent actuellement dans le même domaine, c.a.d. la programmation logique et le parallélisme. Certains d'entre eux examinent plutôt les modèles d'architecture, d'autres préfèrent les questions relatives au langage. Certains privilégient un type spécifique de parallélisme (les types sont présentés dans le chapitre 2). Plusieurs défendent des modifications dans la définition classique de Prolog, parfois en introduisant des mécanismes de contrôle explicite du parallélisme, parfois en proposant un nouveau langage.

Entre les différents projets qui, isolément ou comme partie importante d'un projet plus général, recherchent des solutions aux problèmes posés par l'implantation parallèle d'un langage de programmation logique, nous pouvons citer:

- le projet GIGALIPS [LUS 88] qui concerne principalement les centres de recherche suivants:
  - Argonne National Laboratory, USA;
  - University of Manchester, Angleterre;
  - Swedish Institut of Computer Science, Suède.
- le projet "Fifth Generation Computer Project" qui est conduit par l'ICOT Research Center, au Japon [KUR 88];
- le projet PEPSys, Parallel ECRC Prolog System, en développement depuis 1985 à l'ECRC (European Computer-Industry Research Center), en Allemagne [BAR 88].

1.4. Le Contexte Local

Les travaux et les résultats de cette thèse font partie du projet Opera, conduit par l'équipe Flop, du Laboratoire de Génie Informatique. Ce laboratoire fait partie de l'IMAG (Institut de l'Informatique et Mathématiques Appliquées de Grenoble), groupement de laboratoires communs à l'INPG (Institut National Polytechnique de Grenoble) et à l'UJF (Université Joseph Fourier).

Les recherches de l'équipe Flop se concentrent sur la programmation logique, les systèmes d'exploitation et les outils de développement, principalement dans le cadre du parallélisme.

Le projet Opera ([BRI 90a], [BRI 90b], [BRI 91]) est l'implantation de Prolog pur sur le Supernode, une machine parallèle sans mémoire commune. Actuellement, dans une première phase du projet, seul le parallélisme du type OU a été étudié. Dans cette phase, les tâches suivantes ont été réalisées par les membres de l'équipe Flop:

- la définition du modèle d'exécution;
- la définition de la machine abstraite pour la compilation de Prolog;
- la définition et l'implantation des outils de base pour le développement:
  - un assembleur du transputer, processeur du Supernode;
  - un éditeur de liens;
  - un chargeur et metteur au point pour les cartes à transputer;
  - un compilateur C pour le transputer avec des extensions pour le parallélisme;
  - les bibliothèques de fonctions C;
  - un serveur pour le Supernode: chargeur, communication, entrée/sortie.
- l'adaptation d'un compilateur Prolog à la machine abstraite d'Opera;
- l'implantation (émulation) de la machine abstraite sur le transputer;
- l'implantation d'un premier prototype d'Opera sur le Tnode, la version bas de gamme du Supernode.

1.5. Contribution de l'Auteur

Au moment où l'auteur a commencé son travail de thèse, le projet Opera avait déjà débuté. Divers choix avaient été faits:

- la définition du type de parallélisme à étudier dans la première phase;
- le choix de la machine parallèle;

et diverses tâches étaient en cours:

- la définition du modèle d'exécution;
- la définition de la machine abstraite pour l'implantation de Prolog.

L'auteur a participé à la définition du modèle d'exécution et de la machine abstraite, à l'adaptation d'un compilateur Prolog à la nouvelle machine abstraite, à l'implantation de l'émulation de cette machine, et à l'implantation d'Opera sur la machine parallèle cible.

1.6. La Structure de la Thèse

Le deuxième chapitre consiste en:

- une présentation succinte du modèle de la programmation logique, ainsi que du langage Prolog;
- une introduction au parallélisme en programmation logique: sources, types et modèles.

Dans le chapitre trois, le **modèle OU multi-séquentiel** de parallélisme en Prolog est étudié de façon plus approfondie. Ceci se justifie par le fait que le système Opera, sujet de cette thèse, est un exemple de ce modèle. Nous décrivons les principes de ce modèle, et nous abordons les problèmes principaux de son implantation, ainsi que les solutions les plus remarquables trouvées dans la littérature. Une attention spéciale est donnée à la régulation de charge, un problème crucial dans le cas du modèle multi-séquentiel OU.

Le chapitre quatre est le cœur de cette thèse. Il se traduit par une description détaillée du système Opera, qui comprend:

- les caractéristiques principales du projet Opera: objectifs initiaux et choix de base;
- les principes du système Opera, et les problèmes principaux à résoudre dans une implantation;
- une étude de la gestion des contextes multiples, en proposant deux méthodes pour son traitement;
- une méthode efficace pour l'implantation de la coupure en parallèle;
- la régulation de charge, ici abordée du point de vue des caractéristiques du système Opera;
- l'architecture du logiciel Opera, en présentant les processus et le protocole de messages.

Il faut remarquer une caractéristique initiale du projet Opera: nous avons décidé de privilégier les architectures sans mémoire commune. Ceci affecte fortement certains aspects du système Opera, comme la gestion des contextes multiples et la régulation de charge.

Un résumé de l'implantation d'un prototype d'Opera, sur une classe particulière de machines sans mémoire commune (le Supernode), se trouve dans le chapitre 5. Les premiers résultats de sa performance y sont présentées.

La thèse se termine par les conclusions que nous avons pu tirer de ces études, et de l'implantation du parallélisme OU multi-séquentiel sur une machine sans mémoire commune.

2. Chapitre 2
Programmation Logique, Prolog et Parallélisme

Ce chapitre comprend deux parties. La première partie est une définition succincte du modèle de programmation logique ainsi que du langage Prolog, lequel en est la concrétisation principale. Nous pensons que ceci est nécessaire pour deux raisons:

- proposer à un lecteur qui ne connaît ni l'un ni l'autre, un minimum de concepts nécessaires à la compréhension du texte qui suit;
- bien préciser les concepts que nous lions aux expressions les plus souvent utilisées dans ce texte.

Outre ces raisons, cette partie permettra de préciser, dans la suite, les limites et les possibilités de Prolog par rapport à la programmation logique.

La deuxième partie est une introduction au parallélisme en Prolog: sources, types et modèles.

2.1. Prolog en un Exemple

Cette section introduit Prolog par un exemple simple.

La construction la plus simple en Prolog est le **fait**. Un fait est une affirmation, ou relation, qui est toujours vraie. Quelques exemples sont présentés, accompagnés de leur sens en langue naturelle:

```
pere(jacques, marie).          Jacques est le père de Marie
age(marie, 4).                 Marie a 4 ans.
profession(jacques, dentiste). Jacques est dentiste.
```

Un programme peut contenir un ensemble de faits du même type, ce qui correspond à une base de données.

L'utilisateur peut demander si un fait particulier est "vrai" ou "faux", c.a.d. s'il existe dans la base de données du programme. Cette demande s'exprime par un **but**. Par exemple, étant donné les faits ci-dessous,

```
pere(jacques, marie).
pere(jacques, philippe).
pere(philippe, jean).
```

et le but suivant,

```
:- pere(jacques, philippe).    Jacques est-il le père de philippe?
```

l'exécution de ce programme Prolog répond *OUI*.

L'exécution répond *NON* pour le but

```
:- pere(jacques, martine).
```

L'utilisateur peut encore demander quels sont tous les enfants de Jacques en utilisant une **variable** (une variable commence par une majuscule), *Enfant* dans le but ci-dessous:

```
:- pere(jacques, Enfant).      De quel enfant, Jacques est-il le père?
```

L'exécution du programme répond:

```
Enfant = marie;
Enfant = philippe;
```

Le deuxième type de construction en Prolog est la **règle**. Celle-là permet de résoudre des problèmes plus complexes que le précédent. Une règle comporte deux parties: la **tête** et le **corps**. La tête est vraie si le corps est vrai. La tête a le même format qu'un fait, et le corps est une série de buts. Le programme précédent peut être augmenté par la règle suivante qui exprime la relation de parenté (l'une entre les possibles) entre un grand-père et son petit-enfant:

```
grandpere(GrandPere, PetitEnfant) :-
        pere(GrandPere, Pere),
        pere(Pere, PetitEnfant).
```

Par cette règle, un *PetitEnfant* est le petit-enfant d'un *GrandPere*, si celui-là est le père d'un *Pere* et si celui-ci est le père du *PetitEnfant*. *GrandPere*, *PetitEnfant* et *Pere* sont des variables.

On peut proposer le but

```
:- grandpere(jacques, PetitEnfantJacques).
```

qui correspond à "Est-ce que Jacques a des petits-enfants?", et auquel Prolog répond affirmativement:

```
PetitEnfantJacques = jean;
```

Finalement, Prolog offre la récurrence, qui permet d'exprimer des programmes plus complexes. Par exemple, au programme précédent on peut ajouter les deux règles suivantes, exprimant la relation d'ascendance entre deux personnes:

```
ascendance(Ascendant, Descendant) :-
        pere(Ascendant, Descendant).
ascendance(Ascendant, Descendant) :-
        pere(Ascendant, AscendantPlusProche),
        ascendance(AscendantPlusproche, Descendant).
```

Par ces règles, deux personnes ont une relation d'ascendance si soit l'*Ascendant* est le père du *Descendant* (première règle), soit l'*Ascendant* est le père d'un *AscendantPlusproche*, et celui-ci est un ascendant du *Descendant*. Au but ci-dessous

```
:- ascendance(AscendantsJean, jean).
```

Prolog répond:

```
AscendantsJean = philippe;
AscendantsJean = jacques;
```

Il est important d'observer que, en Prolog, on peut exprimer une fonction par une relation (règle), et qu'on peut l'exécuter de plusieurs façons. Ainsi, pour la fonction *grandpère(personne)*, dont la valeur est le grand-père d'une personne, et exprimée par la règle *grandpere(GrandPere, PetitFils)*, trois exécutions sont possibles:

- au sens classique: on demande les résultats (valeurs) à partir des arguments d'entrée: par exemple, le but *grandpere(GrandPere, jean)*;
- à l'envers: on demande les arguments d'entrée à partir des résultats: par exemple, le but *grandpere(jacques, PetitEnfantJacques)*;
- comme test: les arguments et les résultats sont fournis, et on demande une vérification de la relation entre eux: par exemple, le but *grandpere(jacques, jean)*.

Ainsi, un même programme permet de résoudre des problèmes différents.

2.2. Programmation Logique

La programmation logique est un modèle de programmation dérivé de la logique de prédicats du premier ordre. Celle-ci est l'un des formalismes utilisés en logique pour la représentation et le traitement d'axiomes et de propositions dans le domaine scientifique en général.

Les divers concepts utilisés en programmation logique seront présentés par la logique des clauses de Horn, une sous-partie de la logique de prédicats du premier ordre.

Une description formelle et détaillée de la programmation logique est présentée dans le texte de Lloyd ([LLO 88]). Les chapitres initiaux de [STE 86] et le chapitre 10 de [CLO 85] contiennent des introductions plus simples. Parmi les références sur la définition et les caractéristiques les plus importantes de la programmation logique, les articles [KOW 74] et [KOW 83] peuvent être cités, le premier étant considéré comme l'acte fondateur de la programmation logique.

2.2.1. Syntaxe

En général les auteurs utilisent une syntaxe bien proche de celle utilisée par la logique des prédicats du premier ordre. Afin de rendre ce texte moins long, nous utilisons une simplification de la syntaxe de C-Prolog, lequel est l'une des implantations les plus populaires de Prolog. Cette syntaxe est utilisée dans notre implantation de Prolog, à quelques exceptions près.

Variables:

Les variables représentent des objets arbitraires et s'écrivent comme des suites alphanumériques, qui commencent soit par une lettre majuscule soit par un sous-tiret (_).

Exemples: X, Y, Pere, GrandPere.

Constantes:

Une constante représente un objet distinct, tel qu'un entier ou un atome. Les entiers sont représentés de la façon usuelle. Les atomes sont représentés par des suites alphanumériques qui commencent par une lettre minuscule.

Exemples de entiers: 0, 1, 34, 2409, -56.

Exemples d'atomes: table, jacques, françoise.

Termes:

Un terme est:

- soit une variable;

- soit une constante;
- soit une expression, dite terme fonctionnel (ou composé ou structuré), de la forme $f(t_1, t_2, ..., t_n)$ où f est un symbole fonctionnel (représenté par une suite alphanumérique qui commence par une lettre minuscule), les sous-termes t_i sont des termes (définition récurrente), et n est l'arité du terme (nombre de sous-termes).

Exemple de termes structurés: liste(a, 3, X).

Les constantes sont aussi définies comme termes fonctionnels d'arité zéro.

Formule atomique:

Une formule atomique est une expression de la forme $p(t_1, t_2, ..., tm)$ où p est un symbole de prédicat à m positions (arité), et les t_i sont des termes. Le symbole de prédicat s'écrit comme une suite alphanumérique, débutant par une minuscule.

Exemples: pere(jacques, marcel), mere(marie, X).

Formule:

Une formule est soit une formule atomique soit une des expressions suivantes (à droite dans la liste ci-dessous) où F et G sont des formules:

- négation:	not(F)
- disjonction:	F; G
- conjonction:	F, G
- G implique F:	F :- G
- équivalence:	F <-> G
- quantificateur universel:	∀x F
- quantificateur existentiel:	∃x F

Littéral:

Un littéral est soit une formule atomique soit sa négation.

Clause de programme[1]:

Une clause de programme (clause dans la suite) est une formule de la forme

$$A :- B_1, B_2, ..., B_n.$$

où A est une formule atomique et les B_i sont des littéraux et toutes les variables de la clause sont quantifiées universellement (∀). A est la **tête** de la clause et la conjonction des B_i est le **corps**.

Exemple:

grandPere(X, Y) :- pere(X, Z), pere(Z, Y).

---

[1] Dans les clauses de Horn (but inclus), les sous-buts doivent être des formules atomiques.

Une **clause unitaire**, dénommée aussi **fait**, est une clause dont le corps est vide (*n* est égal à zéro), le symbole :- étant omis.

Exemple: pere(jacques, marcel).

Les clauses dont le corps n'est pas vide (*n* est égal ou supérieur à 1) sont dénommées **règles**.

But de programme:

Un but est une formule de la forme

$$:- B_1, B_2, \ldots, B_n.$$

où chaque B_i est un littéral dénommé **sous-but** et toutes les variables sont quantifiées de façon existentielle.

Prédicat:

Un prédicat est défini par un ensemble fini de clauses de programme dont la tête contient le même symbole de prédicat avec la même arité.

Exemple:

```
ancetre(X, Y) :- parent(X, Y).
ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y).
```

Programme:

Un programme est un ensemble fini de prédicats.

2.2.2. La sémantique déclarative

La sémantique d'un programme logique admet plusieurs définitions ([EMD 76]): déclarative, procédurale et dénotationelle. Cette section est une présentation informelle de la sémantique déclarative.

Les concepts de base de la sémantique déclarative, utilisée dans la programmation logique, sont l'interprétation et les modèles de Herbrand ([LLO 88], [STE 86]). Voici quelques définitions, nécessaires pour la suite:

Règle $(\forall X_i\ (A :- B_1, B_2, \ldots, B_n))$**:**

Pour toutes les valeurs de X_i, A est vrai si $B_1, B_2, \ldots,$ et B_n sont vrais. Pour prouver A, il est suffisant de prouver $B_1, B_2, \ldots, B_n$. Si A est faux au moins l'un des B_i est faux. Si l'un des B_i est faux A peut être faux ou vrai.

Fait $(\forall X_i\ (A))$**:**

Pour toutes les valeurs de X_i A est toujours vrai.

Programme:

Un programme est la spécification formelle d'une théorie dans un certain domaine. Les règles et les faits du programme représentent cette théorie.

But $((\exists X_i\ B_1, B_2, \ldots, B_n)$ **ou** $(\forall X_i$ **:-** $B_1, B_2, \ldots, B_n))$**:**

Existe-t-il des valeurs de X_i pour lesquelles $B_1, B_2, \ldots,$ et B_i sont vrais? Si le but ne contient pas de variables: est-ce que $B_1, B_2, \ldots, B_n$ sont vrais?

Clause vide:

Une clause vide est une contradiction.

Interpréteur d'un langage logique:

Etant donnés un programme et un but, l'interpréteur du langage logique doit essayer de vérifier la satisfaisabilité du but par rapport au programme. La résolution SLD, décrite dans la section suivante, est à la base d'un tel interpréteur.

2.2.3. La résolution SLD

La résolution SLD est une méthode de déduction d'assertions à partir d'axiomes exprimés sous la forme de clauses de Horn. Les trois lettres de SLD ont le sens suivant:

- D: la méthode est applicable aux clauses définies (de Horn);
- L: la méthode est linéaire, c.a.d. à chaque pas une seule formule est conservée;
- S: la méthode utilise des règles de sélection (choix) de formules.

Elle a été décrite, pour la première fois, par Kowalski ([KOW 74]). Désormais la résolution SLD est à la base du modèle d'exécution de la programmation logique. Elle est une simplification du principe de résolution proposé par Robinson ([ROB 65]) pour les clauses dans la forme normale, dont il est très difficile d'extraire une implantation efficace à cause d'un espace de recherche extrêmement vaste.

La résolution SLD utilise la technique de preuve par réfutation: pour démontrer qu'un but G est une conséquence d'un programme P, il est prouvé que la négation du but et le programme sont contradictoires. Dans le cas de la résolution SLD la négation de G est ajoutée au programme et, à partir de ce nouvel ensemble de clauses, on essaie de déduire la clause vide. L'avantage de la preuve par réfutation, dans le cas où G est une conséquence de P, est la certitude d'obtenir la clause vide en un nombre fini de pas à condition qu'un "bon choix" de clause soit fait (ce point sera revu par la suite). Par contre, en appliquant une méthode de déduction qui essaye de dériver G à partir de P, il est plus difficile dans le cas général de trouver le "bon chemin" (qui vérifie G en un nombre fini de pas), même si G est une conséquence de P.

Le programme et le but de la fig. 2.1 ([LLO 88]) seront utilisés pour illustrer les définitions présentées dans la suite.

```
:- p(X, b).                    % But G, un seul sous-but.

p(X, Z) :- q(X, Y), p(Y, Z).   % Programme P, avec 3 clauses.
p(X, X).

q(a, b).
```

Fig. 2.1 - Programme exemple de la résolution SLD

La résolution SLD utilise une opération complexe et spécifique, nommée unification, laquelle applique deux opérations plus simples: la substitution et l'exemplarisation. Ces trois opérations correspondent aux notions suivantes:

Substitution:

Une substitution est un ensemble fini de la forme $\{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ où v_i est une variable, t_i est un terme, v_i est différent de v_j pour tout i différent de j. Par exemple, $\{X/a, Y/b\}$ est une substitution pour les variables de la première clause de P (voir fig.2.1)

Exemplarisation (*instantiation*):

Etant donnés une substitution q et un littéral L, l'exemplarisation Lq est le nouveau littéral produit par le remplacement de chaque référence à v_i dans L par t_i si t_i/v_i appartient à q. D'une façon informelle l'exemplarisation est l'application d'une substitution q à un littéral L. Lq est un exemplaire de L. L'application de $\{X/a, Y/b\}$ au littéral $q(X, Y)$, $p(Y, Z)$ produit $q(a, b)$, $p(b, Z)$ (voir fig.2.1).

Unification:

Etant donnés deux littéraux, L_1 et L_2, une substitution θ est un **unificateur** de L_1 et de L_2 si $L_1\theta = L_2\theta$. Un littéral L_1 est **plus général** qu'un autre littéral L_2 si L_2 est un exemplaire de L_1. On peut dire qu'il existe une substitution θ_a telle que $L_2 = L_1\theta_a$. L'unificateur le plus général de deux littéraux L_1 et L_2 est une substitution θ telle que $L_1\theta = L_2\theta$ et que pour tout autre unificateur α de L_1 et L_2 il est possible de trouver une substitution β telle que $L_1\alpha = (L_1\theta)\beta$ (ou $L_2\alpha = (L_2\theta)\beta$).

Un algorithme d'unification de deux littéraux L_1 et L_2 produit l'unificateur le plus général de L_1 et de L_2 s'il existe, ou, dans le cas contraire, indique cette inexistence, c.a.d. un échec. L'algorithme d'unification doit éviter un cas particulier de substitution: pour $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ il n'existe pas d'occurrence de la variable v_i dans t_i. Ce contrôle s'appelle **vérification d'occurrence** (*occurs-check*). Dans l'exemple de la fig. 2.1 l'unificateur le plus général de $p(X, b)$ et $p(X, Z)$ est $\{Z/b\}$.

Il est maintenant possible de détailler la méthode de la résolution SLD.

Dérivation SLD:

- A: Etant donnés un programme P, composé par les clauses $C_1, C_2, \ldots, C_n$, et un but G, composé par les sous-buts $SG_1, SG_2, \ldots, SG_m$. k est initialisé à 0. Le but G est copié en R_k appelé résolvante.
- B: Un sous-but SG_i de R_k est choisi. Une clause C_j est aussi choisie parmi les clauses du sous-ensemble SP, constitué de clauses dont la tête s'unifie avec SG_i. Toutes les variables de C_j sont renommées de façon à ce que aucune variable de C_j n'apparaisse dans R_k. L'algorithme d'unification est appliqué à SG_i et à la tête de C_j, en produisant l'unificateur θ_k. Une nouvelle résolvante R_{k+1} est produite, à partir de R_k, en remplaçant SG_i par le corps de C_j et par l'application de θ_k. k est incrémenté. L'étape B est répétée.

La dérivation ci-dessus peut avoir trois résultats:

- succès: R_k est vide. Dans ce cas une **réfutation SLD** a été trouvée, c.a.d. la satisfaisabilité du but G a été vérifiée;
- échec: SP est vide;
- boucle infinie: à chaque itération ni R_k ni SP sont vides.

Un exemple de dérivation SLD avec succès, pour la fig. 2.1, contient les étapes suivantes:

- choix du sous-but (un seul) $p(X, b)$;
- choix de la clause $p(X_1, Z_1)$:- $q(X_1, Y_1), p(Y_1, Z_1)$. (les indices indiquent le renommage des variables);
- θ_0 est $\{Z_1/b\}$;
- R_1 est $q(X_1, Y_1), p(Y_1, b)$. ;
- unification de $q(X_1, Y_1)$ et $q(a, b)$, en produisant $R_2 = p(b, b)$. et $\{X_1/a, Y_1/b\}$;
- unification de $p(b, b)$. et $p(X_3, X_3)$ (deuxième clause de $p/2$) en produisant R_3 = vide, ce qui correspond à un succès ou à la preuve de $p(X, b)$.

Dans chaque itération deux choix sont faits: celui du sous-but SG_i et celui de la clause C_j. Le premier, appelé **règle d'évaluation**, n'affecte pas le résultat. Si un succès est produit pour un ordre donné d'exécution des sous-buts, il est possible de produire le même succès pour tous les autres ordres. Ceci est parfois appelé **l'indépendance de la règle d'évaluation** ([LLO 88]): on peut fixer la règle d'évaluation a priori. En outre, tous les sous-buts doivent être exécutés afin de produire un succès.

Par contre, le choix de la clause, ou **règle de sélection**, est important. Selon la règle de sélection, on peut, pour un même programme et un même but, produire l'un quelconque des trois résultats (succès, échec, boucle infinie). Une seule clause est suffisante afin de produire un succès, à condition d'effectuer le "bon choix". La difficulté provient du fait que le "bon choix" n'est pas évident, car il est indécidable.

En considérant que l'utilisateur s'intéresse aux (au moins une) réfutations et que le "bon choix" de la clause n'est pas évident, il faut prévoir une **stratégie (règle) de recherche** en cas d'échec:

- quel sous-but précédent (déjà unifié) choisir?;
- quelle nouvelle clause choisir (re-application de la règle de sélection, en éliminant la clause qui a échouée)?

Cette action est nommée **retour-arrière** dans les stratégies séquentielles. La combinaison d'une règle d'évaluation et d'une stratégie de recherche (règle de sélection incluse) est nommée **procédure de réfutation SLD** ([LLO 88]). Il faut observer que, pour un programme et un but et en considérant toutes les clauses, certaines règles d'évaluation produisent des boucles infinies (en plus des succès et des échecs), pendant que d'autres n'en produisent pas. La règle de sélection devient donc importante, parce qu'on ne trouve pas certains succès si les boucles ne sont pas détectées. Des exemples de procédures de réfutation SLD sont présentés plus loin (voir les figures 2.2 et 2.3), en utilisant la représentation en arbre OU.

Il est prouvé que le but G est en contradiction avec le programme P si, en fixant une règle d'évaluation, tous les choix de clauses (toutes les dérivations SLD) amènent à des échecs. La négation de Prolog, ou **négation par échec**, définie dans la suite, applique le même procédé.

Notion d'arbre ET/OU:

La résolution SLD peut être représentée par un arbre composé de nœuds ET et OU et de feuilles Echec et Succès. Chaque nœud ET correspond à une sélection de sous-but et contient la résolvante courante. La racine contient le but originel et est un nœud ET. Chaque branche ET (qui part d'un nœud ET) correspond au choix d'un sous-but. A la pointe d'une branche ET il peut y avoir:

- une feuille Echec: il n'y a pas de clause qui puisse être unifiée avec le sous-but;
- un nœud OU: il groupe toutes les clauses qui peuvent être unifiées avec le sous-but respectif.

Chaque nœud OU correspond à une sélection de clause et contient le sous-but choisi précédemment. Chaque branche OU correspond au choix d'une clause dont la tête sera unifiée avec son sous-but. A la pointe d'une branche OU il peut y avoir:

- succès: la clause respective est un fait et la résolvante courante contient un seul sous-but;
- un nœud ET: il contient la nouvelle résolvante où le sous-but du nœud OU précédent a été remplacé par le corps de la clause respective.

La satisfaisabilité du but originel est prouvée s'il y a au moins une feuille Succès. La satisfaisabilité du but originel ne peut pas être prouvée s'il n'y a pas de feuilles Succès.

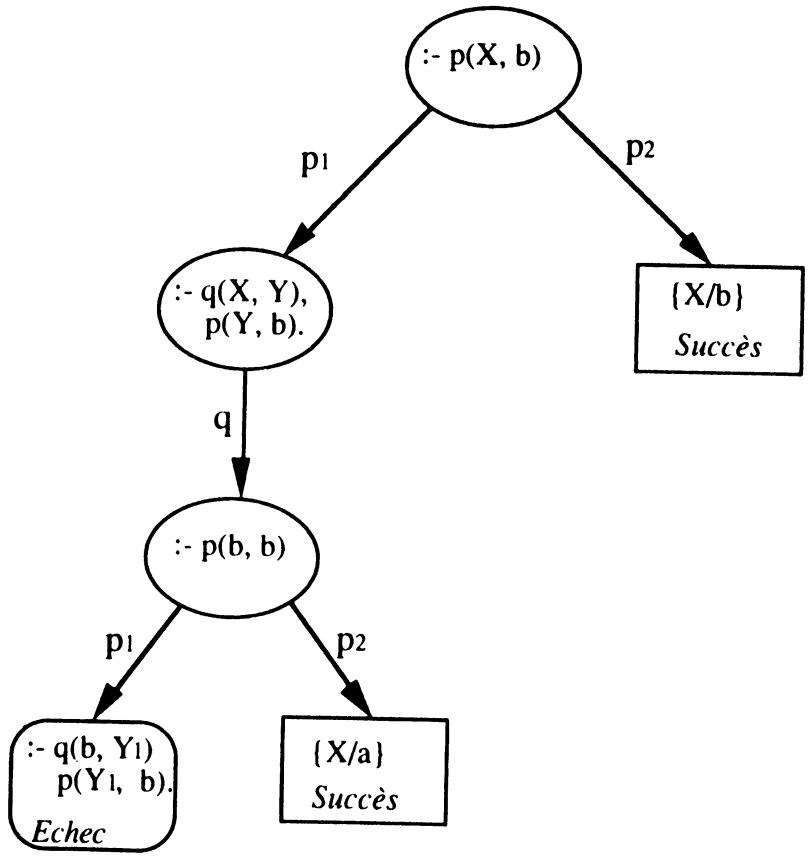

Fig. 2.2 - Arbre OU fini

Etant donné que la règle d'évaluation n'affecte pas les résultats, on peut maintenir une seule branche par nœud ET. En général les nœuds ET sont omis soit en figeant une règle d'évaluation soit en annotant le sous-but choisi dans chaque nœud OU. L'arbre ET/OU se transforme en un arbre OU.

Les figures 2.2 et 2.3 sont des exemples de représentations en arbre OU de deux différentes procédures de réfutation SLD, appliquées au programme de la fig. 2.1. La première utilise une règle d'évaluation qui choisit toujours le sous-but le plus à gauche. Cet arbre ne possède pas de boucles infinies. La règle d'évaluation de la fig. 2.3 choisit toujours le sous-but le plus à droite, en produisant des boucles infinies. Les deux possèdent le même nombre de succès.

2.2.4. Interprétation Procédurale

L'interprétation procédurale ([KOW 74], [KOW 79]) est une analogie entre la résolution SLD et les concepts classiques des langages de programmation.

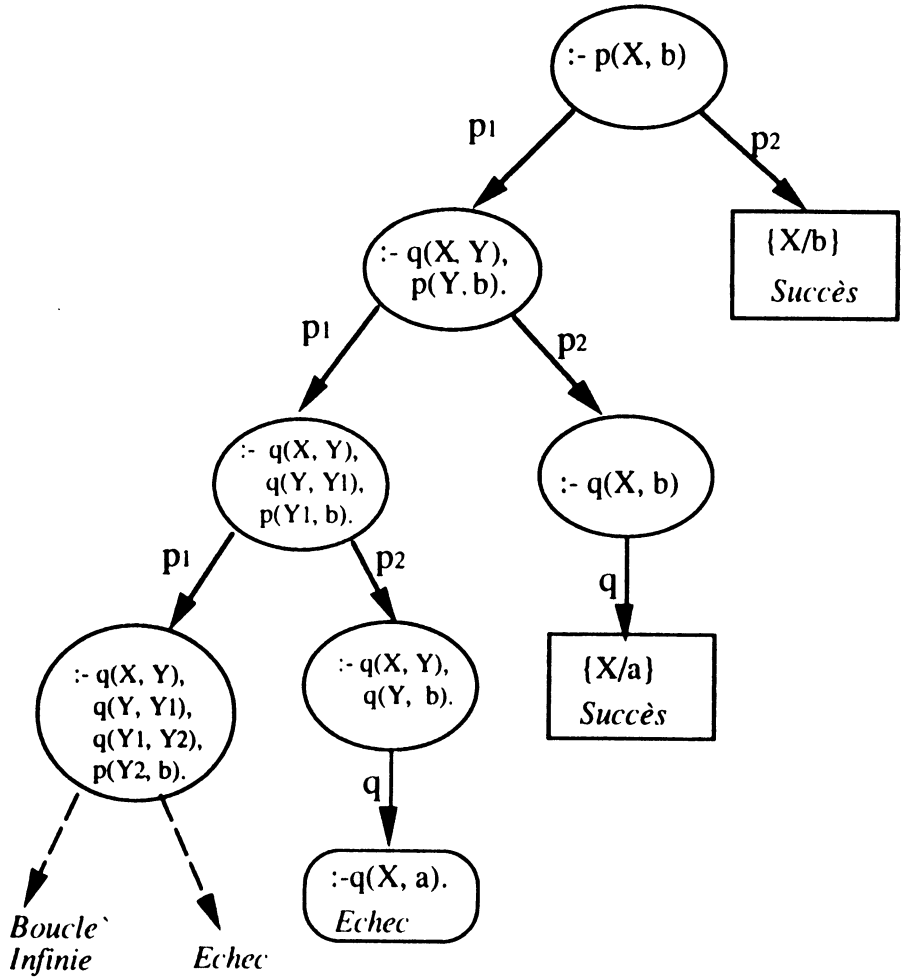

Fig. 2.3 - Arbre OU infini

Les points principaux de cette analogie sont:

Procédure:

Une procédure est l'ensemble des clauses d'un programme, dont les têtes ont le même symbole de prédicat et la même arité. Le nom de la procédure est formé par *symbole/arité*. Chaque clause est une définition (alternative) de la procédure. Le corps d'une définition est composé exclusivement d'appels de procédures.

Appel de procédure:

Chaque littéral du corps d'une clause et du but est un appel de procédure.

Arguments:

Les arguments d'un appel sont tous les termes de chaque littéral.

Unification:

L'unification est interprétée comme une seule opération qui englobe:

- passage de paramètres: une variable de la tête de la clause est unifiée à un sous-terme (constant ou non) d'un argument;
- retour de résultats: une variable d'un argument est unifiée à un sous-terme (constant ou non) de la tête de la clause.

Exécution du programme:

La solution d'un problème G (but) par un programme P passe par l'exécution de chaque sous-but G_i de G. Pour exécuter chaque G_i il est nécessaire de résoudre tous les sous-buts B_j d'une clause dont la tête s'unifie avec le sous-but G_i. L'interprétation procédurale impose la contrainte suivante: pour exécuter le sous-but G_{i+1} d'un but G, il ne doit plus rester de sous-buts B_j non-exécutés du sous-but G_i précédemment choisi. L'exécution d'un sous-but, G_i ou B_j, s'achève si la clause avec laquelle il est unifié possède un corps vide.

Représentation en arbre OU:

Chaque nœud OU représente la liste d'appels à exécuter, en explicitant le premier (règle d'évaluation). Chaque branche correspond à une définition de procédure dont le nom concorde avec l'appel. Une feuille de succès indique une solution au problème initial. Une feuille d'échec est une alternative de calcul qui ne produit pas de solution.

2.3. Prolog

Cette section présente le langage Prolog. Nous essayons de mettre en évidence les différences par rapport au modèle de la programmation logique, et les caractéristiques qui posent des problèmes pour une implantation parallèle.

2.3.1. Prolog Pur

Le langage Prolog (PROgrammation en LOGique) est une concrétisation du modèle de programmation logique décrit dans les sections précédentes. Prolog a été défini par A. Colmerauer au début des années 70. Les études de Colmerauer portaient sur les langues naturelles (grammaires, traduction automatique), et un langage adapté à ce type de problème fut défini et réalisé ([COL 72]). Ce langage fut définitivement fondé avec la rencontre avec les travaux de Kowalski sur la logique ([COH 88], [VAN 86]).

L'exécution d'un programme Prolog "pur" est conforme à une mise en œuvre de la résolution SLD, appliquée à un ensemble de clauses et un but initial. Cette mise en œuvre, ou procédure de réfutation SLD, est définie par les règles suivantes :

- R1: la règle d'évaluation choisit toujours, et seulement, le premier sous-but à gauche;
- R2: la règle de sélection a deux parties:
  - Prolog choisit toujours la première clause, dans l'ordre d'écriture, unifiable avec le sous-but choisi;
  - après un succès ou un échec, l'exécution reprend le nœud OU le plus récent, contenant encore des clauses non exécutées. La prochaine clause de ce nœud est choisie.

- R3: dans la nouvelle résolvante le corps de la clause est mis à gauche, c.a.d. la clause sélectionnée sera exécutée avant les autres sous-buts de la résolvante antérieure.

Les règles R1 et R3 coupent tous les sous-arbres dérivés d'un nœud ET de l'arbre ET/OU, à l'exception d'une branche, celle qui correspond au premier sous-but à gauche de la résolvante. Ceci permet la transformation de l'arbre ET/OU en un arbre OU. Le deuxième point de la règle R2 correspond au concept de retour-arrière. Toutes ces règles correspondent à un parcours **en profondeur d'abord** (R1 et R3) et **de gauche à droite** (R2) de l'arbre OU. Le parcours en profondeur d'abord équivaut à l'interprétation procédurale.

Cette stratégie, en séquentiel, ne produit pas toutes les solutions (succès) dans les cas de programmes dont l'arbre OU contient des branches infinies. Cependant elle est simple et elle consomme peu de mémoire. Elle contraste avec la stratégie **en largeur d'abord**, où on essaye d'avancer "en parallèle" toutes les branches de l'arbre OU. Cette deuxième stratégie évite les impasses que sont les arbres infinies pour la stratégie en profondeur. Par contre, elle exige un espace en mémoire plus grand, et son contrôle est plus complexe.

Il est possible de résoudre divers problèmes académiques avec Prolog, réduit à la logique de clauses de Horn. Cependant, pour la programmation d'applications réelles, il a été nécessaire d'introduire des prédicats prédéfinis (calcul arithmétique, entrée/sortie, ...) dans le langage. Plusieurs d'entre eux ne peuvent pas recevoir une interprétation purement logique. Dans les prochaines sections quelques-uns de ces prédicats seront analysés, en particulier ceux dont la sémantique et/ou l'implantation sont affectées par le parallélisme.

2.3.2. La Coupure

L'exécution d'un prédicat Prolog effectue une recherche exhaustive et produit en général diverses solutions. Si pour chaque valeur d'un argument d'appel, une seule solution est possible, un tel prédicat est **déterministe**. Une fois la solution trouvée, il est inutile de continuer à en chercher d'autres (suppression de calculs inutiles, qui produisent des échecs). Parfois, pour des prédicats non-déterministes, il n'est pas utile de chercher d'autres solutions après qu'une première ait été trouvée. Dans le but de réduire le temps et la mémoire nécessaires à la recherche de ces solutions inutiles ou inexistantes, un prédicat prédéfini, nommé **coupure** (représenté par *!*), a été introduit.

La sémantique de la coupure est la suivante: étant donné une clause C_i d'un prédicat p de la forme

A_1 ...
...
A_i :- B_1, ..., B_j, !, B_{j+1}, ..., B_k % *clause* C_i
A_{i+1} ...
...
A_n

l'exécution de *!* coupe toutes les alternatives restantes parmi celles créées à partir de l'appel du prédicat p, c.a.d. les clauses qui suivent C_i (A_{i+1} à A_n) dans la définition de p et les alternatives créées par l'exécution de B_1 à B_j.

Certains emplois de la coupure n'affectent pas la lecture déclarative (purement logique) de chaque clause. Ce type d'emploi est nommé coupure **verte**. Son seul objectif est l'efficacité: éviter la recherche d'autres solutions valides ou d'échecs. Si la coupure est enlevée le programme produit les résultats corrects. L'exemple classique est le prédicat *maximum/3* qui retourne le maximum de deux nombres. La coupure $!_1$ est verte parce que la deuxième clause peut être lue de façon déclarative.

```
maximum(X, Y, X) :- X ≥ Y, !1.
maximum(X, Y, Y) :- X < Y, !2.
```

Par contre, dans d'autres emplois, nommés coupure **rouge**, la seule lecture possible est celle définie par l'ordre de déclaration de clauses. En général cette restriction est due à l'omission des conditions qui valident les clauses suivantes à la coupure. Elles ne doivent être exécutées que si la clause qui contient la coupure échoue avant la coupure. Si la coupure est enlevée et la clause n'échoue pas, les clauses suivantes produiront des solutions invalides. La coupure *!1* de *maximum/3* devient rouge, si la condition $X < Y$ de la deuxième clause est enlevée.

```
maximum(X, Y, X) :- X ≥ Y, !1.
maximum(X, Y, Y).
```

2.3.3. La Négation

Le modèle de programmation logique décrit dans les sections précédentes ne comporte pas la négation. Les faits négatifs sont simplement omis. Par exemple, dans le cas du programme exemple de la première section, il n'est pas possible de déclarer, en utilisant le prédicat *pere/2*, que *jean* n'est pas le père de *marie*.

La solution adoptée pour l'expression de la négation est le concept de **négation par échec**. La négation d'un but est vrai s'il n'est pas possible de prouver le but. Dans le cas des faits, si le but-fait n'est pas déclaré dans le programme, alors la négation de ce but est vrai.

Dans le contexte de la résolution SLD, étant donné un but *not(B)*, la solution ci-dessus implique un parcours de toutes les alternatives du but *B* afin de vérifier qu'il n'existe aucune solution. A ce moment *not(B)* est considéré vrai, *not(B)* est éliminé de la résolvante et aucune substitution pourvue par *B* n'est conservée. La négation par échec ne peut être utilisée que lorsque l'arbre de recherche du but nié ne possède pas de boucles infinies. En effet, la présence d'une seule branche infinie implique la non-terminaison de la négation.

Notons que la notion de négation par l'échec ne correspond pas exactement à la négation logique. Par exemple, la solution du but

```
:- not(p(X)), q(X).
```

et des faits:

```
p(a).
p(b).
q(c).
```

est dépendant de l'ordre de choix du sous-but:

- échec si *not(p(X))* est sélectionné avant *q(X)*: *p(a)* est une solution à *p(X)* donc *not(p(X))* est faux;
- succès si *q(X)* est sélectionné avant *not(p(X))*: le sous-but *p(c)* n'a pas de solution.

Par contre, le choix de clauses n'affecte pas les solutions parce que toutes les clauses doivent être examinées pour confirmer la négation.

L'implantation classique de la négation est réalisée au moyen de la coupure. Le prédicat (prédéfini dans le langage) *not/1* peut se décrire par:

```
not(p) :- p, !, fail.
not(p).
```

Si *p* est prouvé, la deuxième clause de *not/1* et les autres alternatives de *p* sont coupées et *not/1* retourne un échec. Si toutes les alternatives de *p* échouent *not/1* retourne un succès par la deuxième clause. La coupure de la première clause est rouge parce que la lecture déclarative de la deuxième clause n'a pas de sens.

Des travaux, ayant comme but la recherche de la négation sure, ont été réalisés ([NAI 86], [BEK 90]). Une solution possible est de différer l'exécution du *not* jusqu'à ce que son argument soit clos.

2.3.4. Les Prédicats à Effet de Bord

La plupart des implantations de Prolog offrent divers prédicats prédéfinis qui provoquent des effets de bord. Parmi ceux-ci les principaux sont:

- entrée/sortie: par exemple, read/1, write/1;
- modifications dynamiques des programmes par inclusion/exclusion de clauses: par exemple, assert/1, retract/1;
- les "bases de données" ou table de termes à accès par clef.

En général, l'utilisation correcte de ces prédicats est dépendante des règles de sélection de Prolog. Si par exemple la position des appels de ces prédicats dans une clause est modifiée il est probable que le programme produira d'autres résultats. Le même phénomène peut se produire si l'ordre d'exécution des clauses est changé. Un exemple très simple est la lecture d'un terme, suivie de son écriture:

..., read(X), ..., write(X), ...

2.4. Les Machines Abstraites pour Prolog

Depuis le premier compilateur Prolog ([WAR 77]), les travaux sur l'amélioration de la compilation du langage ont conduit à la définition d'une machine abstraite de référence: la WAM, du nom de son créateur Warren ([WAR 83]). Bien que d'autres propositions de machines pour Prolog existent ([CLO 85], [CHA 86], [BEK 86]), la WAM est la machine la plus utilisée pour la compilation. Nous allons décrire les éléments et mécanismes principaux de cette machine séquentielle, qui sont nécessaires pour la compréhension des problèmes posés par la conception d'une machine abstraite parallèle.

Une machine abstraite Prolog (MAP) de type WAM est en général définie par un ensemble de types de données, un ensemble de registres, un ensemble d'instructions et une organisation de la mémoire. Dans la suite, l'organisation de la mémoire et la représentation des variables, ainsi que leurs relations avec le retour-arrière, seront examinées. Pour plus de détails sur la compilation ou sur les instructions, le lecteur peut consulter l'annexe B. Des références sur la WAM sont: [WAR 83], [GAB 85], [BOI 88], [AIT 90].

2.4.1. L'organisation de la Mémoire et le Retour-Arrière

La mémoire est organisée en 3 piles (conforme le schéma simplifié de la figure 2.4):

- la pile **locale**: utilisée pour le contrôle des appels de prédicats et du retour-arrière, et pour l'allocation des variables locales à une clause;
- la pile **globale**: utilisée pour la création des termes structurés;
- la pile **traînée**: utilisée pour la mise à jour des variables, lors du retour-arrière.

La pile locale contient deux types de structures de données:

- les **environnements:** un environnement contient les variables locales d'une clause en exécution;
- les **nœuds OU**: un nœud OU (ou point de choix) contient une sauvegarde de l'état de la machine, permettant le retour-arrière vers des clauses inexplorées.

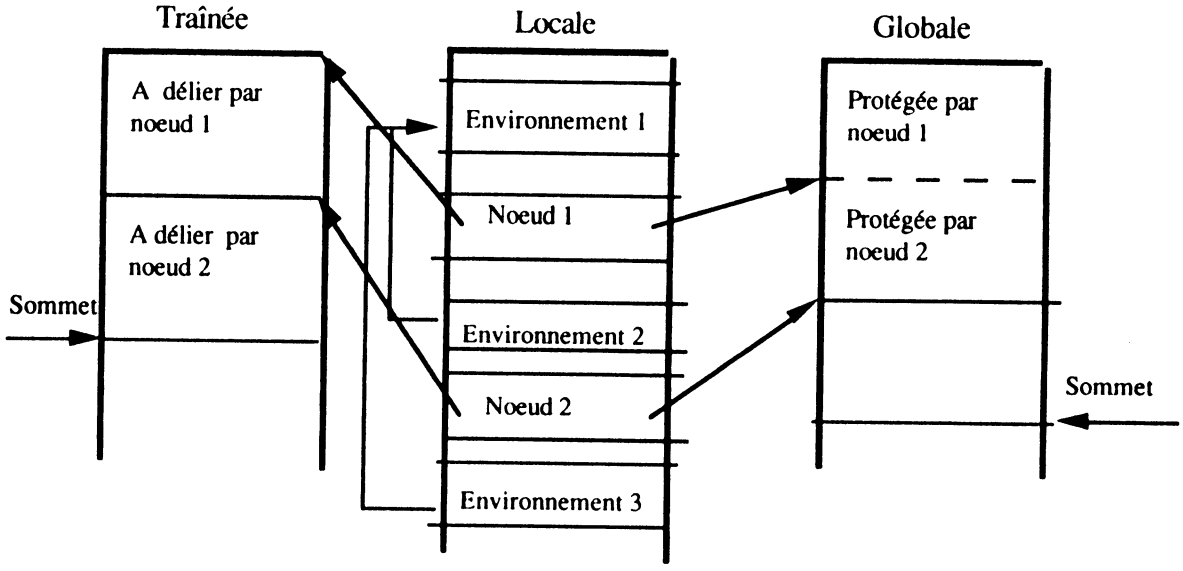

Fig. 2.4 - Mémoire simplifiée d'une MAP

L'environnement correspond à la notion de bloc d'activation de procédure dans la compilation des langages classiques. Il contient entre autres les informations nécessaires au retour à la clause appelante: prochain sous-but et pointeur vers son environnement (chaînage dynamique). Il est alloué au début de l'exécution de la clause, la taille étant déterminée par le nombre de variables de la clause. A la fin de l'exécution de la clause, il est libéré. L'environnement sert à renommer les variables d'une clause (ce qui est nécessaire dans la résolution SLD).

Un nœud OU est créé si le prédicat appelé par un sous-but est non-déterministe (plusieurs clauses). Les divers nœuds OU sont empilés. Le plus récent est utilisé en cas d'échec (retour-arrière), de façon à restaurer l'état de la machine pour l'exécution de la prochaine clause du prédicat. Un nœud OU est détruit (libéré), si cette prochaine clause est la dernière clause du prédicat. Un nœud OU protège automatiquement les environnements nécessaires au retour-arrière, du fait que les deux structures sont allouées dans la même pile physique. L'espace d'un environnement ne peut pas être réutilisé (après la fin de la clause), si un nœud OU a été créé pendant son exécution. En outre, un nouvel environnement est toujours alloué au sommet de la pile. L'environnement protégé par un nœud OU n'est récupéré que lors d'un retour-arrière à un nœud OU plus ancien. Ce mécanisme de protection et de récupération de mémoire, ainsi que les mécanismes équivalents des piles globale et traînée, exigent la sauvegarde des sommets des piles dans les nœuds OU (état de la machine).

Les termes structurés, nécessaires à l'exécution du programme, sont construits dans la pile globale. L'espace alloué n'est récupéré que lors d'un retour-arrière à un nœud OU, créé avant le terme, c.a.d. en cas d'échec ou de recherche d'une autre solution.

Le rôle de la pile traînée sera examinée après la présentation de l'implantation des variables.

2.4.2. Les Variables: Représentation et Etats

Une variable logique est représentée par une cellule de mémoire, dans la pile locale, si elle est locale à une clause, ou dans la pile globale, si elle est un sous-terme d'un terme structuré.

L'initialisation de la cellule (à la première référence) la met à **libre** ("valeur inconnue"). Après substitution, la variable reprend une valeur. Cette substitution est représentée par un pointeur vers le terme qui représente la valeur de la variable. Cette opération est nommée **affectation** ou **liaison**. Certaines variables peuvent être créées avant un nœud OU et affectées par une substitution après ce nœud. En cas de retour-arrière à ce nœud, ces affectations doivent être défaites sans détruire les variables. Cette opération est nommée **déliaison**.

Le graphe d'états d'une variable est montré dans la figure 2.5.

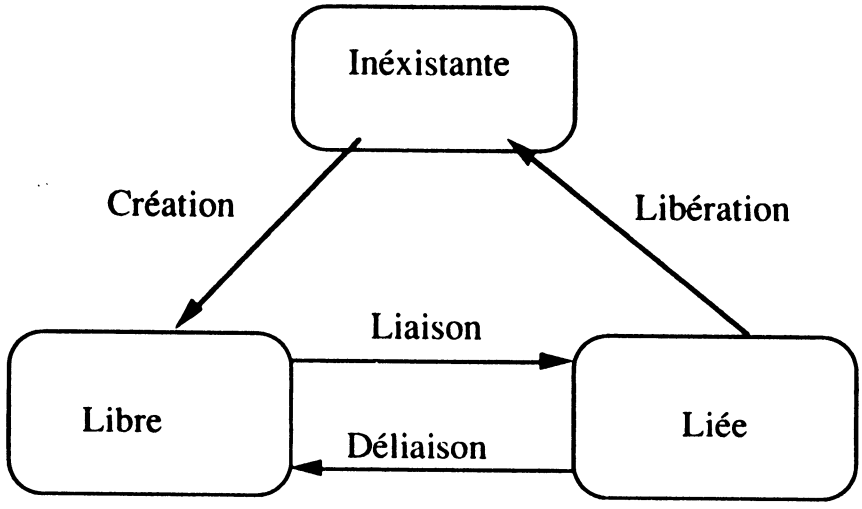

Fig. 2.5 - Graphe d'états d'une variable

Les liaisons **conditionnelles**, ou **inconditionnelles**, (fig. 2.6) sont celles qui sont, ou ne sont pas, séparées de la création de la variable par la création d'un nœud OU. Une liaison inconditionnelle n'est jamais défaite.

2.4.3. La Pile Traînée

Les déliaisons peuvent être effectuées à partir d'un enregistrement de chaque liaison conditionnelle. En général ces enregistrements sont maintenus dans une pile nommée pile traînée. Lors d'un retour-arrière vers un nœud OU, cette pile est parcourue entre le sommet actuel et le sommet au moment de la création du nœud OU. Toutes les variables enregistrées dans cette partie de la pile sont déliées.

2.5. Parallélisme en Programmation Logique

Dans cette section, nous présentons brièvement les différentes sources et types de parallélisme en programmation logique, les différentes approches pour la définition d'un langage de programmation logique parallèle et les problèmes posés par l'exploitation de ces sources de parallélisme, ainsi que des solutions simplificatrices.

2.5.1. Les Sources de Parallélisme

Les sources potentielles de parallélisme dans l'exécution d'un programme logique sont:

- le parallélisme OU: l'exécution en parallèle des clauses d'un prédicat;
- le parallélisme ET: l'exécution en parallèle des sous-buts d'une clause ou d'un but;
- le parallélisme dans l'unification: l'exécution en parallèle de l'unification d'un sous-but avec la tête d'une clause.

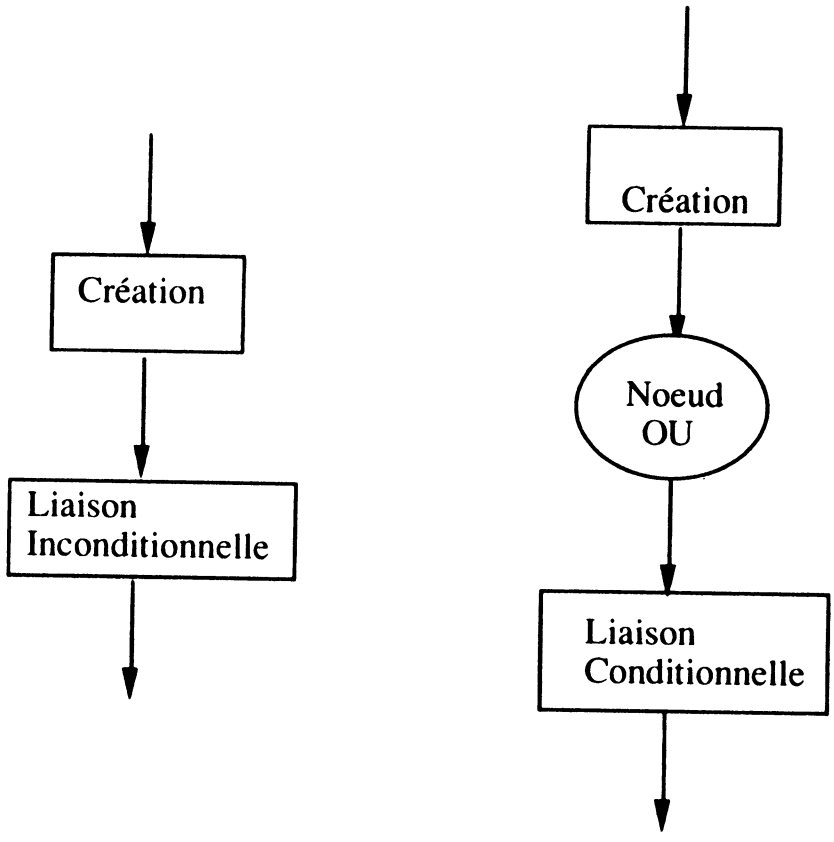

Fig. 2.6 - Liaisons inconditionnelle et conditionnelle

La troisième source est peu intéressante. Les travaux de Dwork et Yasuura ([DWO 84], [YAS 83]) ont conclu que l'unification est une opération plutôt séquentielle, sauf dans des cas spéciaux comme la vérification de concordance de modèles (*pattern matching*). La suite du texte ne traitera pas donc de ce type de parallélisme.

L'exploration de ces sources est à l'origine de plusieurs types de parallélisation d'un langage logique tel que Prolog:

- le parallélisme OU pur: exploration exclusive de la source OU;
- le parallélisme ET total: exécution en parallèle de tous les sous-buts d'une clause;
- le parallélisme ET indépendant: exécution en parallèle des sous-buts qui ne partagent pas de variables;
- le parallélisme ET de flot: exécution en parallèle des sous-buts qui partagent une variable dans une relation producteur/consommateur;
- le parallélisme global: exploration simultanée des deux sources, ET et OU, ce qui conduit à diverses variantes.

2.5.2. *Parallélisme Explicite et Implicite*

Une autre classification du parallélisme en programmation logique est relative à la manière dont le parallélisme apparaît au niveau du langage. Deux types se distinguent:

- le parallélisme implicite: le langage ne contient aucun mécanisme de contrôle du parallélisme. Dans le cas de Prolog la syntaxe n'est pas modifiée et seule la sémantique de prédicats prédéfinis à effet de bord doit être adaptée au parallélisme;
- le parallélisme explicite: le langage contient des opérateurs de contrôle du parallélisme. Il est nécessaire ici de différencier deux sous-types:

- Prolog "étendu": certains prédicats prédéfinis (d'utilisation non-obligatoire), sont ajoutés à Prolog. Le programmeur peut alors exprimer et contrôler l'exécution en parallèle. L'exemple le plus simple est une annotation qui permet au programmeur d'indiquer les prédicats qui peuvent être exécutés en parallèle (ou non).
- nouveau langage: l'introduction de certains mécanismes de contrôle provoque une modification substantielle de la sémantique originelle de Prolog, voire un écart important vis à vis de la logique. Les langages logiques gardés, décrits plus loin, en sont un exemple.

2.5.3. Modèles Théoriques et Multi-Séquentiels

Un modèle de parallélisation est en général caractérisé par la façon dont:

- le parallélisme est détecté;
- les processus sont créés et organisés;
- l'exécution en parallèle est contrôlée.

.

Les modèles de parallélisme en programmation logique peuvent être encore classés en modèles théoriques et en modèles multi-séquentiels ([CHA 89a], [CHA 89b], [ROB 88]).

La préoccupation majeure des modèles théoriques est la détection complète, et la représentation correcte, du parallélisme. Souvent, le point de départ est une analyse statique du programme, laquelle détecte le parallélisme potentiel et produit un graphe pour la représentation de celui-ci. Le contrôle de l'exécution est alors réalisé à partir de ce graphe: dynamiquement, de nouveaux processus sont créés à la rencontre d'une nouvelle source de parallélisme. D'un point de vue concret ces modèles présentent certains inconvénients (pas nécessairement chez tous):

- processus: la création de processus n'est pas contrôlée (parallélisme **vivace**), ce qui peut entraîner un nombre de processus bien supérieur à la limite déterminée par les ressources en calcul et/ou mémoire de la machine;
- communication: le contrôle de l'exécution détermine un échange important de messages entre les processus, soit pour synchroniser, soit pour passer des arguments ou pour retourner des solutions. Ceci, combiné au grand nombre de processus, peut saturer le réseau de communication (quel qu'il soit);
- granularité: le non-contrôle de la granularité du parallélisme peut conduire à des processus dont le coût de création excède la durée d'exécution.

L'exemple classique de modèle théorique est le modèle ET/OU de Conery ([CON 85]). Les caractéristiques principales en sont:

- il existe deux types de processus, les ET et les OU;
- un processus ET est alloué au but initial et à chaque clause non-unitaire d'un prédicat appelé: un processus ET crée un fils OU pour chaque sous-but;
- un processus OU est alloué à chaque sous-but du but initial ou d'une clause: un processus crée un fils ET pour chaque clause non-unitaire, dont la tête unifie avec les sous-but;
- les processus envoient des messages de succès ou d'échec à leurs parents, et des messages de réexécution et de révocation. Les messages de succès contiennent les solutions, c.a.d. les terme liés aux variables.

On constate un parallélisme important, puisqu'il consiste en un développement en parallèle de l'arbre ET/OU du programme (1 processus par nœud). Ce parallélisme est à grain fin.

Par contre, le premier but des modèles multi-séquentiels est l'augmentation de la vitesse d'exécution par rapport au modèle séquentiel. Il s'agit d'éviter la création anarchique de processus:

- processus: leur nombre est borné du fait du nombre de processeurs de la machine parallèle. La création de processus n'est réalisée que s'il existe des ressources pour l'exécuter (parallélisme **paresseux**);
- communication: elle est limitée à des moments particuliers: lorsque une nouvelle source productive de parallélisme est activée et, éventuellement, au moment de retourner les solutions;
- granularité: elle est contrôlée de façon à garantir un gain de performance. Ceci nécessite des mécanismes de régulation de charge. Une branche de l'arbre ET/OU n'est exécutée en parallèle que si elle contribue à l'accroissement d'efficacité (parallélisme paresseux).

Dans ce sens, l'exécution est initialement séquentielle. Elle ne devient parallèle que si ceci produit un gain, d'où le nom de modèle multi-séquentiel. Un intérêt supplémentaire est de profiter de la majorité des optimisations développées pour les compilateurs et les machines abstraites séquentielles.

2.5.4. Le Parallélisme OU

Le OU est une source de parallélisme: c'est la possibilité d'exécuter en parallèle les différentes clauses d'un prédicat. Ceci peut être étendu à l'exécution en parallèle des branches de l'arbre OU de la résolution SLD, c.a.d. la clause et le reste de la résolvante actuelle lors de la création d'un nœud OU. Ce modèle, nommé parallélisme OU multi-séquentiel, présente des propriétés intéressantes:

- simplicité: le modèle peut se réaliser par une simple extension des implantations séquentielles de Prolog;
- complétude: le modèle se rapproche de la stratégie en largeur pour la résolution SLD, en augmentant le nombre de solutions (succès), ou la probabilité d'une solution, dans le cas de programmes qui produisent des boucles infinies;
- gros grain: le grain inclut la clause plus le reste de la résolvante;
- peu de communication: les branches parallèles, étant indépendantes, n'ont besoin ni de communication ni de synchronisation, sauf au départ et dans des cas particuliers comme les prédicats à effet de bord.

Par contre, le parallélisme OU n'apparaît que dans les programmes non-déterministes, ou qui, au moins, possèdent quelques branches en échec de taille importante.

2.5.5. Le Parallélisme ET

Le parallélisme ET est l'exécution en parallèle des sous-buts d'une résolvante. Son implantation pose deux problèmes de fond:

- à cause du non-déterminisme: comment combiner les ensembles de solutions de chaque sous-but?
- à cause des variables partagées: comment concilier les liaisons différentes à la même variable, produites par l'exécution de sous-buts différents?

Evidemment, la combinaison des deux problèmes est la situation la plus complexe. L'interprétation classique de Prolog a résolu ces deux problèmes par l'ordre séquentiel d'exécution des sous-buts, combiné avec le retour-arrière.

Un autre problème aussi important, lié aux prédicats prédéfinis, provient de certains prédicats, nommés **consommateurs**, qui ne peuvent être appelés qu'avec des arguments non-variables. Cette restriction est souvent respectée en appelant un autre prédicat, nommé

producteur, qui lie les variables passées au consommateur. Ceci implique un ordre séquentiel d'exécution de ces prédicats, le producteur avant le consommateur. Cet ordre est garanti par l'interprétation classique de Prolog qui respecte l'ordre d'écriture des appels. Dans le cas du parallélisme ET, si cette restriction n'est pas contrôlée, l'exécution du consommateur va entraîner des échecs ou des boucles infinies.

Plusieurs modèles d'exécution proposent des restrictions du parallélisme ET. Une première approche, appelée parallélisme **ET indépendant**, ne propose l'exécution en parallèle que pour les sous-buts qui n'ont pas de variables libres partagées. Les sous-buts dépendants sont ceux qui partagent une ou plusieurs variables. L'un d'entre eux est dénommé **producteur**, les autres étant des **consommateurs**. Dans ce cas, le producteur est exécuté avant les consommateurs, de façon à lier les variables libres partagées. Les consommateurs peuvent être exécutés en parallèle, si les variables partagées ne sont plus libres. Les problèmes originels se transforment alors dans la détection de variables libres partagées. Trois façons de résoudre ce nouveau problème sont possibles. La première est l'analyse dynamique des arguments des sous-buts d'une clause ([CON 85]). Cette analyse permet de trouver tout le parallélisme indépendant, mais son coût de calcul, dû aux termes structurés (arbres), tend à être très élevé.

L'analyse statique, exécutée pendant la compilation et alliée à des annotations du programmeur, est la deuxième façon. Le coût d'exécution est nul, mais en général, le degré de parallélisme détecté sera moindre que celui du parallélisme indépendant potentiel ([CHA 85]).

La dernière possibilité, nommée parallélisme **ET restreint**, combine les deux autres de façon à réduire le coût de la première et à augmenter le degré de parallélisme détecté. Pendant l'exécution, de simples tests sur les variables sont nécessaires ([DEG 84], [HER 86a], [HER 86b]).

Une caractéristique positive du parallélisme ET est le fait qu'il apparaît dans tous les programmes, dès qu'il présente des clauses avec au moins 2 sous-buts. Néanmoins le grain des sous-buts est souvent très fin.

Une approche différente au parallélisme ET est proposée par les langages logiques gardés. Cette approche est examinée dans une section à part du fait de l'importance que ces langages ont acquis.

2.5.6. Le Parallélisme ET/OU

La combinaison des parallélismes OU et ET pose le problème du non-déterminisme dans le parallélisme ET. Une première solution est l'exécution parallèle **complète** (toutes les solutions) et indépendante de tous les sous-buts, exécution suivie d'une fonction de combinaison des ensembles de solutions en vérifiant l'intersection pour les variables communes (jointure relationnelle). Cette solution est pour l'instant ignorée, à cause de la complexité de la deuxième phase et de son coût très élevé en mémoire. Dans le cas de variables partagées, le coût en calcul peut être supérieur à celui de l'interprétation séquentielle. Celle-ci coupe d'un seul coup un sous-ensemble d'alternatives (un sous-arbre), si un sous-but, à gauche dans la résolvante, restreint le nombre de valeurs (solutions) d'une variable partagée par les sous-buts à droite. Par contre, en parallèle, chaque solution de chaque sous-but sera calculée une seule fois.

Quelques modèles proposent des limitations à l'exploitation de l'un ou de l'autre type de parallélisme, de façon à simplifier le contrôle de l'exécution. On peut citer:

- parallélisme ET déterministe: dans un modèle OU, les sous-buts sont exécutés en parallèle, s'ils sont déterministes ([CHA 89a]);
- limitation du parallélisme ET à deux sous-buts non-déterministes ([CAR 88]);
- jointure paresseuse, adoptée dans le projet PEPSys: la jointure des solutions produites par des sous-buts non-déterministes n'est réalisée qu'au moment où elle est nécessaire et s'il existe des processeurs inactifs ([CHA 89a]);

- la combinaison du parallélisme OU et du parallélisme ET de flot (voir la section sur les langages gardés), adoptée dans le modèle Andorra ([YAN 90]).

2.5.7. Les Langages Logiques Gardés

Les langages gardés présentent un parallélisme ET de flot explicite avec un non-déterminisme contrôlé. Ils ont pour objectif de limiter l'explosion du nombre de processus lié au parallélisme OU. Parlog, Concurrent Prolog (CP) et Guarded Horn Clauses (GHC) ([CHA 89b] en sont trois exemples. P-Prolog et Delta Prolog ([CHA 89b]) possèdent des similitudes avec les trois autres langages gardés, avec cependant des différences sensibles.

La caractéristique principale de ces langages est l'introduction du concept de gardes ([HOA 78]). En simplifiant, les trois langages définissent la clause gardée par:

$$A :- G_1, \ldots, G_m \mid B_1, \ldots, B_n. \quad m, n \geq 0$$

où $G_1, \ldots, G_m$ sont des gardes, c.a.d. des conditions qui doivent être vraies pour que, sous un appel donné, la clause fasse partie du sous-ensemble de clauses qui peuvent être exécutées.

Le contrôle du non-déterminisme est effectué sur ce sous-ensemble: une seule clause est aléatoirement choisie pour continuer l'exécution. Les autres clauses sont arrêtées et le retour-arrière n'est jamais utilisé. Le non-déterminisme de la programmation logique, ou **non-déterminisme libre** (*don't know non-determinism*), est ainsi remplacé par le **non-déterminisme contrôlé** (*don't care non-determinism*) des modèles de calcul de processus communicants (CSP [HOA 78]). En terme de Prolog, on a éliminé tout retour-arrière ou recherche exhaustive de solution.

Le parallélisme ET de flot est réalisé en utilisant des variables partagées par des sous-buts comme des canaux de communication. Une relation du type producteur/consommateurs est établie entre les sous-buts. Cette relation est déterminée soit explicitement par le programmeur soit par une analyse des termes de la tête des clauses. Un sous-but sera producteur si sa référence à la variable est annotée en sortie, et inversement consommateur si l'annotation est d'entrée. La détermination de l'annotation, entrée ou sortie, de la référence à la variable varie en fonction du langage:

- Parlog: par la déclaration de modes d'entrée/sortie sur les arguments du prédicat (appelé par le sous-but qui référence la variable);
- CP: par des annotations sur les références aux variables dans les sous-buts;
- GHC: par l'analyse (implicite) des termes de la tête de la clause.

L'avantage principal du parallélisme ET de flot est la possibilité pour les sous-buts de travailler en parallèle sur des éléments différents d'une solution partiellement construite. Par exemple étant donnée le but *p(X), q(X)* et si *X* est une liste construite par le prédicat *p/1*, celui-ci peut travailler sur l'élément d'ordre *i+1* de la liste pendant que le prédicat *q/1* travaille sur l'élément *i*.

Suivant les auteurs de ces langages ([SHA 89]) leur application principale est la programmation du logiciel de base des machines futures, par exemple le développement des systèmes d'exploitation parallèles et distribués.

2.6. Conclusions

Les principaux points de ce chapitre sont:

- Prolog est un langage qui permet de résoudre certains problèmes d'une façon élégante et concise;
- Prolog présente des sources implicites de parallélisme;

- le parallélisme ET existe dans toutes les applications. Cependant son grain est plus fin que celui du parallélisme OU et il exige une synchronisation entre les sous-buts parallèles, au cours de l'exécution;
- le parallélisme global est nécessaire à un système qui se veut efficace pour tous les programmes. Néanmoins, il est encore très difficile de combiner, sans trop de restrictions, les parallélismes ET et OU;
- les implantations concevables du parallélisme en programmation logique se réduisent à:

    - l'approche multi-séquentielle OU: elle est la plus simple et elle permet des implantations efficaces;
    - l'approche parallélisme ET restreint (multi-séquentielle);
    - l'approche des langages gardés: le non-déterminisme est écarté;
    - la combinaison du modèle multi-séquentiel OU avec le parallélisme ET restreint (par exemple, le projet PEPSys [CHA 89a]).

3. Chapitre 3
Le Parallélisme OU Multi-séquentiel

Ce chapitre est une présentation des modèles multi-séquentiels OU parallèles. Cet état de l'art est justifié par le fait que Opera, sujet de cette thèse, est un tel système.

Le chapitre est organisé comme suit. Une brève introduction rappelle les objectifs et les propriétés du modèle multi-séquentiel OU parallèle, ses principes de fonctionnement et les conditions nécessaires à l'existence dans un programme d'un tel parallélisme.

Nous décrivons les principaux problèmes de conception d'un système multi-séquentiel OU parallèle et les solutions possibles les plus remarquables.

Le problème crucial de la régulation de charge, comme garantie d'un accroissement de performance par le parallélisme, est posé, et des éléments de réponse sont apportés.

Un système Prolog OU parallèle ne se réduit pas à la résolution parallèle. Des opérateurs spécifiques du langage Prolog doivent voir préserver leur sémantique. C'est le cas de la coupure, comme des opérateurs à effet de bord (entrée/sortie, gestion de base de données, ...). Des solutions possibles sont rapidement décrites.

Enfin, ce chapitre termine par un bilan sur les systèmes actuellement implantés.

3.1. Les Principes

L'intérêt du parallélisme OU multi-séquentiel provient des raisons suivantes:

- le parallélisme OU est plus simple à implanter que le ET;
- les modèles multi-séquentiels, dans la perspective d'un accroissement de la performance, sont plus réalistes que les modèles théoriques, principalement si on considère les caractéristiques des machines parallèles actuelles.

Le parallélisme OU, dans un système multi-séquentiel, est l'exécution en parallèle des diverses résolvantes, dérivées à partir des nœuds du programme. Dans un système multi-séquentiel ces résolvantes sont exécutées en parallèle par plusieurs MAP (WAM), suivant une mise en œuvre classique de Prolog.

En général ces modèles partagent les caractéristiques suivantes:

- la machine parallèle comprend un nombre fixe de MAP, nombre qui est fonction des ressources en mémoire et processeur;
- le but initial est soumis à une MAP qui commence une résolution SLD avec les règles de sélection de Prolog;
- si un nombre intéressant d'alternatives, à définir en fonction de la granularité, existe dans les nœuds OU d'une MAP active (une seule à l'état initial), et si une autre MAP est inactive, une partie de ces alternatives est transférée de la MAP active à l'inactive; la première MAP continue son exécution avec la partie restante des alternatives (au moins la branche sur laquelle elle travaillait); tout le sous-arbre (lié à une alternative) de l'arbre OU est transférée à la MAP inactive, et pas seulement la clause qui est à l'origine de l'alternative;
- toutes les MAP suivent la résolution SLD de Prolog et épuisent les alternatives présentes par retour-arrière quand la branche courante arrive à une feuille (succès ou échec);

- chacune des MAP peut produire ses solutions de façon indépendante, si l'ordre des solutions n'est pas important. Dans le cas contraire, celles-ci doivent être collectées afin de permettre la présentation finale correcte. Il est important de remarquer que les solutions partielles (à un sous-but quelconque) ne sont pas nécessaires à une MAP différente de celle qui les a produites.

Un avantage important des modèles multi-séquentiels est de profiter des techniques d'optimisation des implantations séquentielles les plus efficaces.

Une seule MAP, et donc d'un seul processus, est nécessaire pour exécuter une branche de chaque sous-arbre de l'arbre OU. En outre une MAP n'abandonne jamais une branche après l'avoir commencée. Chaque MAP peut parcourir successivement plusieurs branches. Les parties communes d'un sous-ensemble de branches sont parcourues par la même MAP, c.a.d. chaque arc est parcouru une seule fois. Le temps d'exécution est le temps d'exécution en séquentiel de la branche la plus grande, plus les sur-coûts dus aux installations d'alternatives pendant le parcours de cette branche.

Les systèmes multi-séquentiels OU varient en fonction de la règle de choix des branches à transférer de la MAP active à l'inactive. On rappelle que la résolution SLD permet tous les choix de clauses possibles. Cette règle détermine donc une autre stratégie de parcours de l'arbre OU, complémentaire à celle définie par le retour-arrière de Prolog. Cette stratégie peut approcher le parcours en largeur d'abord, en augmentant le nombre de succès produits dans les cas d'arbres OU avec des branches infinies.

Une variante remarquable des modèles multi-séquentiels a été proposée dans [CLO 88]. L'objectif est la suppression de toute communication entre deux MAP. Dès le début, toutes les MAP sont actives. Chacune reçoit une branche à parcourir, laquelle est définie par une séquence de bits (l'arbre OU doit être préalablement mis sous forme d'arbre binaire: deux clauses par prédicat). Une MAP prend l'alternative de gauche, c.a.d. la première, si le bit est 0, et prend celle de droite si le bit est 1. Une fonction de contrôle est nécessaire pour garantir l'obtention de toutes les solutions. En effet, les séquences de bits, nommées **oracles**, sont, en principe, incomplètes, ou encore, le nombre d'oracles est plus grand que le nombre de MAP. Cette incomplétude est due à l'impossibilité de déterminer, par avance, la longueur de chaque branche. Plusieurs fonctions de contrôle sont proposées et les plus simples n'ont pas besoin du retour-arrière, ce qui permet la suppression de la pile traînée. Dans ce modèle le coût de communication entre MAP est remplacé par:

- le coût de duplication du calcul des arcs communs à plusieurs branches;
- le coût de l'obtention de la continuation d'un oracle incomplète.

Cependant les premiers sur-coûts sont nuls si le nombre de MAP est plus grand que le nombre de branches, ceci déterminant un temps d'exécution égal au temps d'exécution en séquentiel de la branche la plus grande.

3.2. La Condition d'Existence du Parallélisme OU

La condition nécessaire du parallélisme OU est l'occurrence de prédicats avec au moins deux clauses. Cependant l'indexation des clauses par le type, ou par le contenu, des arguments du sous-but peut restreindre le nombre de clauses à une seule exécutable.

Un prédicat avec plusieurs clauses peut être déterministe pour tous les appels: une seule clause retourne une solution, toutes les autres échouent soit par l'échec de l'unification du sous-but avec leur tête, soit par l'échec d'un sous-but de leur corps. Dans le premier cas d'échec, le grain du parallélisme OU se réduit à l'unification parallèle des têtes des clauses avec le sous-but.

Du point de vue strict de l'efficacité du parallélisme, en considérant les caractéristiques de communication des machines actuelles, le parallélisme OU devient intéressant si l'arbre OU présente des branches suffisamment longues pour permettre des gains de performance. Ceci équivaut à la condition "le grain des exécutions parallèles (durée d'exécution) est supérieur au coût de communication nécessaire pour le transfert des alternatives entre deux MAP".

Les divers systèmes multi-séquentiels diffèrent par la façon de réduire ces coûts. Cependant des programmes réels ont des caractéristiques favorables au parallélisme OU; on en trouve des exemples dans les domaines de la langue naturelle, de la preuve de théorèmes, de l'interprétation et de la compilation des langages, des applications de la théorie de graphes, de la recherche d'une séquence de caractères dans un document et encore d'autres ([ALI 90], [CHA 89b], [WAR 87a]).

3.3. La Gestion des Contextes Multiples

Dans les modèles multi-séquentiels, une MAP active fournit une ou plusieurs branches suspendues, décrites dans un ou plusieurs nœuds OU, à une MAP inactive. Chaque branche se transforme à son tour en un arbre si elle crée des nœuds OU dans son exécution. Ce sous-ensemble de branches sera par la suite appelé une **tâche**, la MAP active sera appelée **exportatrice** et l'inactive **importatrice**. Toute l'opération sera une **installation de tâche**. Dans la suite, on considère une organisation de la mémoire en piles.

La MAP importatrice, pour pouvoir exécuter sa tâche, a besoin d'accéder à une partie de la mémoire de la MAP exportatrice. Cette partie contient toutes les données du programme et du contrôle de l'exécution, créées avant le nœud OU le plus récent (nommé **PPR** par la suite) de ceux qui exportent des branches. Le contrôle, du point de vue de l'interprétation procédurale, correspond aux points de retour aux procédures (clauses) encore actives, et, du point de vue de la programmation logique, à la suite de la résolvante actuelle. Les données sont soit des variables, soit des termes structurés, ceux-ci pouvant, à leur tour, contenir des variables. Cette partie de la mémoire sera dans la suite appelée **section commune** entre les MAP exportatrice et importatrice. Du fait de la notion même d'arbre OU de recherche, 2 MAP ont toujours une section commune.

L'implantation d'un système multi-séquentiel nécessite de définir une représentation des sections communes permettant:

- l'accès parallèle par les MAP à leurs sections communes;
- le maintien de la cohérence de ces sections communes.

Ce problème de maintien de cohérence apparaît puisque des MAP différentes vont tenter de substituer des valeurs particulières aux variables libres de leur section commune.

Cette représentation doit en outre correspondre à des compromis d'efficacité, prenant en compte les différents types d'accès et leur coût, selon les architectures de machines parallèles.

3.3.1. La Gestion de la Mémoire

Par hypothèse de base des systèmes multi-séquentiels, tout processeur de la machine parallèle exécute une ou plusieurs MAP classiques, de façon à préserver l'efficacité de l'exécution séquentielle. Chaque MAP possède donc sa propre mémoire, organisée en piles. Il existe deux façons simples de permettre à des MAP d'accéder concurremment à une section commune:

- le partage d'une mémoire physique contenant cette section commune;
- la duplication de la section commune entre les MAP utilisatrices.

Duplication:

L'accès concurrent aux zones de mémoire de la MAP exportatrice (constituant la section commune) est limité à la période de copie de ces zones, vers la mémoire de la MAP importatrice. Cependant cette méthode augmente la consommation globale de mémoire, au sens de la machine parallèle. Elle peut être implantée sans aucune modification de la MAP, à l'exception évidemment de précautions nécessaires au maintien de la cohérence des données pendant les accès concurrents de la copie.

Partage:

Si tout processeur peut accéder à la mémoire d'un autre processeur, une section commune peut résider dans la mémoire de la MAP qui l'a initialement créée. Toutefois, ce partage influe sur la liberté de gestion de mémoire de cette MAP dans les deux situations suivantes:

- S1: optimisation d'appel terminal: une MAP termine une clause précédant le nœud PPR et essaie de récupérer l'espace alloué à son environnement, car n'ayant plus d'alternatives suspendues postérieures à cette clause. Si l'autre MAP a encore besoin de l'environnement de cette clause (fig. 3.1), cet environnement ne peut pas être dépilé;
- S2: épuisement des alternatives: une MAP, après avoir parcouru toutes les branches inférieures au nœud PPR, doit faire un retour-arrière à un nœud OU précédent, ou, si celui-ci n'existe pas, demander une tâche à une troisième MAP: dans les deux cas elle risque de récupérer de la mémoire encore nécessaire à l'autre MAP (fig. 3.2) car étant incluse dans leur section commune.

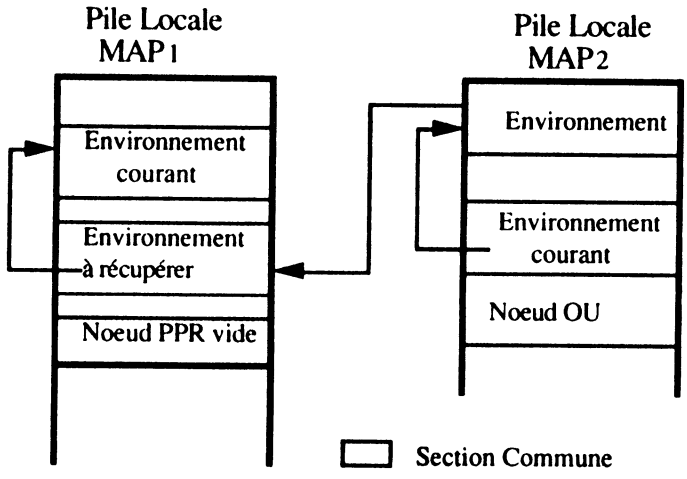

Fig. 3.1 - Récupération d'environnement commun

Ces deux situations d'accès concurrent (S1 et S2) correspondent à une récupération de mémoire qui doit être retardée. En effet, il est suffisant d'étendre l'action protectrice du nœud PPR jusqu'à la fin de toutes (ou toutes moins une) ses branches, cette action étant déjà existante dans l'exécution séquentielle, à l'exception de la dernière branche. Néanmoins la récupération des éléments de la section commune peut être bloquée par d'autres allocations, réalisées postérieurement à la section commune, pendant la protection du nœud PPR. Des variations de ce problème, dit **du trou noir,** sont décrites dans [HER 86c] et [WAR 87b] (*garbage slot problem, trapped goal problem* et *hole*).

Dans un système où la mémoire est divisée de façon statique entre les MAP (segmentation [WAR 87b]), la seule MAP qui peut récupérer la section commune est celle qui l'a créée. La section commune peut rester bloquée longtemps, si cette MAP a importé une branche non-descendante du nœud PPR, pendant la période de protection.

Une façon d'éviter les trous est l'introduction de restrictions dans l'importation de tâches (ordonnancement). Par exemple, une MAP ne peut importer que des tâches descendantes de sa section commune.

La récupération d'un environnement de la section commune pourrait se réaliser au moment où toutes les branches ont "exécuté" la clause correspondante. Cependant un tel contrôle n'est pas simple, son coût n'étant probablement pas compensé par les gains. En plus, cette récupération est limitée par les trous noirs.

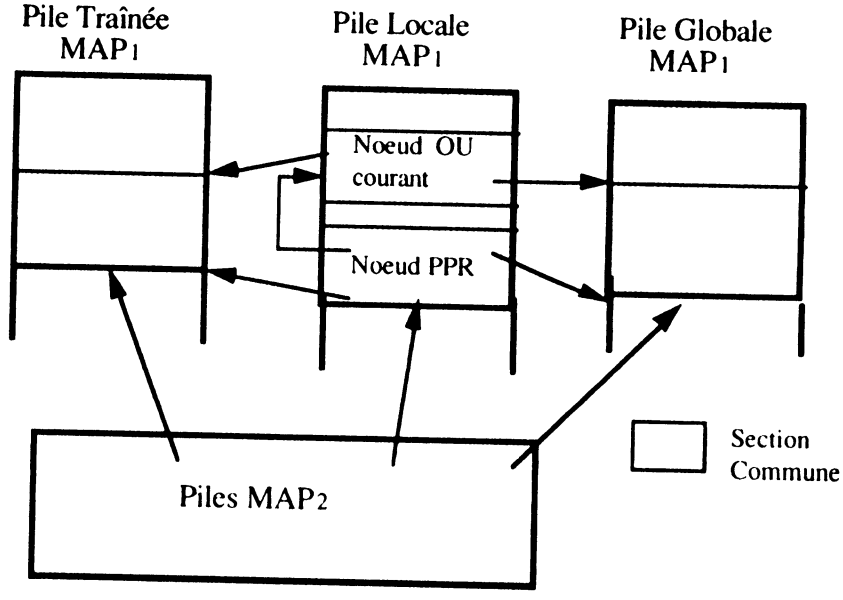

Fig. 3.2 - Récupération totale de la section commune

Bilan:

Ces deux façons de partager la section commune correspondent parfaitement à une caractéristique importante des machines parallèles actuelles: la possibilité, ou l'impossibilité, pour un processeur d'accéder à la mémoire des autres processeurs. Du fait de la grande différence entre les temps d'accès à une mémoire locale et à une mémoire non-locale dans les machines sans mémoire commune (au moins pour les machines actuelles), et du grand nombre d'accès effectués par la MAP importatrice aux données de la section commune, la solution partage de piles n'est pas convenable pour cette classe de machines. Par contre, la copie des piles peut être utilisée dans les deux classes de machines, évidemment avec une consommation de mémoire plus élevée. La plupart des modèles actuels utilisent le partage de piles puisque ils ont été implantés, au moins initialement, sur des machines avec mémoire commune, en raison des facilités de programmation et de la meilleure disponibilité de cette classe de machines.

3.3.2. Le Maintien de la Cohérence des Sections Communes

Une section commune apparaît lorsque une MAP importe une partie des alternatives d'une MAP exportatrice. Cette section commune correspond à l'ensemble des données créées avant le nœud OU le plus récent (noté PPR) des alternatives exportées.

Il s'ensuit que toutes les liaisons de variables effectuées par la MAP exportatrice, dans cette section commune et postérieurement à la création du nœud PPR, sont invalides pour la MAP importatrice. Pour la deuxième MAP, ces variables sont encore libres. De plus, les deux MAP seront en conflit pour les liaisons des variables libres de cette section commune. Ce problème des variables libres se décrit par un exemple simple.

La variable X, d'un fragment d'un programme Prolog quelconque (fig. 3.3), est créée par la MAP_i pendant l'exécution du corps de la clause C, avant le sous-but p.

```
        ...p(X, Y), q(X, Y), ...        corps de la clause C.
p(a, Y) :- r(Y).
p(b, Y) :- s(Y).
r(c).
s(d).
...
```

Fig. 3.3 - Liaisons multiples (a)

Au début de l'exécution du prédicat *p*, un nœud OU est créé pour contrôler les diverses clauses de *p*. La MAP_i prend la première alternative, lie *X* à *a* (et *Y* à *c*) et continue l'exécution par le sous-but *q(a, c)* (fig. 3.4).

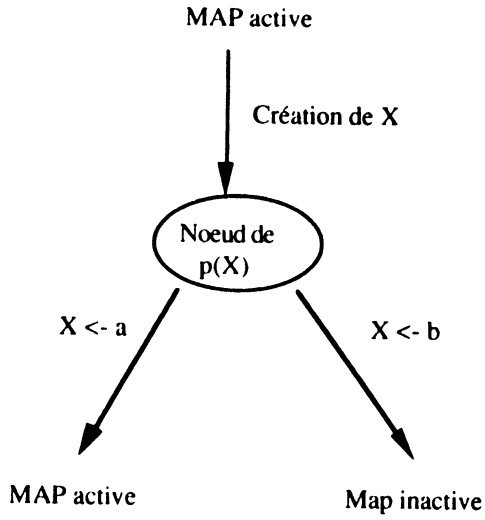

Fig. 3.4 - Liaisons multiples (b)

Une MAP_j inactive prend la deuxième alternative. X appartient à la partie commune aux deux MAP. La MAP_j lie *X* à *b* (et *Y* à *d*) et continue l'exécution par le sous-but *q(b, d)*, de façon indépendante. De même, d'autres variables, *Y* par exemple, créées avant le nœud OU de *p(X)* et encore libres comme *X*, peuvent être liées par les sous-buts suivants de la résolvante actuelle avec des valeurs différentes. Les liaisons conditionnelles déjà effectuées par la MAP exportatrice (sur *Y* par exemple: $Y = c$) ne doivent pas être considérées par la MAP importatrice.

La solution à ce problème se trouve soit dans une installation particulière des variables logiques, soit dans la liaison et le déréférencement de variables, ou encore par une combinaison des deux. Une des différences entre les modèles multi-séquentiels provient de la solution à ce problème. En effet, ces solutions impliquent des coûts différents d'installation et d'accès.

La liaison et le déréférencement de variables sont les opérations les plus utilisées par une MAP. Certaines solutions au problème des liaisons conditionnelles induisent des sur-coûts indépendants du degré du parallélisme. D'autres solutions ont des sur-coûts proportionnels au nombre de processeurs. Par contre, l'installation ne survient que lors des transferts de tâches, donc son coût total est en général dépendant du degré du parallélisme, et il affecte les gains de performance par rapport au nombre de processeurs.

Le mécanisme classique de liaison (conditionnelle ou non) dans une implantation séquentielle est appelé **liaison superficielle** ([CHA 89b]). La liaison superficielle de Myazaki ([MYA 85]), proposée pour l'implantation du langage Concurrent Prolog sur une machine séquentielle, est basée sur le mécanisme analogue de Lisp ([BAK 78]). Elle implique seulement la sauvegarde de la valeur courante (précédente) dans la pile traînée. Dans une implantation parallèle, ce mécanisme n'est pas utilisable pour les liaisons conditionnelles, si la

section commune est physiquement partagée. Différents mécanismes ont alors été proposés, sous le nom général de **liaison profonde** ([CHA 89b]). La liaison profonde implique un sur-coût dans la liaison et dans l'accès aux valeurs des variables.

L'opposition liaison superficielle/liaison profonde est plutôt physique et liée à l'implantation; l'opposition liaison conditionnelle/liaison inconditionnelle dépend des situations respectives de l'exemplarisation et de la liaison dans l'arbre OU. Il est nécessaire de maintenir ces deux concepts parce que plusieurs méthodes utilisent la liaison superficielle pour certaines liaisons conditionnelles.

Les sections suivantes décrivent rapidement les solutions principales dans le cadre de la copie ou du partage, comme moyen d'implanter les sections communes.

3.4. La Copie des Sections Communes

Dans un système multi-séquentiel basé sur la copie, le partage des sections communes est résolu trivialement, et le maintien de la cohérence de ces copies se réduit à trouver, lors de la copie vers une MAP importatrice, les liaisons de variables à défaire.

3.4.1. La Copie Simple de Piles

La solution la plus simple est une copie complète de toutes les piles, sans modification de la MAP ([YAS 84], [ALI 90]). La pile traînée doit être intégralement transférée afin de permettre la réalisation des déliaisons. Ce transfert de la pile traînée présente une source d'inefficacité spécifique: une partie des liaisons, enregistrées dans cette pile, n'est pas nécessaire, puisque leurs variables n'appartiennent pas à la section commune: ces variables ont été créées après le PPR.

L'avantage principal de cette solution de copie intégrale est le sur-coût nul en exécution séquentielle.

3.4.2. La Copie avec Datation

Cette variante de la copie évite le transfert de la pile traînée ([SOH 85]). Les variables portent un champ additionnel contenant la date de liaison. Celle-ci correspond à la profondeur de la branche de l'arbre au moment de la liaison. La déliaison est réalisée en comparant la date de liaison avec la profondeur du nœud exporté PPR. La datation introduit un sur-coût dans la liaison des variables et dans le traitement des nœuds OU (gestion de la date).

La figure 3.5 illustre cette technique. La variable X doit être déliée si le nœud 4 est transféré. Par contre, la liaison $X = a$ est valide pour la MAP importatrice si le nœud 5 est transféré.

3.5. Le Partage des Sections Communes

Il s'agit essentiellement de trouver un moyen d'éviter de confondre les différentes liaisons des variables de la section commune entre les différentes MAP. Ceci se réalise en jouant sur l'installation des variables ou sur le déréférencement de celles-ci.

3.5.1. Le Tableau de Liaisons

Dans [WAR 87a], D.H.D. Warren résume les caractéristiques de diverses méthodes qui utilisent le partage de piles, et discute leurs avantages et désavantages. Dans [WAR 87b], il propose la méthode simple et efficace du tableau de liaisons (SRI model). Cette méthode a aussi été proposée par D.S. Warren ([WAR 84]), dans un autre contexte.

En plus des piles traditionnelles, chaque MAP possède un tableau de liaisons. La création d'une variable dans une pile s'accompagne de l'allocation de la prochaine position libre dans le

tableau. Cette position est sauvegardée dans la cellule de la variable avec l'indication libre. Au moment de la liaison, si celle-ci est inconditionnelle, la valeur est sauvegardée dans la cellule, en détruisant la position dans le tableau. Dans le cas d'une liaison conditionnelle, la valeur est sauvegardée dans le tableau (fig. 3.6).

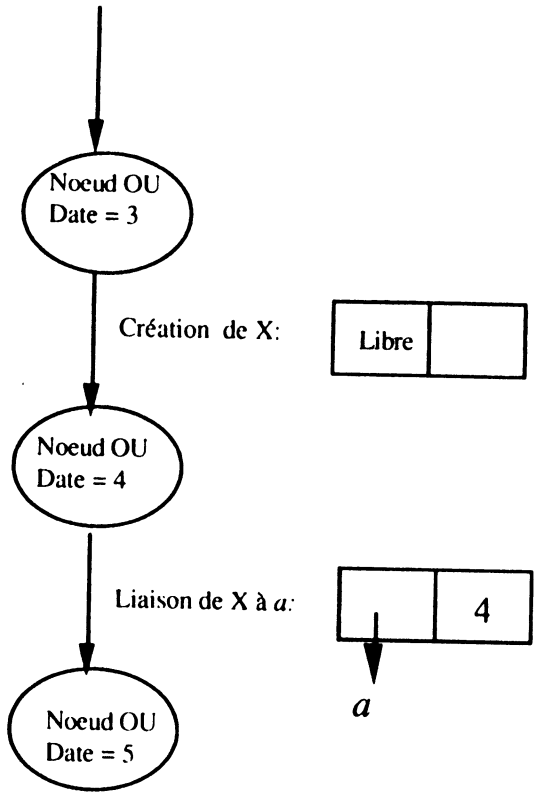

Fig. 3.5 - Datation

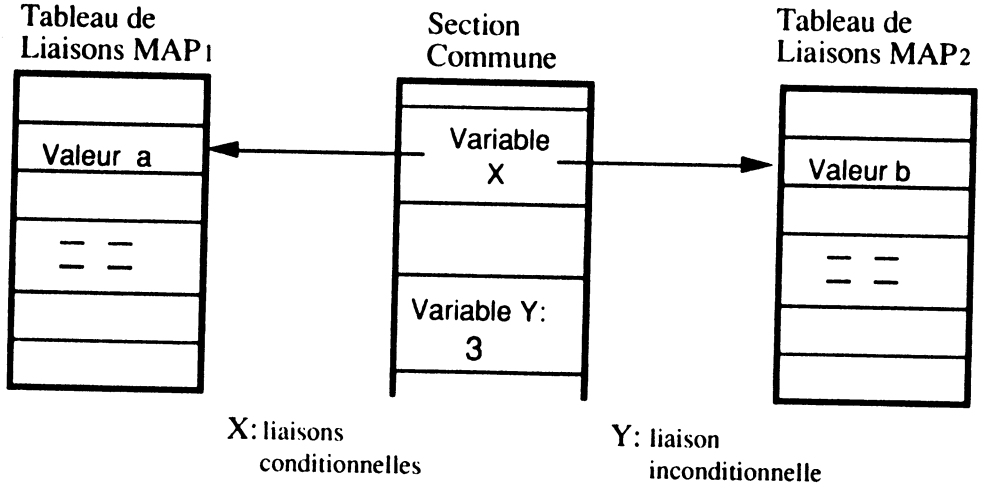

Fig. 3.6 - Tableau de liaisons

Le déréférencement d'une liaison inconditionnelle n'a aucun sur-coût. Par contre, dans les cas de variables libres et de liaisons conditionnelles, il faut y accéder indirectement par le tableau.

Dans le cas où les MAP possèdent déjà une section commune, l'opération d'installation s'exécute en trois phases:

- la déliaison des liaisons conditionnelles produites par la MAP importatrice sur les variables communes: les liaisons à délier sont celles effectuées après le précédent nœud OU commun (PPR);
- la section non-commune du tableau est initialisée à libre: cette phase peut être supprimée si le tableau est initialisé de façon statique;
- l'importation des liaisons conditionnelles produites par la MAP exportatrice entre le précédent et le nouveau nœud PPR (fig. 3.7).

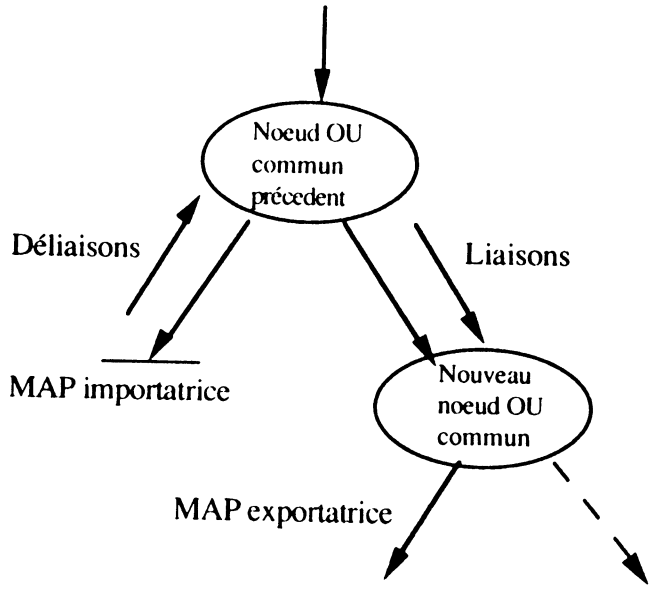

Fig. 3.7 - Installation avec tableau de liaisons

L'avantage de cette méthode est le sur-coût faible et constant dans la liaison et dans le déréférencement. Par contre, le coût de l'installation est proportionnel au nombre de liaisons conditionnelles, qui existent dans le chemin qui va de la position actuelle de la MAP importatrice jusqu'au nouveau nœud PPR, en passant par le précédent nœud PPR.

3.5.2. Les Liaisons Privilégiées

Une liaison privilégiée est une liaison conditionnelle effectuée par la MAP qui a créé la variable. Elle a été utilisée dans la machine abstraite ANLWAM ([BUT 86]), une extension parallèle de la WAM. Les liaisons conditionnelles effectuées par les autres MAP sont nommées liaisons non-privilégiées. La MAP doit pouvoir vérifier si elle a créé une variable ou si celle-ci a été héritée d'une MAP exportatrice. En général, une comparaison des adresses de la variable et des sommets des piles relatifs au nœud PPR est suffisante.

Dans la ANLWAM les liaisons privilégiées sont effectuées de façon superficielle. Un drapeau particulier différencie une liaison privilégiée d'une liaison inconditionnelle. Les autres MAP, qui héritent une telle variable, doivent la lier de façon profonde.

L'avantage des liaisons privilégiées est le coût faible du déréférencement par la MAP mère de la variable. Ceci peut être important dans les parties séquentielles de l'exécution d'un programme.

3.5.3. Les Tables d'Adressage Associatif

Les liaisons conditionnelles peuvent être maintenues dans des tables d'adressage associatif. Dans la ANLWAM ([BUT 86]), une table d'adressage associatif contient les liaisons d'un arc de l'arbre. Elle peut être accédée par les MAP qui parcourent des branches dérivées de cet arc. Les tables des arcs qui constituent une branche sont chaînées. L'installation

d'une tâche se résume à une mise-à-jour des pointeurs nécessaires à l'accès à la liste de tables de la branche, entre la racine et le nœud PPR. Le déréférencement exige une recherche dans toutes les tables créées après la création de la variable. La comparaison de dates de création se ramène à une comparaison d'adresses. Dans la ANLWAM il n'est pas possible de vérifier si une liaison privilégiée est valide, ou non, pour une MAP qui a hérité la variable (voir les deux situations dans la fig. 3.8), donc les liaisons privilégiées sont dupliquées dans les tables.

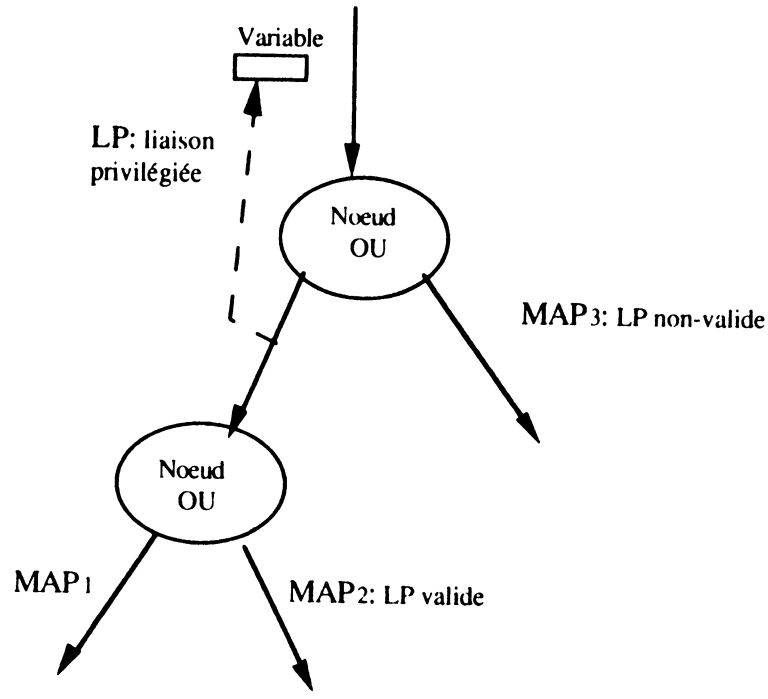

Fig. 3.8 - Liaison privilégiée

Le projet PEPSys utilise une table par MAP, ce qui implique une table contenant les liaisons conditionnelles de toute une branche. La MAP importatrice accède à la table de la MAP exportatrice et aux tables héritées par celle-ci. Les liaisons privilégiées, nommées locales, sont maintenues en mode superficiel dans les MAP qui héritent ces liaisons. Ceci est un avantage par rapport à la ANLWAM, dont les liaisons privilégiées héritées sont dupliquées dans les tables. La datation est utilisée pour vérifier la validité des liaisons des MAP mères.

3.5.4. La Liste de Liaisons et l'Historique

Dans BOPLOG ([TIN 87]) les diverses liaisons d'une variable sont maintenues dans une liste de liaisons, chaque variable ayant une liste exclusive. La vérification de la validité s'effectue par datation et par un historique de l'exécution parallèle. La date est un compteur, incrémenté à chaque alternative exécutée soit en séquentiel, soit en parallèle.

L'historique de chaque MAP est une liste des MAP mères, dont chaque élément contient la période de validité (pour la MAP associée à l'historique) des liaisons effectuées par la MAP de l'élément. Evidemment, la liaison et le déréférencement sont des opérations complexes, parce qu'il faut parcourir l'historique. Néanmoins, les auteurs considèrent que leur coût n'est pas excessif à cause du petit nombre de liaisons par variable et de la petite longueur de chaque historique.

Etant donné que la déliaison n'est pas nécessaire, la pile traînée peut être supprimée. L'avantage de la méthode est une installation rapide puisqu'elle n'exige que la copie de l'historique de la MAP exportatrice. Par contre, il est difficile de récupérer la mémoire occupée par les liaisons qui ne sont plus nécessaires.

3.5.5. La Fermeture d'Environnements

Une fermeture d'environnements *E* est un ensemble d'environnements dont la totalité des pointeurs ne pointent pas vers des variables situées hors des environnements de *E*. Cette méthode, proposée par Conery ([CON 87]), exige l'exécution d'un algorithme de fermeture d'environnement après chaque unification et à la fin d'une clause non-unitaire. L'algorithme opère sur l'environnement courant et celui de la clause mère, en fermant l'un des deux. L'un des résultats importants de l'algorithme est le maintien de toutes les variables dans les environnements, autrement dit les variables des termes structurés sont représentées par une référence vers une variable dans *E*.

L'installation d'une tâche correspond à la copie de l'environnement relatif au sous-but qui est à l'origine du nœud PPR. Cet environnement, et, celui qui sera créé pour une clause du nœud OU, contiennent toutes les variables nécessaires à l'unification du sous-but avec la tête de la clause. Evidemment l'exécution de la tâche exige un accès aux termes créés avant le nœud OU. De même, au moins dans le cas du modèle ET/OU de Conery, les résultats doivent être retournés à la MAP exportatrice à la fin de la tâche, puisque la continuation de la résolvante n'est pas transférée à la MAP importatrice.

L'avantage de cette méthode est l'inexistence de pointeurs vers des variables non-locales. Dans une machine où les temps d'accès d'une mémoire globale et d'une mémoire locale sont différents, la méthode permet un temps d'accès constant. Par contre, le coût de l'algorithme est proportionnel au nombre total d'éléments des termes structurés pointés par les éléments de l'environnement à fermer.

3.6. L'Ordonnancement

Cette section traite du problème de l'ordonnancement dans les modèles multi-séquentiels. Le besoin du contrôle de la charge y est justifié, les problèmes posés par la réalisation d'un contrôle optimal sont présentés et diverses méthodes proposées pour ce faire sont décrites.

L'ordonnancement dans les modèles multi-séquentiels a deux objectifs:

- un accroissement de parallélisme ne doit pas conduire à une baisse d'efficacité;
- pour un nombre donné de processeurs, obtenir la meilleure performance possible.

Le premier point provient de ce qu'une augmentation du parallélisme peut dégrader les performances, si les temps d'installation de tâches sont supérieurs à leurs temps d'exécution. Ceci équivaut à un problème de contrôle du grain de parallélisme exploitable. Le second point consiste à choisir parmi toutes les tâches exportables, le sous-ensemble produisant le meilleur gain de performance.

3.6.1. Les Conditions Minimales du Parallélisme Efficace

Cette section analyse le problème du contrôle du grain de parallélisme exploitable. Pour illustrer le problème, prenons le cas simple de deux MAP, l'une active (MAP_a) et l'autre inactive (MAP_i). La MAP_a possède une seule branche suspendue (*S*), et exécute une branche *A*. Les temps de l'exécution séquentielle sont:

- TC_a: le temps d'exécution résiduel de *A*;
- TC_s: le temps d'exécution de S.

L'installation de *S* sur la MAP_i implique deux coûts additionnels:

- TE_s: le temps d'exportation associé à la MAP_a;
- TI_s: le temps d'importation associé à la MAP_i.

Ces coûts d'installation comprennent tout ce que chaque MAP réalise: préparation pour le transfert, sur-coût de communication et mise-à-jour de l'état des MAP.

Ainsi, le parallélisme est plus efficace que l'exécution séquentielle si l'expression suivante (*CP*) est vraie:

$$TC_a + TC_s > \max(TC_a + TE_s, TC_s + TI_s) \quad (CP^1)$$

La condition *CP* peut être divisée en deux:

$$TC_a + TC_s > TC_a + TE_s$$

$$TC_a + TC_s > TC_s + TI_s$$

En simplifiant:

$$TC_s > TE_s \quad (CP_e)$$

$$TC_a > TI_s \quad (CP_i)$$

La condition CP_e est simple: il ne faut pas exporter une tâche si son temps d'exportation est supérieur à son temps de calcul.

La condition CP_i est moins intuitive: il ne faut pas exporter une tâche si son temps d'importation est supérieur au temps résiduel d'exécution de la tâche restante. La MAP_i commencerait le calcul de *S* après la fin de *A* par MAP_a si CP_i n'est pas vraie. La figure 3.9 est un schéma des cas où CP_e et CP_i sont respectées et violées.

Les temps *TE* et *TI* peuvent souvent être exprimés comme des fonctions de la taille de certaines données de l'état actuel de la MAP_a. Ces fonctions dépendent des caractéristiques de la MAP utilisée et de la méthode utilisée pour la gestion de contextes multiples. Par exemple, dans le cas d'une implantation par copie de piles, ce temps est fonction linéaire de la taille des piles à transférer.

La difficulté majeure de l'utilisation de *CP* provient du calcul des *TC*. Une prédiction exacte nécessite une exécution complète puisque les *TC* dépendent des arguments des sous-buts de chaque exécution. L'estimation de ces temps à la compilation est un problème ouvert ([DEB 90]); son but serait de produire une fonction de calcul de ces temps, qui serait évaluée à l'exécution.

Dans l'attente d'une telle méthode, il faut se contenter d'estimateurs plus grossiers, sacrifiant l'exactitude à la simplicité. Ce sont de simples heuristiques et il est nécessaire de vérifier qu'elles garantissent bien l'augmentation des performances pour la plupart des exécutions des programmes. Ces approches suivent deux voies:

- l'introduction d'annotations par le programmeur dans le programme, de façon à éviter l'exécution parallèle de certains prédicats;
- la prédiction dynamique et approximative de la taille de chaque tâche, éventuellement combinée à une analyse statique simple qui produise des poids sur la complexité relative des clauses.

Par exemple, on peut prendre comme métrique le sous-but, et estimer un temps de calcul par la longueur d'une résolvante (branche OU), ou encore prendre le nombre de choix possibles

---

[1] (*CP*) compare l'exécution parallèle (2 MAP parallèles) à l'exécution séquentielle (1 MAP parallèle) sur une machine parallèle. La comparaison à une **implantation séquentielle** sur un processeur (MAP séquentielle) doit corriger le terme gauche de l'inéquation par le facteur de sur-coût dû à l'implantation de la MAP parallèle.

dans un nœud OU et estimer le temps de calcul d'une tâche par le nombre de choix à explorer. Dans ce cas, une idée simple de contrôle serait de ne pas permettre l'exportation d'une tâche si elle même, ou celle qui reste à la MAP exportatrice, n'ont pas un nombre minimal d'alternatives ou de nœuds OU. Les faiblesses de cette méthode résident dans le risque:

- de ne pas exploiter le parallélisme potentiel parce qu'une tâche très lourde possède un nombre de clauses inférieur au nombre défini par le contrôle;
- de diminuer la performance parce qu'une tâche très simple, mais composée d'un nombre élevé de clauses, est exportée.

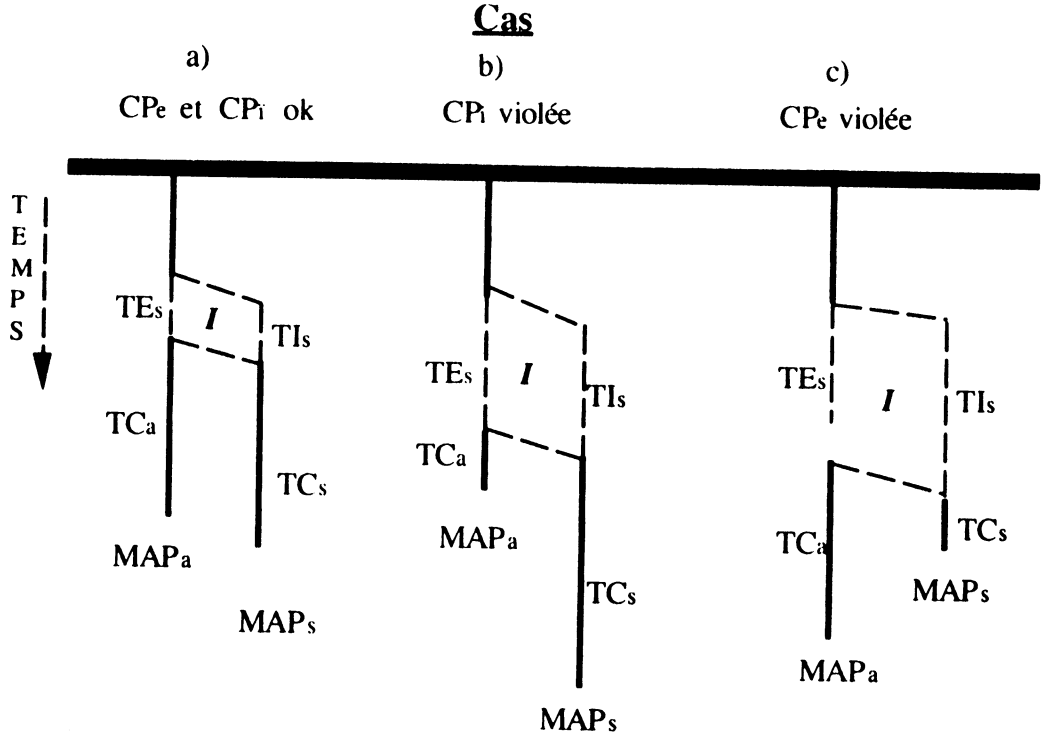

Fig. 3.9 - Cas des conditions minimales

On pourrait espérer une diminution du nombre de mauvaises décisions, c.a.d. une amélioration du contrôle en affectant un poids à chaque choix (clause). Celui-ci serait estimé par la longueur de la branche correspondante.

Dans tous les cas, l'estimation de ces temps à l'exécution, ainsi que son utilisation pour le contrôle ne doit pas dégrader les performances. Ainsi, il faut concevoir une structure de données de prédiction, qui permette des accès efficaces au moment du contrôle. L'accès à des informations, sur toutes les clauses suspendues d'un nœud OU, exige en général quelques modifications dans l'implantation d'une MAP, du fait qu'un nœud OU ne maintient que la prochaine clause à exécuter. En outre, le *TC* d'une alternative suspendue est déterminé par la clause correspondante et par les sous-buts à droite dans la résolvante courante (courante au moment de la création du nœud). Donc, si les informations statiques ne concernent que la clause, la prédiction du *TC* total exige soit un accès aux informations des prédicats appelés par la suite dans la résolvante, soit le maintien du *TC* relatif à la résolvante. Etant donné que le contrôle doit considérer plusieurs nœuds OU de la MAP_a active, il faut maintenir le *TC* de la résolvante par nœud. Il faut encore observer qu'au moment du contrôle, la MAP_a a déjà parcourue une partie de la branche, et donc qu'une partie du temps TC_a est écoulée.

3.6.2. La Régulation de la Charge

Le deuxième objectif de l'ordonnancement est d'assurer, pour un nombre donné de processeurs, l'exécution la plus efficace possible. D'une manière générale, une MAP est classée dans l'un des trois états suivants:

- inactive (MAP_i): elle ne possède aucune branche à exécuter;
- surchargée (MAP_s): elle est active et possède un ensemble *ES* d'alternatives exportables, appartenant à 1 ou plusieurs nœuds OU. *ES a* au moins un sous-ensemble qui satisfait les conditions minimales du parallélisme;
- muette (MAP_m): elle est active, mais *ES* est vide ou il ne possède pas de sous-ensemble qui satisfait les conditions minimales du parallélisme.

On peut distinguer trois cas de régulation de charge (si on ignore les MAP muettes):

- une MAP surchargée (MAP_s) et une MAP inactive (MAP_i);
- plusieurs MAP_s et une MAP_i;
- une MAP_s et plusieurs MAP_i.

Cas 1 MAP_s et 1 MAP_i:

Considérons tout d'abord le cas le plus simple. Dans le but d'obtenir la meilleure performance, il faut choisir une tâche *e*, entre les sous-ensembles de *ES*, qui minimise la durée d'exécution du programme, c.a.d. le maximum des temps totaux d'exécution de deux MAP. Ceci est exprimé par:

$$\min(\max(TC_a + TE_e, TC_e + TI_e)) \quad (CP_2)$$

où TC_a est le temps résiduel de la tâche restante à MAP_s après exportation, et TC_e, TE_e et TI_i sont respectivement les temps d'exécution, d'exportation et d'importation de la tâche *e*. TC_a est calculé à partir du temps résiduel d'exécution TC_t de la MAP_a exportatrice:

$$TC_a = TC_t - TC_e$$

Ainsi TC_e, TE_e et TI_e peuvent être déterminés à partir de chaque *e*. La difficulté de l'utilisation de cette expression est due à:

- la complexité du calcul des *TC*;
- la possibilité d'avoir un grand nombre de $e \in ES$.

Si on considère que les *TE* et *TI* sont égaux (ou voisins) et constants (par rapport à la tâche *e*), CP_2 devient:

$$\min(TC_a - TC_e)$$

soit

$$TC_a = TC_e = TC_t/2$$

Cette expression indique que les deux MAP terminent leurs tâches au même moment, sous réserve que des nouvelles installations de tâches ne se sont pas produites.

Dans ce cas, le contrôle de performance doit calculer un $e \in ES$ qui divise TC_t en deux (avec éventuellement une compensation due à une différence entre *TE* et *TI*). Cette simplification n'élimine pas le problème du calcul de *TC*. En effet, une division par deux du nombre d'alternatives *N* ne correspond pas à une division par deux de TC_t.

Cas N MAP_s et 1 MAP_i:

Dans le cas où nous disposons d'une MAP inactive et de N MAP surchargées, il faut choisir entre les *e* de plusieurs MAP_s, en minimisant le maximum de toutes les MAP. Le but est encore le même: aller plus vite avec $N_a + 1$ qu'avec N_a machines. La solution est simple si on considère résolu le problème de l'évaluation des *TC*: On choisit la MAP_s qui a le plus grand TC_t, à la condition que CP_e et CP_i soient vérifiées. Ensuite, on revient à la première situation: une MAP_s et une MAP_i.

Cas 1 MAP_s et N MAP_i:

Dans le cas où il existe une MAP surchargée pour plusieurs MAP inactives, il faut diviser de façon équilibrée la charge totale de la MAP_s entre plusieurs MAP_i. Le but est de réduire le coût total d'installation, en considérant toutes les exportations, ce qui augmente la complexité du problème.

En résumé, le coût d'une méthode de régulation de charge, qui chercherait une solution optimale, serait donc très élevé (en plus d'une définition complexe).

Là aussi des méthodes simples de régulation de charge ont été proposées et implantées. Ces méthodes déterminent la tâche à exporter soit par le critère du *TC* le plus élevé, soit par le critère des *TE* et *TI* les plus bas. Les *TC* et les *TE/TI* sont évalués par des heuristiques comme:

- le nombre de nœuds OU;
- le nombre d'alternatives;
- la profondeur des nœuds OU;
- la proximité dans l'arbre OU entre la MAP_s et la MAP_i.

3.6.3. Quelques Méthodes de Régulation de Charge

Cette section résume quelques méthodes de régulation de charge proposées dans la littérature.

Une stratégie de régulation de charge peut être divisée en trois parties, en reprenant la classification des états des MAP utilisée dans la section précédente:

- l'observation de l'état de chaque MAP;
- l'appariement d'une MAP surchargée (MAP_s) avec une MAP inactive (MAP_i);
- la détermination de la tâche à exporter de la MAP_s.

Le deuxième partie présente deux possibilités:

- une MAP_s cherche une MAP_i;
- une MAP_i cherche une MAP_s.

Evidemment, afin de diminuer le temps d'inactivité des processeurs, l'appariement doit être réalisé promptement. La deuxième solution est en général préférable puisqu'elle évite des sur-coûts dans les MAP_s. Néanmoins, une recherche constante par plusieurs MAP_i peut provoquer des sur-coûts généraux élevés, quand les MAP_i sont nombreuses et les MAP_s non.

Un exemple concret est la situation où, toutes les MAP sont soit inactives (*I*), soit muettes (*M*). Avant qu'une MAP muette devienne surchargée, il est peut-être préférable de laisser les MAP_i en état d'attente périodique, afin d'éviter des accès inutiles à la mémoire ([BUT 88]). Ainsi dans ce cas, l'unique MAP_s peut commencer la recherche avant les MAP_i, lorsque elle devient surchargée. Elle peut trouver un appariement meilleur, et avec un sur-coût plus petit, que l'appariement produit par l'une des MAP_i en attente ([CAL 89]). Dans la solution inverse, où toutes les MAP_i cherchent la MAP_s, le choix est en principe déterminé par la MAP_i la plus "rapide".

Plusieurs méthodes utilisent une représentation de l'arbre OU courant ([BUT 88], [CAL 89], [CHA 89a]). Cette représentation implique un sur-coût variable selon le type de machine. Il est presque nul dans le cas des machines à mémoire commune, puisque cette représentation est une extension simple de la pile de nœuds OU de la MAP. Une MAP_i recherche un nœud OU avec des alternatives suspendues, en parcourant l'arbre OU. Dans une machine sans mémoire commune, il n'existe pas de représentation complète et unique de cet arbre, mais des représentations partielles. Une représentation indépendante et spécifique de cet arbre devient alors nécessaire.

Une méthode simple d'appariement est une recherche en profondeur d'abord dans l'arbre OU jusqu'à trouver un nœud qui contient des alternatives suspendues. Elle a été utilisée dans le projet PEPSys ([CHA 89a]) avec des résultats médiocres. La recherche, effectuée par la MAP_i, commence à partir de la racine dans le cas général. Cependant, afin de limiter les trous noirs (conforme la section 3.3.1 "La Gestion de la Mémoire"), elle est initialement réalisée dans les sous-arbres des MAP_a filles de la MAP_i, c.a.d. celles qui ont importé une branche de la MAP_i. Un autre régulateur de charge de PEPSys, nommé "Robin des Bois", présente des meilleurs résultats. Il utilise deux critères de minimisation relative des coûts d'installation:

- critère d'installation: d'abord la MAP_i cherche une tâche dans les MAP_s qui sont filles de la MAP_i;
- critère de charge: le premier critère échouant, la MAP_i choisit la MAP_s qui a le plus grand nombre d'alternatives (ce qui exige la maintenance d'un compteur spécial).

Dans le projet Aurora trois régulateurs de charge, nommés Argonne, Manchester et Wave-front, ont été initialement proposés. Tous les trois, utilisant l'arbre OU, ont des caractéristiques communes:

- la première caractéristique est que la recherche est effectuée dans une partie des nœuds OU, dénommés **publics**, les autres étant des nœuds **privés**. Le principal critère de choix d'un nœud public par une MAP_i est la proximité de ce nœud à la branche que la MAP_i exécutait. Une seule alternative est exportée;
- la deuxième caractéristique est l'existence d'un seul nœud public par branche, ce qui correspond à un parcours en largeur d'abord de l'arbre OU. Un nœud OU devient public quand son nœud père (public) n'a plus d'alternatives suspendues (initialement un seul nœud, le premier, est public).

Les différences entre les régulateurs d'Aurora se situent dans les structures auxiliaires. Le régulateur d'Argonne ([BUT 88]) n'utilise aucune donnée globale. Chaque nœud public contient une indication de l'existence de tâches exportables dans le nœud ou dans ses fils. Une MAP_i cherche un nœud public d'abord dans les branches sœurs. Les MAP inactives ne restent pas en attente tout le temps: elles se réveillent périodiquement pour vérifier si une MAP active a créé un nœud public. A cet instant toutes les MAP_i commencent à se diriger vers ce nœud.

Le régulateur de Manchester maintient des structures globales, parmi lesquelles les principales sont deux tableaux de MAP:

- le premier indique les MAP_i et leur coût d'installation à partir de la racine;
- le deuxième enregistre le nœud public de chaque MAP_s et leur coût d'installation à partir de la racine.

Un tableau de bits est associé à chaque nœud public. Le tableau montre les MAP qui travaillent sur les branches dérivées de ce nœud. L'implantation est plus complexe et son sur-coût à l'exécution est plus grand. On note deux avantages:

- la sélection d'un nœud public par une MAP_i est plus efficace du fait de l'utilisation du tableau de bits et des coûts d'installation du tableau de MAP_s;

- en cas d'absence de MAP_s, les MAP_i s'arrêtent et la recherche d'une MAP_i est effectuée par la MAP_s qui crée le premier nœud public.

Le régulateur Wave-front ([BRA 88]) n'utilise pas de données globales mais maintient une liste linéaire à double chaînage des nœuds publics, ce qui permettrait une recherche plus rapide des nœuds publics.

De même la régulation de charge du modèle Muse ([ALI 90]) utilise le critère de la proximité entre MAP_i et MAP_s. Cependant, à l'opposé de la plupart des modèles où une seule alternative est exportée, un certain nombre de nœuds OU deviennent accessibles aux deux MAP. Ceci évite de nouvelles exportations jusqu'à l'épuisement de ces nœuds. Ce partage introduit un sur-coût de synchronisation et exige une mémoire commune pour la représentation de ces nœuds.

Dans le modèle BOPLOG ([TIN 87]) on choisit la MAP_s qui possède le plus grand nombre de nœuds OU. Ce nombre est une indication approximative de la charge de la MAP_s mais pas de la tâche transférée.

Dans le modèle Orbit ([YAS 84]), une proposition de machine sans mémoire commune, le régulateur de charge est un processus centralisé qui reçoit périodiquement des informations sur l'état de chaque MAP. A chaque demande d'importation par une MAP_i, le choix d'une MAP_s est réalisé en fonction de son nombre de nœuds OU. Une même MAP_s peut recevoir une suite de demandes d'exportation. La MAP_s divise sa pile de nœuds OU en fonction du nombre de demandes (équipartition).

Dans le modèle Kabu-Wake, des informations sur l'état de chaque MAP circulent sur un anneau de contrôle. Chaque MAP_s choisit la MAP_i dont l'état passe en premier.

Il faut remarquer que plusieurs systèmes, principalement ceux dont le niveau d'implantation est plus avancé comme Aurora et PEPSys, utilisent des annotations qui, au moins, évitent l'exportation des certains nœuds ou des alternatives dont le programmeur connaît par avance la faible granularité.

3.7. Les Opérateurs Séquentiels de Prolog

Ces opérateurs sont de deux types:

- les opérateurs de contrôle de l'arbre de recherche, comme la coupure;
- les prédicats prédéfinis à effet de bord, comme les opérations d'insertion/exclusion dynamique de prédicats et de données (*assert/1*, *retract/1*, ...), ou les opérations d'entrée/sortie.

La coupure, étant définie pour le contrôle de l'arbre de recherche en stratégie séquentielle, est donc par nature très liée à celle-ci. Les autres prédicats sont en fait des opérateurs d'affectation classique, et, par la même, la cohérence ne peut être assurée que par le respect d'un ordre dans ces affectations, ordre donné par la stratégie séquentielle.

3.7.1. La Coupure

Deux approches ont été proposées pour la coupure en parallèle, parfois dans le même modèle:

- l'introduction de nouveaux opérateurs de coupure dont la sémantique est mieux adaptée au parallélisme ([ALI 90], [BUT 88], [CAL 89]);
- des méthodes spéciales pour l'implantation de la sémantique classique de la coupure.

Nous allons nous restreindre à résumer la deuxième approche, qui est nécessaire pour la préservation des programmes séquentiels existants. Il est encore possible de diviser cette approche en deux:

- les alternatives conditionnées par une coupure sont suspendues jusqu'au moment où il est certain que la coupure ne sera pas exécutée;
- les mêmes alternatives sont activées en parallèle et, éventuellement, tuées au moment de l'exécution de la coupure.

La première idée correspond à une inhibition du parallélisme. Les méthodes d'implantation ([ALI 87], [YAS 84]) ne sont pas simples mais elles n'exigent que des modifications localisées dans la compilation de Prolog et dans la MAP. Le but de ces modifications est la production d'informations qui permettent une vérification rapide de la condition d'exportation d'une alternative, sans un sur-coût important pour le calcul séquentiel. La condition est la non-existence d'une coupure qui risque de supprimer l'alternative. Le désavantage de cette approche est une inhibition inutile du parallélisme, si la coupure n'est pas exécutée à cause d'un échec dans un sous-but qui précède la coupure.

La difficulté principale de l'implantation de la deuxième idée, activer et couper à posteriori, vient de la possibilité d'exécuter une coupure en parallèle qui ne serait pas exécutée en séquentiel ([HAU 88]). Dans ce cas la coupure coupe des alternatives qui n'en sont pas en séquentiel. Ce problème peut se produire quand une coupure c_c est conditionnée par une autre coupure c_m. En s'exécutant, c_m coupe certaines alternatives et peut provoquer l'échec total d'un sous-but qui précède c_c, ce qui interdit l'exécution de c_c. En parallèle il faut prévoir cette situation et retarder l'exécution de c_c. La figure 3.10 illustre cette possibilité.

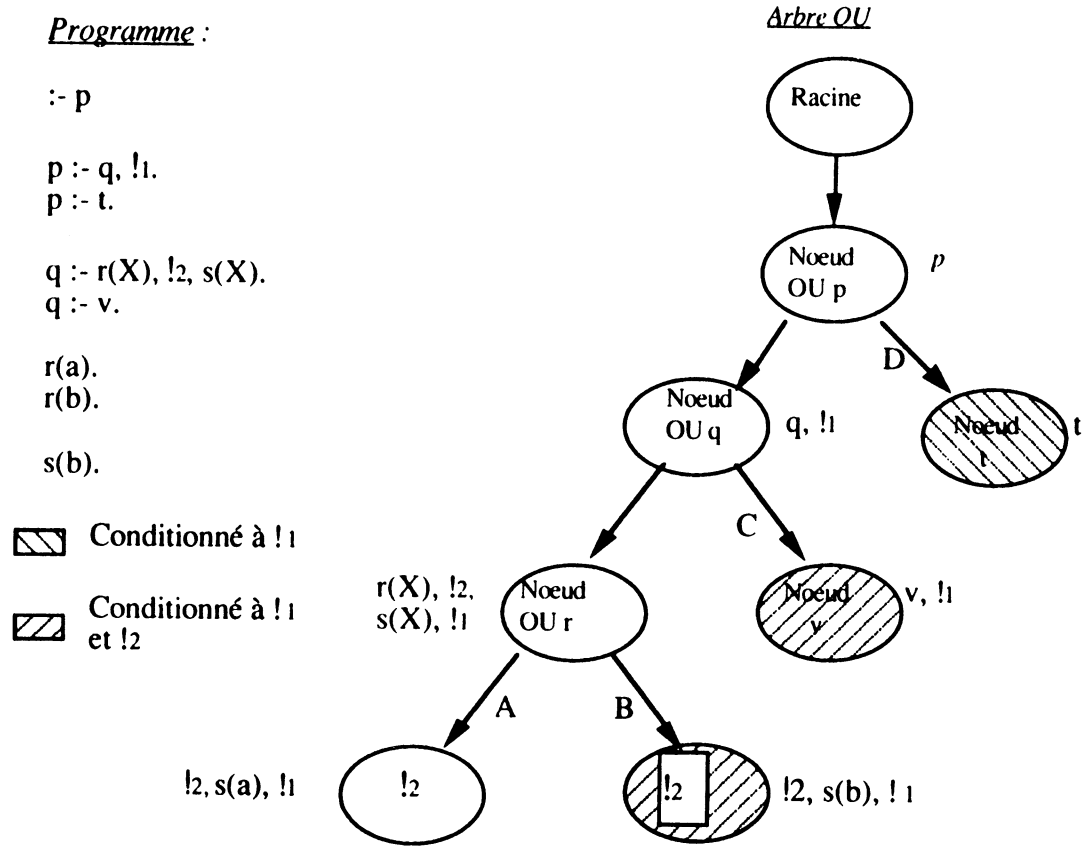

Fig. 3.10 - Coupure conditionnée

En séquentiel, la branche *A* exécute la coupure $!_2$ et coupe les branches *B* et *C*. Ensuite, le sous-but *s(b)* échoue, $!_1$ n'est pas exécuté et le contrôle se poursuit avec la branche *D*. En

parallèle les branches *B* ou *C* peuvent tuer la branche *D* si $!_1$ est exécuté avant que la branche *A* exécute $!_2$. Dans cet exemple $!_1$ est dépendent de $!_2$.

Une solution à ce problème serait de détecter les dépendances d'une coupure. La solution la plus simple et la moins efficace suspend l'exécution d'une coupure si sa branche n'est pas la branche la plus à gauche dans l'arbre courant, c.a.d. on considère que toutes les coupures sont conditionnées par une autre coupure. Des solutions plus efficaces exigent une analyse de l'arbre courant afin de déterminer les branches qui seront coupées par la coupure en exécution, aussi bien que par des coupures dans les branches situées plus à gauche. Ces branches peuvent être coupées immédiatement, la coupure des autres branches étant soumise à des modifications dans l'arbre.

Les branches, dont la validité est conditionnée par la non-exécution d'une coupure, sont dénommées **tâches spéculatives**. Dans une implantation de la coupure du type activer/couper, elles pourraient être activées et ensuite tuées. Le régulateur de charge pourrait exploiter cette possibilité en leur donnant une priorité inférieure à celle qu'il donne aux autres alternatives.

3.7.2. Les Effets de Bord

L'implantation parallèle des prédicats à effet de bord exige certains contrôles si la sémantique séquentielle doit être respectée.

Dans beaucoup de programmes l'ordre d'entrée ou de sortie des données de l'exécution séquentielle doit être maintenue dans l'exécution parallèle. Des exemples sont la lecture du programme source et l'écriture du fichier objet dans un compilateur. Par contre, dans d'autres cas, les solutions diverses à un même problème peuvent être imprimées dans un ordre quelconque.

Le besoin de respecter l'ordre de l'exécution séquentielle apparaît encore dans les programmes qui utilisent les prédicats qui modifient le programme, par exemple, *assert/1* et *retract/1*.

Deux possibilités peuvent donc être envisagées dans une implantation parallèle pour les prédicats à effet de bord:

- respecter l'ordre d'exécution séquentielle de ces prédicats;
- exécuter ces prédicats dans un ordre quelconque, c.a.d. en ignorer le problème.

Une implantation de ces deux possibilités exige deux syntaxes, du fait que le choix de la sémantique est à charge du programmeur. Chaque prédicat doit avoir deux versions, chacune avec un nom différent. L'implantation de la deuxième version est identique à celle de l'exécution séquentielle.

Une solution ([HAU 88]) qui préserve l'ordre séquentiel de Prolog est la suivante:

- l'exécution du prédicat est suspendue si sa branche n'est pas la branche la plus à gauche dans l'arbre courant;
- l'exécution est reprise quand sa branche devient la branche la plus à gauche.

Cette solution exige des mécanismes de blocage et réveil d'une MAP, et exige une fonction de vérification de la position relative d'une branche dans l'arbre courant. L'implantation de cette fonction est simple si l'arbre courant est déjà représenté, comme dans les modèles sur machines avec mémoire commune.

3.8. Les Résultats

Les divers systèmes implantés présentent deux caractéristiques importantes:

- ils offrent un ensemble raisonnable de prédicats prédéfinis;
- ils partent d'une implantation séquentielle de Prolog ayant un bon niveau de qualité et de performance.

Ces caractéristiques ont permis une évaluation de ces prototypes sur des programmes réels.

Des programmes identiques ont été utilisés pour l'évaluation des prototypes des modèles Aurora ([SZE 89], [MUD 89]), Muse ([ALI 90]) et PEPSys ([CHA 89a]). Ces programmes sont divisés en trois groupes en fonction de l'augmentation de performance obtenue:

- augmentation élevée (groupe *H*): le puzzle *sel et moutarde* de Lewis et Carrol, le problème de placement de 8-reines et une application touristique (*Tina*), développée à l'ECRC;
- augmentation moyenne (groupe *M*): 5 buts différents soumis au programme CHAT80, lequel est une application de traitement du langage naturel, et le puzzle *house* (ou *zebra*);
- augmentation basse (groupe *B*): 2 autres buts du CHAT80 et le puzzle *farmer*.

Les trois prototypes implantent un Prolog compilé. Le compilateur SICStus ([CAR 88b]) est utilisé dans Aurora et Muse; PEPSys utilise un compilateur propre au projet. La MAP, dans les trois cas, est dérivée de la WAM. Son interpréteur est écrit en langage C.

La table 3.1 présente un résumé des résultats des évaluations suivantes:

- Aurora avec le régulateur de charge Manchester, sur la machine Sequent Symetry S27, la configuration étant de 12 processeurs et de 16 MB de mémoire commune;
- Muse (A) sur la machine Sequent Symetry S81, avec une configuration de 16 processeurs et de 32 MB de mémoire commune;
- Muse (B) sur une machine expérimentale, ayant 6 processeurs, une mémoire locale et une autre globale;
- PEPSys sur la machine Siemens MX-500 (Sequent Balance 8000): la configuration est de 8 processeurs et de 16 MB de mémoire commune.

Les sur-coûts de Aurora et de Muse sur un processeur par rapport au système SICStus séquentiel sont respectivement 29% et 5%.

Aurora a encore été évalué sur la machine Butterfly GP1000, avec 40 processeurs, laquelle possède une mémoire locale pour chaque processeur et une mémoire globale. Les premiers résultats, déjà publiés, sont moins bons:

- 8-reines: augmentation linéaire jusqu'à 10 processeurs;
- 10-reines: augmentation linéaire jusqu'à 40 processeurs.

Evaluation	Nb. de Proces.	Groupe H	Groupe M	Groupe B
Aurora	11	9,4 à 10,5	5,13 à 7,7	2,12 à 3,14
Muse (A)	15	13,2 à 14,5	5,2 à 8,7	2,2 à 2,4
Muse (B)	6	5,6 à 5,9	3,7 à 5,1	1,9 à 2,4
PEPSys	8	5,4 à 7,1	3,9 à 4,6	2 à 3

Table 3.1 - Résultats de Aurora, Muse et PEPSys

Dans l'évaluation de Muse, d'autres applications réelles, ayant une bonne granularité pour le parallélisme OU, ont été utilisées, toujours avec des résultats équivalents à ceux du groupe H de la table 3.1. Les exemples sont un système de traitement du langage naturel, un programme pour la preuve de théorèmes, un interprétateur d'un langage gardé.

Le modèle Kabu-Wake, implanté à partir d'un interpréteur Prolog, a été évalué sur une machine expérimentale, avec 16 processeurs et sans mémoire commune. Cette machine possède 2 réseaux d'interconnexion, le premier étant utilisé pour la régulation de charge, le deuxième pour l'installation de tâches ([SOH 85], [KUM 86], [MAS 86]). Les programmes utilisés sont le N-reines et un analyseur syntaxique de la langue japonaise. Les résultats, pour le premier et pour des phrases complexes soumises au deuxième, sont de l'ordre de 11 à 12 sur 13 processeurs. Par contre, une phrase simple permet seulement un gain de 4 sur 13 processeurs.

3.9. Conclusions

Les principales conclusions de ce chapitre sont:

- le parallélisme OU multi-séquentiel est plus simple à implanter que les autres;
- il permet l'utilisation des optimisations du modèle séquentiel;
- il a une granularité plus élevée que celle du parallélisme ET;
- il existe dans diverses applications réalistes;
- l'ordonnancement est important du fait de la faible granularité de certaines branches de l'arbre OU (par rapport au coût d'installation). Cependant il est très difficile de concevoir un bon ordonnanceur en raison de l'impossibilité (actuelle) d'évaluer précisément la charge de chaque branche;
- une solution partielle à ce dernier problème est l'introduction de pragmas sur la granularité des branches, ce qui restreint le parallélisme implicite de Prolog (qualité importante pour la programmation parallèle);
- les opérateurs séquentiels de Prolog exigent des solutions qui restreignent le parallélisme.

4. Chapitre 4
Présentation d'Opera

Cette partie est consacrée au système Opera ([BRI 90a], [BRI 90b], [BRI 91]). Elle débute par une présentation des objectifs initiaux du projet et de ses hypothèses de travail (1). Ensuite, se trouve une description des principes du système Opera et des principaux problèmes rencontrés (2). Une attention particulière est portée à la gestion des contextes multiples (3). La présentation d'une implantation parallèle de la coupure (4) précède l'exposé du problème de la régulation de charge dans Opera (5). Cette partie s'achève par une description rapide de l'architecture du logiciel Opera (6).

4.1. Le Projet

Cette section présente brièvement le projet Opera, développé par l'équipe Flop, du laboratoire LGI. Elle en décrit les objectifs, les contraintes spécifiques, les décisions techniques principales et introduit le problème de l'ordonnancement.

4.1.1. Objectifs et Choix Initiaux

L'équipe Flop avait pour thème de recherche l'exécution efficace de programmes Prolog. Le parallélisme était une des voies à explorer. Parmi les sources de parallélisme possibles en Prolog, le parallélisme OU a été choisi, car:

- il est le plus simple à implanter;
- il offre un grain de parallélisme plus gros que le parallélisme ET.

Une deuxième décision importante a été d'adopter le parallélisme implicite, afin d'obtenir un accroissement de performance des programmes existants, sans les modifier. Le langage Prolog n'est donc pas changé.

Parmi les architectures parallèles possibles, une architecture sans mémoire commune a été choisie. La machine cible, pour l'implantation du prototype, est le Supernode: un réseau de processeurs communicants, développé dans le cadre du projet Esprit P1085. Les raisons de ce choix sont:

- l'intérêt de la nouveauté: la programmation de ces machines n'est pas maîtrisée comme celle des machines à mémoire commune, et il existe peu d'environnement de programmation;
- la collaboration au développement d'une machine européenne;
- les machines de ce type présentent un nombre de processeurs généralement plus élevé que celui des machines à mémoire commune.

4.1.2. Hypothèses de Travail

Le système Opera a été conçu à partir de trois hypothèses de travail:

- la supériorité du modèle multi-séquentiel relativement à la technologie actuelle des machines parallèles;
- la maîtrise de coûts induits par le parallélisme (communication, installation de tâches) par une architecture du système, appropriée à la machine parallèle;
- la production d'un gain de performance par un ordonnancement de l'exécution des tâches (contrôle du parallélisme).

Notre première hypothèse est relative à l'opposition entre modèle théorique et modèle multi-séquentiel. Nous pensons que, présentement, seul le modèle multi-séquentiel permet un

accroissement significatif de performance, du fait des caractéristiques des machines sans mémoire commune actuelles.

Rappelons le principe de base des modèles multi-séquentiels: l'exécution est initialement séquentielle, et elle devient parallèle s'il existe des processeurs libres. Une tâche parallèle est l'exécution d'une résolvante. Les tâches parallèles sont créées paresseusement ou à la demande.

Dans un modèle théorique de parallélisme OU, une tâche est l'exécution d'une clause. Elle ne comprend pas le reste de la résolvante, qui suit la clause (ou le sous-but correspondant). Le coût de parallélisation d'une clause peut s'avérer en général beaucoup plus élevé que son coût d'exécution. En effet, le degré du parallélisme n'est pas contrôlé. Chaque clause d'un prédicat donne lieu à une tâche (création vivace de processus). Ceci risque soit d'épuiser la mémoire de la machine, soit d'étrangler son réseau de communication.

Donc, le coût total de parallélisation d'un programme donné, dans le modèle théorique, est plus élevé que dans le modèle multi-séquentiel. En plus, l'application des optimisations du modèle séquentiel peut être difficile dans l'implantation d'un modèle d'exécution radicalement différent ([HER 86c], [TIC 84]). L'exécution des sections déterministes est aussi plus efficace dans le modèle multi-séquentiel.

Un contre-argument à ce raisonnement est que le modèle théorique permet de trouver un nombre de tâches parallèles plus élevé que le multi-séquentiel, principalement en combinant les parallélismes ET et OU, comme dans [CON 85]. Il serait alors plus adapté à des machines dont le nombre de processeurs est de l'ordre, par exemple du millier.

Pour choisir entre ces deux solutions, il nous faut tenir compte du coût d'installation d'une tâche parallèle. L'accroissement de performance d'exécution d'un programme donné, en fonction du nombre de processeurs, est limité par le fait qu'une partie des tâches présente un coût d'installation annulant tout gain de performance par parallélisation.

Notre première hypothèse de travail est donc que le modèle multi-séquentiel est plus approprié aux caractéristiques des machine parallèles actuelles.

Degré de parallélisme et coût d'installation de tâches sont modérés sur les machines à mémoire commune et forts pour les machines sans mémoire commune (à mémoire distribuée).

Ceci est particulièrement vrai pour la famille Supernode ([HAR 86], [MUN 89], [WAI 90], [TOU 90], [CAR 89]), dont le degré de parallélisme peut atteindre 1024, mais où l'absence de mémoire commune, alliée à des communications relativement lentes, rend le partage d'information et l'installation de tâches très coûteux.

Certains traits techniques de ce type de machine doivent être utilisés afin de garder ces coûts dans des limites raisonnables. Ainsi le Transputer ([INM 89]), qui est le processeur de base du Supernode, comporte des circuits de communication:

- de débit croissant avec le volume de données échangées, du fait d'un coût d'amorçage du transfert élevé;
- pouvant fonctionner en parallèle de l'unité de traitement, mais en la ralentissant par vol de cycles mémoire.

Par ailleurs, le Supernode présente un réseau d'interconnexion réconfigurable dynamiquement. Il permet d'ajuster les connexions aux besoins de la communication.

Notre seconde hypothèse de travail est qu'une bonne architecture du système Opera permettra de maintenir les coûts de communication et d'installation de tâches à un niveau, rendant possible l'exploitation effective du parallélisme intrinsèque de Prolog.

Cet aspect ne sera pas particulièrement développé dans notre travail. Il fait l'objet d'une présentation spécifique dans la thèse de Michel Favre.

Enfin, notre dernière hypothèse de travail est qu'un algorithme d'ordonnancement approprié permettra de garantir une efficacité d'exécution au moins égale à celle de l'exécution séquentielle.

Un tel algorithme de régulation parait nécessaire a priori, du fait du coût important d'installation de tâches parallèles. Sa difficulté de réalisation repose sur:

- la définition de bons estimateurs des temps de calcul de tâches;
- la définition d'une règle de répartition des tâches;
- l'utilisation d'une architecture parallèle (le Supernode) interdisant la construction d'un état global.

Plutôt que de conditionner la réalisation d'Opera à la définition des estimateurs et de la règle d'ordonnancement, nous avons préféré proposer une architecture du système prévoyant un ordonnanceur.

Cet ordonnanceur sera dans un premier temps une règle naïve dont le seul but est de pouvoir expérimenter et mesurer le comportement parallèle de programmes Prolog.

C'est à partir de ces observations que sera élaborée une stratégie garantissant un accroissement de performance.

4.1.3. Décisions Techniques Importantes

Deux décisions importantes ont été adoptées dès le début des études. La première est le choix d'une implantation compilée de Prolog. En effet, les sur-coûts d'installation du parallélisme peuvent être masqués par un calcul trop lent. Une implantation inefficace de Prolog augmente de façon artificielle les débits de communication. Dans ce cas les gains de performance du parallélisme seraient peu significatifs.

Prolog a été implanté par compilation en code natif. La machine abstraite, nommée TWAM (Transputer Warren Abstract Machine), est basée sur la Machine Abstraite de Warren (WAM) ([WAR 83]). Cette machine est considérée comme la plus efficace pour implanter Prolog. Elle est aussi la plus utilisée.

La deuxième décision concerne la gestion des contextes multiples. La méthode "copie des piles" a été choisie au détriment du partage des sections communes des piles. Ce choix est justifié par:

- l'absence d'une mémoire commune;
- l'inefficacité d'un accès non-local du fait du coût d'amorçage trop élevé pour le nombre de données à transférer à chaque accès;
- la fréquence élevé de ce type d'accès;
- la possibilité de transférer en parallèle du calcul, permettant ainsi l'annulation du coût d'exportation d'une tâche.

4.1.4. Résumé des Caractéristiques Principales

Les décisions principales, qui caractérisent le projet Opera, sont les suivantes:

- type de parallélisme: OU implicite;
- type de modèle d'exécution: multi-séquentiel;
- classe d'architecture parallèle: sans mémoire commune;
- implantation de Prolog: compilation en code natif;
- gestion des contextes multiples: copie.

4.2. Les Constituants Séquentiel et Parallèle

Comme tout système multi-séquentiel, on peut découper Opera en une partie qui relève de la technique de compilation et d'implantation séquentielle de Prolog, et une partie proprement parallèle qui repose sur la coopération d'exécutions séquentielles.

La structure de cette section suit ce découpage, la partie parallèle étant évidemment la plus importante.

4.2.1. La partie Séquentielle

Cette section résume l'implantation séquentielle de Prolog. Les points abordés concernent le langage, la machine abstraite, la compilation et les optimisations. Les annexes A et B approfondissent les points les plus intéressants.

4.2.1.1. Le Langage

Le langage Prolog n'a pas encore une définition standard officielle. La plupart des implantations présentent des différences, principalement au niveau syntaxique et au niveau des prédicats prédéfinis.

Il a été décidé d'adopter la syntaxe et la sémantique de C-Prolog [PER 88]. C-Prolog est issu du DECsystem-10 Prolog, lequel est considéré comme le standard de facto (définition Edinburgh). La sémantique séquentielle de la coupure a été conservée.

Le langage implanté s'est restreint à la partie Prolog pur. Quelques déclarations (optionnelles) ont été introduites dans le langage, pour permettre la compilation séparée (modules). Un jeu minimum de prédicats prédéfinis a été implanté. Les prédicats à effet de bord ne l'ont pas été, si l'on excepte la coupure (cf. section 4.4, plus loin). L'annexe A fournit une liste de tous ces prédicats.

4.2.1.2. Compilation et Machine Abstraite

Cette section présente rapidement le modèle d'implantation séquentiel de Prolog Opera. Le lecteur peut trouver, dans l'annexe B, un rappel de divers concepts de base, comme:

- un résumé des éléments de la WAM;
- une introduction à la compilation de Prolog vers la WAM;
- les optimisations de l'exécution séquentielle.

Prolog Opera a été implanté par compilation. Le compilateur Prolog transforme le source dans une séquence d'instructions de la machine abstraite Transputer Warren Abstract Machine (TWAM). La TWAM a été définie dans le cadre du projet Opera. Elle est une adaptation de la WAM ([WAR 83]), dans le but d'optimiser l'exécution séquentielle et parallèle. Le code TWAM est ensuite transformé en une séquence d'instructions de la machine cible (code natif).

Les caractéristiques principales de la WAM sont:

- l'unification est optimisée à partir de la connaissance des termes de la tête de la clause: une instruction spécialisée pour chaque type de terme (variable, constante, structure, liste);
- les arguments d'un appel (sous-but) sont transmis par des registres spéciaux, nommés arguments, et non par la pile comme il est usuel. Ils sont sauvegardés plus tard, si nécessaire lors d'un autre appel ou pour la mise à jour en cas de retour-arrière;
- au classique environnement d'exécution de procédure, s'ajoute un autre type de structure destinée au contrôle de l'arbre OU: le point de choix ou nœud OU.

4.2.1.3. Les Optimisations d'Opera (TWAM)

Les principales modifications introduites dans la spécification de la TWAM et dans la compilation de Prolog sont:

- les constantes ont été séparées en deux nouveaux types: entiers et atomes;
- les listes sont compilées comme une seule structure;
- la disjonction est compilée localement;
- une méthode de compilation uniforme est proposée pour diverses variations sémantiques de la coupure séquentielle;
- des nouvelles instructions d'indexation ont été ajoutées pour les constantes et les structures, en mélangeant les fonctions de retour-arrière et d'indexation par valeur.

Certaines d'entre elles proviennent de la machine PLM (une variation de la WAM). Ceci est dû au fait que le compilateur Opera est une version d'un compilateur initialement développé pour la PLM ([VAN 84]).

4.2.2. Le Modèle Opera

Opera est un système multi-séquentiel, c.a.d. un modèle où plusieurs machines séquentielles coopèrent au parcours de l'arbre OU.

Le problème principal de mise en œuvre de cette coopération est celui de la gestion cohérente de multiples contextes, c.a.d. celui des parcours de cet arbre ayant des parties communes.

4.2.2.1. Schéma de Coopération Parallèle

Rappelons le schéma de coopération parallèle entre des machines séquentielles Prolog (cf. chapitre 3, section 3.1 "Les Principes" du modèle multi-séquentiel OU):

- Opera est constitué d'un ensemble de N TWAM, N étant fonction des ressources en mémoire et processeurs de la machine cible;
- chaque TWAM poursuit une interprétation procédurale, en créant des nœuds OU et en faisant des retours-arrière quand elle arrive à une feuille de l'arbre OU;
- au début de l'exécution, toutes les TWAM étant inactives, une seule reçoit la résolvante initiale et devient active;
- chaque TWAM inactive devient active en acquérant des tâches suspendues dans les nœuds OU des TWAM actives;
- une TWAM active devient inactive quand, arrivant à une feuille, elle ne possède aucune alternative suspendue;
- l'exécution se termine quand toutes les TWAM deviennent inactives.

4.2.2.2. Le Problème de la Gestion des Contextes Multiples

La gestion des contextes multiples est une conséquence de la décision d'utiliser la méthode "copie des piles", décision elle-même justifiée par l'architecture sans mémoire commune.

Le problème principal de cette gestion est d'assurer l'efficacité du transfert d'une tâche d'une machine à une autre: c'est l'installation de tâche.

Cette installation de tâche se traduit en sous-problèmes (cf. chapitre 3, section 3.3 "La gestion des contextes multiples") d'efficacité:

- transfert de piles;

- implantation de la liaison conditionnelle;

ou de maintien de cohérence:

- des sections de piles de machines différentes correspondant à un parcours commun de l'arbre OU;
- des données de piles du fait du parallélisme entre transfert et calcul d'une machine abstraite.

Evidemment, ces problèmes sont liés. De plus, il faut éviter de dégrader l'exécution séquentielle d'une machine abstraite.

4.3. La Gestion de Contextes Multiples

Le problème principal est donc l'installation efficace d'une tâche. Cette installation doit:

- préserver l'efficacité de l'exécution séquentielle;
- utiliser au mieux le parallélisme des processeurs comme du réseau de communication.

4.3.1. Principes

4.3.1.1. Méthode Naïve

Le schéma d'installation naïf d'une tâche dans la méthode de copie est le suivant (cf. chapitre 3, section 3.4.1 "La copie simple de piles"):

- on transfère les points de choix, dont des alternatives sont exportées;
- les sections des piles globale et locale, protégées par les points de choix transférés, sont entièrement transférées;
- la pile traînée est entièrement transférée;
- la TWAM importatrice effectue une **déliaison non-locale** sur les sections de piles reçues, à partir de la portion de traînée créée depuis le point de choix transféré le plus récent.

Une telle installation naïve a des conséquences immédiates:

- un synchronisme entre transfert et exécution de la TWAM importatrice:
  - interdiction des retours-arrière au cours du transfert;
  - interdiction d'accès aux piles durant leur transfert (cohérence des données transférées).
- une inactivité de la TWAM importatrice car la déliaison non-locale ne peut procéder qu'après le transfert de toutes les piles;
- une redondance des transferts.

Une première redondance est de constater que le transfert total de la pile traînée implique celui de liaisons de variables non-transférées (créées postérieurement aux points de choix transférés).

Plus grave est le transfert de sections de piles déjà transférées chez la TWAM importatrice. En effet, plusieurs installations de tâches peuvent répéter le transfert des mêmes sections de piles vers une même TWAM. Observons qu'une TWAM importatrice a nécessairement des parties communes de piles avec une TWAM exportatrice (à l'exception du cas où la TWAM importatrice reçoit sa première tâche). Ces parties communes sont dues à des coopérations antérieures entre les deux TWAM, ou entre chacune des deux TWAM et une troisième TWAM (ou encore entre chacune des TWAM et deux autres TWAM, qui ont coopéré

avec une cinquième TWAM, etc...). Ces parties communes sont relatives au nœud OU commun (le plus récent) aux branches actuelles des deux TWAM. Evidemment, on considère la branche "avant-importation" pour la TWAM importatrice.

Eviter le transfert de cette section commune[1] est important, surtout quand une installation de tâche correspond à des nœuds OU très profonds dans l'arbre OU ([ALI 90]). Cette optimisation de la copie est nommée **copie incrémentale**.

4.3.1.2. Copie Incrémentale

Le principe d'installation d'une tâche parallèle en copie incrémentale devient alors:

- détermination des sections de piles protégées par les points de choix, dont des alternatives sont exportées (*A*). Ces sections correspondent aux nouvelles sections communes (voir fig.4.1);
- CI1: détermination des sections communes de piles déjà dupliquées (*B*). Ces sections correspondent à des coopérations antérieures;
- CI2: déliaison locale, par la TWAM importatrice, de liaisons invalides sur les variables des sections *B*. C'est l'équivalent du retour-arrière séquentiel;
- transfert par la TWAM exportatrice des sections non-dupliquées (*A* - *B*), des piles globale et locale;
- transfert par la TWAM exportatrice de la pile traînée, la section étant déterminée par "pile entière moins *B*";
- mise à jour des liaisons importées:

    - déliaison non-locale par la TWAM importatrice des liaisons invalides sur les sections *A* - *B*;
    - CI3: installation par la TWAM importatrice des liaisons valides, effectuées par la TWAM exportatrice, sur les sections *B* de la TWAM importatrice (on considère que les liaisons valides sur les sections *A* - *B* sont automatiquement installées par le transfert des piles globale et locale).

On remarque que les étapes CI1, CI2 et CI3 distinguent la copie incrémentale de la copie naïve, et que la copie incrémentale repose sur l'invariance des copies partielles de la section commune après la mise à jour des liaisons.

L'étape CI3 exige des procédés spéciaux de la part de la TWAM exportatrice, du fait que les liaisons en question sont disséminées dans les sections *B* des piles globale et locale. L'étape CI1 est due au fait que la WAM effectue une déliaison au retour-arrière, si elle a une alternative suspendue, ce qui n'est pas le cas de la TWAM importatrice.

4.3.1.3. Objectif d'une Implantation

L'objectif est clair: diminuer les coûts d'installation de tâches, ce qui se traduit par:

- OB1: limiter le volume de données à transférer;
- OB2: diminuer les contraintes de synchronisation des TWAM relativement au transfert;
- OB3: diminuer le temps de calcul spécifique à l'installation (déliaison non-locale, installation de liaisons importées, ...).

---

[1] Dans la suite, l'expression "section commune", au singulier, sera utilisée pour noter les sections communes de toutes les piles.

La solution, dans le premier cas (OB1), est immédiate: la copie incrémentale. Dans le second cas (OB2), il s'agit de proposer une représentation adéquate des piles et des variables pour permettre:

- à la TWAM exportatrice, de continuer le calcul pendant le transfert;
- à la TWAM importatrice, d'effectuer la mise à jour des liaisons importées dès que possible (pendant le transfert).

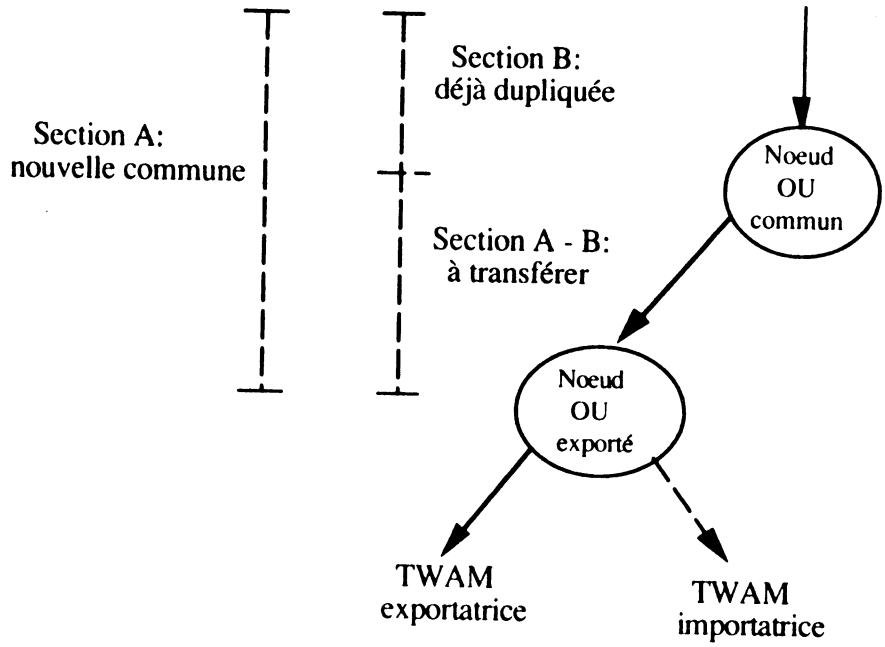

Fig. 4.1 - Sections communes

La solution, dans le dernier cas (OB3), est liée aux solutions apportées aux deux premiers cas, et dépend des fréquences d'apparition des structures de données (variables, liaisons) dans les différentes sections (*B*, *A* - *B*, ...).

Dans tous les cas, il est en principe préférable de reporter le calcul spécifique d'installation sur la TWAM importatrice (et pas sur la TWAM exportatrice), du fait qu'elle est oisive pendant l'installation.

Pour la représentation adéquate des piles et des variables, une solution possible est de regrouper toutes les variables dans une pile spécifique, ou tableau de liaisons (cf. chapitre 3, section 3.3.3, [WAR 87b]), qui peut:

- soit être transféré et délié;
- soit être reconstruit.

La déliaison, ou la reconstruction, de cette pile serait effectuée par la TWAM importatrice en parallèle au transfert des piles.

Ces deux possibilités conduisent aux deux variantes suivantes:

- VD: pile de variables plus datation (section 4.3.3);
- VV: pile de variables et traînée plus valeur (section 4.3.4).

Avant de procéder à une description précise des solutions proposées pour Opera, il nous faut indiquer les conséquences de cette décision, des points de vue parallèle comme séquentiel.

4.3.2. Problèmes

L'introduction de la copie incrémentale comme d'une pile de variables n'est pas de nature à changer fortement l'implantation séquentielle. Il nous faut cependant vérifier si les optimisations "classiques" de la WAM sont encore utilisables.

4.3.2.1. Liaison et Copie Incrémentale

L'objectif de la copie incrémentale est de ne transmettre qu'une seule fois une section commune vers une TWAM donnée. Ceci implique que, entre deux installations consécutives de tâche, une section commune ne doit pas avoir évoluée, une fois la mise à jour de liaisons effectuée. C'est apparemment exact au niveau logique, où une variable est libre ou liée. Par contre, on constate que ce phénomène peut se produire, si l'on examine comment les variables sont gérées par une WAM.

a) Allocation, initialisation et liaison:

Les variables Prolog sont implantées dans la WAM sous la forme de cellule, allouée dans la pile locale (environnement) ou dans la pile globale (variable d'un terme structuré). Elles sont accédées depuis une cellule d'environnement, directement ou via un pointeur.

La position dans l'environnement est calculée de façon statique à la compilation. L'environnement est alloué à l'entrée d'une clause (l'instruction *allocate*), et récupéré progressivement à chaque appel (optimisation de réglage d'environnement (*triming* [WAR 83]) et totalement avant le dernier appel (optimisation d'appel terminal(TRO [WAR 80]), cf. l'annexe B, section B.3.2 et B.3.3).

Une variable n'est cependant initialisée qu'au moment de sa première occurrence (les instructions *put_variable* et *unify_variable*) cf. l'annexe B, section B.3.1).

b) Divergence des copies partielles:

Du fait de la non-atomicité de l'allocation/initialisation, les copies d'une section commune peuvent évoluer de façon incohérente, au sens de la copie incrémentale. Par exemple, on peut se trouver dans la situation suivante:

Soit un prédicat *p()*:

p(...) :- q(...), r(X, ...), s(X, ...)

La $TWAM_1$ exécute la résolvante *p(...)* <...>, qui après l'appel de *p()* devient

q(...), r(X, ...), s(X, ...) <...>

L'environnement de *p()* est alloué, et *X* y a une position *"x"*.

Si l'on suppose que *q()* crée un point de choix, et qu'une alternative est prise par la $TWAM_2$, celle-ci effectue une copie des sections communes avec la $TWAM_1$, qu'elle ne possède pas déjà. Ces copies incluent nécessairement l'environnement de *p()*, donc la valeur actuelle de *X* (non-initialisée).

La $TWAM_1$ finit son alternative de *q* et procède à l'exécution de *r*. Elle initialise *X*, par exemple comme un pointeur vers une cellule libre de la pile globale. La cellule choisie dépend du sommet actuel de la pile globale de la $TWAM_1$.

La $TAWM_2$ échoue avant la liaison de son "*X*" et demande du travail à la $TWAM_1$. Si ce travail est postérieur au point de choix de *q*, la $TWAM_2$ recopiera les portions des piles locale et

globale de la $TWAM_1$ postérieures à *q*. Mais, elle conservera son initialisation de *X*, au lieu de récupérer la valeur de la $TWAM_1$ (quelle qu'elle soit).

Inversement, la $TWAM_2$ pourrait échouer avant l'initialisation de *X*, et récupérer un point de choix postérieur à l'appel de *r* par la $TWAM_1$. Là aussi, la $TWAM_2$ conserverait éventuellement la cellule *x* de l'environnement à l'état libre, bien que la $TWAM_1$ ait successivement:

- initialisé la cellule *x* à un pointeur vers une cellule de la pile globale;
- lié cette cellule de la pile globale à une valeur (avec traînage de cette liaison).

En effet, la mise à jour des liaisons par la $TWAM_2$ ne restaurera que la liaison, mais pas l'initialisation.

c) Maintien de la cohérence des copies:

Pour supprimer ce problème, il faudrait traîner les initialisations en plus des liaisons. Une meilleure solution est de forcer une initialisation des variables, avant le premier appel à un prédicat pouvant engendrer des points de choix.

Les modifications à apporter à une compilation "classique" sont mineures:

- initialiser toutes les variables apparaissant, pour la première fois, après le premier sous-but non-déterministe, par une instruction nouvelle *init_perm*(Y_i), dont la fonction est l'initialisation de la variable;
- pour ces variables, remplacer toutes les instructions du type *"instruction"_variable* par les instruction équivalentes *"instruction"_value*.

Par exemple, la clause

p(X) :- q(Y), r(Z), s(Z, V), ...

est compilée:

"code_tête",		
init_perm(Z),	init_perm(V),	
put_variable(Y,X_1),	call(q/1),	
put_value(Z,X_1),	call(r/1),	
put_value(Z,X_1),	put_value(V,X_2),	call(s/2), ...

Remarque:

La détermination du caractère déterministe d'un prédicat est un problème complexe. Ainsi, on considère non-déterministe le premier sous-but apparaissant après les prédicats prédéfinis, connus pour être déterministes (*integer/1*, *var/1*, ...).

4.3.2.2. Gestion de la Mémoire et la Pile de Variables

La pile de variables est un simple tableau regroupant toutes les variables ([WAR 87b], [WAR 84]). Toute variable Prolog se représente par un emplacement dans la pile locale (dans un environnement) ou dans la pile globale (dans un terme). Cet emplacement désignera toujours une cellule du tableau.

a) Gestion de la mémoire:

L'allocation dans cette pile de variables se fera à l'initialisation de la variable ou à la construction d'un terme. La libération ne pourra s'effectuer qu'au retour-arrière. Cette pile de variables est donc gérée comme la pile globale.

On perd ainsi une partie des optimisations de réglage d'environnement et d'appel terminal, puisqu'on ne pourra récupérer que la place allouée dans l'environnement, à l'exclusion de celle allouée dans la pile de variables.

b) Pile de variables locales et pile de variables globales

La solution à ce problème est relativement simple. Il suffit d'introduire deux piles variables et de calquer leur gestion sur celle des piles locale et globale (cf. [WAR 87b], [LUS 88]).

La pile de variables globales sera utilisée pour l'allocation de toutes les variables logiques apparaissant dans des termes, ou qu'il est nécessaire de préserver, du fait des optimisations de réglage d'environnement et d'appel terminal portant sur la pile de variables locales.

La pile de variables locales servira, comme la pile locale, pour l'allocation des variables locales à la clause.

Le problème d'atomicité de l'allocation/initialisation se pose doublement dans ce cas, car toute allocation se fera dans la pile globale/locale et dans la pile de variables globales/locales. La solution proposée précédemment (cf. la section 3.2.1) résout le problème.

4.3.2.3. Liaison Superficielle/Liaison Profonde

Le mécanisme de pile de variables introduit une indirection supplémentaire dans le déréférencement (cf. le chapitre 3, section 3.5.1), car il est équivalent à une liaison superficielle.

Cette technique de pile de variables a pour objectif de permettre une déliaison non-locale rapide. Elle n'intéresse donc que les liaisons conditionnelles. En conséquence, une optimisation est de réserver l'utilisation des piles de variables pour les liaisons conditionnelles, et d'utiliser la liaison profonde pour les liaisons inconditionnelles.

La mise en place d'une telle distinction peut se réaliser en reconnaissant une partie des liaisons conditionnelles/inconditionnelles à la compilation, et l'autre à l'exécution.

4.3.2.4. Variable Conditionnelle/Inconditionnelle

Déterminer à la compilation, les liaisons inconditionnelles et conditionnelles revient à calculer d'une part les modes des variables ([OUD 87]), d'autre part le caractère déterministe ou non d'un prédicat. Ces problèmes sont encore du domaine de la recherche.

Il est par contre possible de classer les variables en **inconditionnelles** (faisant l'objet d'une liaison inconditionnelle manifeste dans le programme source) et **conditionnelles** (les autres).

Les variables conditionnelles sont donc celles qui peuvent donner lieu à une liaison conditionnelle. Elles sont potentiellement libres après l'unification de la tête de la clause, et avant l'appel du premier sous-but non-déterministe (créant un nœud OU) pouvant les lier.

Les variables conditionnelles sont locales ou globales (apparaissant dans un terme de la tête ou du corps de la clause). Selon le cas, elles sont représentées seulement dans la pile locale, ou représentées dans la pile globale et accédées depuis la pile locale. Notons que les variables

de la tête peuvent être déjà liées à leur première occurrence (passage de paramètres). Ce n'est pas le cas des variables dont la première occurrence est dans le corps.

Une génération d'instructions spécifiques pour les variables conditionnelles permettra de prendre en compte la différence entre liaison conditionnelle/inconditionnelle.

4.3.2.5. Gestion des Points de Choix et la Pile Locale

Dans la WAM, les opérations sur les environnements sont indépendantes des celles sur les points de choix. Cependant, la WAM utilise une seule zone de mémoire (pile locale) pour empiler ces deux structures de données. Ceci ne simplifie que la protection des environnements par les points de choix: l'empilage d'un nouveau point de choix protège automatiquement les environnements déjà empilés.

Dans Opera, l'installation de tâches transfère des branches de l'arbre OU, de la TWAM exportatrice vers la TWAM importatrice. Ces branches sont enregistrées dans les points de choix et dans le code. Le transfert de tâches implique donc des modifications dans les points de choix de la TWAM exportatrice, du fait qu'il faut éviter la duplication du même calcul par les deux TWAM (c'est inutile). A l'extrême, ces modifications se traduisent par l'enlèvement des points de choix dont toutes les branche ont été exportées. La récupération de l'espace occupé par ces points de choix, dans la TWAM exportatrice, n'est pas triviale, à cause des environnements qui s'intercalent avec les premiers.

L'installation de tâches transfère aussi des environnements nécessaires à l'exécution de la tâche exportée par la TWAM importatrice. Dans le cas de la pile (physique) unique pour les environnements et les points de choix, les environnements à transférer peuvent être intercalés avec des points de choix non-exportés. Par conséquent, le transfert d'environnements doit être réalisé:

- soit en étapes: une pour chaque zone contiguë d'environnements. Cependant, pour la même longueur de données, cette solution est moins efficace qu'un seul transfert, parce que le coût d'amorçage est multiplié par le nombre de zones;
- soit en un seul transfert, mais incluant les points de choix non-exportés. Ici, on transfère des données non-nécessaires à la TWAM importatrice; le pire, celle-ci doit éviter l'exploitation de branches associées à ces points de choix (calcul inutile), ce qui exige des procédés additionnels au travail d'installation de tâches, comme par exemple marquer ces points de choix.

La solution naturelle à tous ces problèmes, liés à la gestion des points de choix, est la partition de la pile locale en deux piles (physiques):

- la pile locale, qui ne contient que les environnements;
- la pile de points de choix.

Cette modification ne dégrade en rien l'exécution séquentielle et a même un effet positif vis à vis d'opérations telles que l'optimisation d'appel terminal. Elle rend indépendante l'exécution séquentielle de la gestion du OU parallélisme, qui ne s'intéresse qu'à la pile choix.

4.3.2.6. Bilan

La mise en place d'une méthode à copie incrémentale, utilisant une zone réservée aux variables pour une mise à jour rapide des liaisons, nécessite de produire un code plus spécifique du point de vue des instructions d'initialisation et d'unification.

Cette contrainte est indépendante de la façon dont sont implantées ces instructions (cf. les sections 4.3.3 et 4.3.4).

Ces instructions apportent un sur-coût sur l'exécution séquentielle. Celui-ci se traduit par une installation plus lourde des variables (initialisation) et par un accès moins efficace (indirection supplémentaire de la liaison superficielle).

4.3.3. La Méthode Pile de Variables Plus Datation (VD)

Cette méthode effectue une déliaison non-locale parallèle des piles de variables pendant le transfert des autres piles. Cette déliaison "à la volée" n'est parallèle que si l'on peut savoir à la consultation d'une variable si la liaison est valide ou non. L'association d'une date de liaison à une variable est suffisante ([SOH 85]). La déliaison doit être réalisée, si la date de liaison est postérieure à la date de liaison du plus récent des points de choix transférés (PPR).

La méthode de datation des liaisons accroit les sur-coûts:

- sur-coût temporel: les mécanismes d'allocation et d'initialisation des variables deviennent plus complexes, à cause de la pile de variables et de la datation;
- sur-coût temporel: les mécanismes de liaison et de déliaison sont plus lourds pour les mêmes raisons;
- sur-coût temporel: le mécanisme de déréférencement des variables libres et des liaisons conditionnelles ont une indirection de plus;
- sur-coût mémoire: deux nouvelles cellules par variable (valeur et date) ont été ajoutées.

Cependant, il est possible de réduire ces sur-coûts en:

- conservant le mécanisme de liaison profonde pour les liaisons inconditionnelles;
- préservant les mécanismes de récupération de mémoire des variables locales (optimisations de réglage d'environnement et d'appel terminal) même liées conditionnellement.

Notons que le transfert de la pile traînée est devenu inutile pour la déliaison non-locale.

4.3.3.1. Organisation de la Mémoire

Il est important de remarquer que les variables continuent à être allouées dans les piles locale et globale, la pile de variables n'étant qu'une structure supplémentaire utilisée pour les liaisons conditionnelles. Donc, pour préserver les mécanismes de récupération des variables locales (optimisations de réglage d'environnement et d'appel terminal), deux piles de variables sont nécessaires ([LUS 88]):

- pile de variables locales: pour les variables conditionnelles locales;
- pile de variables globales: pour les variables conditionnelles globales.

Les deux nouveaux registres de piles sont:

- VL: sommet de la pile de variables locales;
- VG: sommet de la pile de variables globales.

Les deux registres sont sauvegardés dans les points de choix (*B.VL* et *B.VG*). Une nouvelle cellule (*E.VL*) est ajoutée à chaque environnement: elle contient un pointeur vers la section de la pile de variables locales, allouée pour les variables de l'environnement.

Chaque liaison occupe deux cellules, nommées **structure variable** dans la suite:

- valeur: la valeur est une référence au terme auquel la variable a été liée;
- date: la date au moment de la liaison.

4.3.3.2. Allocation et Initialisation de Variables

L'allocation, l'initialisation et la libération des variables inconditionnelles sont rigoureusement identiques à celles de la WAM séquentielle.

Pour les variables conditionnelles, l'allocation et la libération de cellules dans les piles locale ou globale sont effectuées de façon identique à la WAM séquentielle.

Par contre, cette allocation s'accompagne de l'initialisation avec la référence à une structure variable allouée dans la pile de variables locales ou globales. Cette structure doit à son tour être initialisée à une valeur "libre".

Dans le cas d'une variable conditionnelle globale, l'allocation et l'initialisation ont lieu en même temps (cf. la section 4.3.2.1); pour des variables conditionnelles locales, l'allocation et l'initialisation doivent être regroupées pour éviter le problème d'incohérence due à la copie incrémentale (cf. la section 4.3.2.1).

a) Variables conditionnelles locales:

L'allocation et l'initialisation d'une variable locale conditionnelle doit se réaliser de façon identique dans la pile locale et dans la pile de variables locales. Il est donc nécessaire de gérer un environnement local et un environnement dans la pile de variables locales.

L'instruction *call* se voit rajouter un argument constant supplémentaire (*VLs*), qui indique le nombre de structures variables à préserver. L'instruction *allocate* alloue l'environnement (pile locale) et les structures variables locales. Le sommet *VL* de la pile de variables locales est calculé de la façon suivante (similaire au calcul du sommet de la pile locale):

si B.VL > E.VL + CP.VLs
  alors VL ← B.VL
  sinon VL ← E.VL + CP.VLs

L'initialisation se réalise par une version spécifique de l'instruction *init_perm* (cf. la section 4.3.2.1), où un nouvel argument V_j (*init_perm*(Y_i V_j)) représente la position de la structure variable allouée pour la variable Y_i. Cette instruction initialise Y_i à la structure variable, dont l'adresse est calculée par $E.VL + V_j$.

L'instruction *put_value* est utilisée pour toutes les occurrences locales dans le corps, à l'exception des celles réservées à *put_unsafe_value*. Cette instruction est modifiée pour traiter le cas où il faut créer une variable globale: une structure variable est alors allouée dans la pile de variables globales.

Par exemple, la clause

p(Y) :- q(X), r(X), s(Y).

est compilée en (Y_2 est alloué à X)

"code_tête" , init_perm(Y_2,V_1), call(q), put_unsafe_value(Y_2,X_1), ...

Ces mécanismes d'allocation et d'initialisation sont très similaires à ceux d'Aurora ([LUS 88]). Par contre, dans Aurora l'initialisation est effectuée à la première occurrence, du fait que la section commune est physiquement partagée. Ceci est un avantage, si un échec se produit avant la première occurrence. Dans Opera, il serait possible d'incrémenter *VL* à chaque initialisation, et de le décrémenter à chaque sous-but (*call*), du fait que l'initialisation est réalisée avant le premier sous-but. Ceci éviterait la sauvegarde de *VL* dans l'environnement, mais la

mise à jour constante de *VL* semble coûter plus cher que son calcul (et sauvegarde), à chaque allocation générale.

b) Variables conditionnelles globales:

Dans la WAM, l'allocation de variables globales est effectuée au moment de la première occurrence. Il est donc certain que ces variables ont la même initialisation dans les TWAM qui les partagent. Néanmoins, des modifications ont été introduites dans la compilation de variables globales de la tête, dans le but d'éviter l'allocation de structures variables pour celles qui sont déjà liées avant le premier sous-but.

La dernière occurrence avant le deuxième appel utilise une variante, nommée *cond*, des instructions *unify_variable*, *unify_value*, *unify_local_value* et *put_value*. Ces variantes allouent une structure variable et lient la variable globale à cette structure, si, en fonction de l'instruction:

- unify_variable_cond: le mode est écriture;
- unify_local_value_cond: le mode est écriture et la variable est globalisée;
- unify_value_cond: le mode est écriture et la variable (pas l'élément qui sera lié à la variable) est encore libre;
- put_value_cond: la variable est encore libre.

La première occurrence d'une variable globale dans le corps est toujours codée par *unify_variable_cond* et *unify_local_value_cond*, les variantes normales étant réservées aux variables de la tête ayant une deuxième occurrence avant le deuxième sous-but. En exécution séquentielle, le seul sur-coût de cette optimisation est le test de variable libre dans les *unify_value_cond* et *put_value_cond*.

Par exemple, les variables de la clause suivante

$$p(S^1, [Y, Z^1, T^1],) :- q(T^2, [V^1, S^2, Z^2]), r([S^3]), \ldots$$

sont affectées par:

- Y, V^1: unify_variable_cond;
- Z^1, T^1: unify_variable;
- Z^2: unify_value_cond;
- T^2: put_value_cond;
- S^2: unify_local_value_cond;
- S^3: unify_value.

Le schéma du code TWAM est:

```
...
get_variable(S,X1);          get_structure(X2);
unify_variable_cond(Y);      unify_variable(Z);      unify_variable(T);

put_value_cond(T,X1);        put_structure(X2);
unify_variable_cond(V);      unify_local_value_cond(S);
unify_value_cond(Z);         call(q/2);

put_structure(X1);           unify_value(S);
call(r/1);
...
```

c) L'initialisation des piles de variables:

La structure variable, locale ou globale, est initialisée de la façon suivante, dans tous les cas:

- valeur: libre;
- date: libre.

Cette initialisation peut s'effectuer de façon statique (une seule fois) à condition que la coupure re-initialise les sections coupées. Cette option permet de traiter tous les occurrences locales comme *put_value*. Une autre option est l'initialisation lors de la première occurrence (*variable*), locale ou globale, en évitant le sur-coût au moment de la coupure.

4.3.3.3. Liaison et Déréférencement

Les liaisons sur les variables inconditionnelles procèdent de façon standard (liaison profonde). Par contre, une variable conditionnelle peut être liée conditionnellement ou inconditionnellement.

Les liaisons inconditionnelles sont implantées par des références directes au terme, sans passer par la structure variable éventuellement allouée à la variable. Elles se produisent dans quatre situations:

- SL1: à l'initialisation dans la tête: par un *get_variable*;
- SL2: à l'unification des variables globales de la tête: par exemple, par un *unify_value* en mode lecture;
- SL3: à l'exécution des sous-buts s'il n'existe pas de nœuds OU entre l'allocation de la variable et sa liaison: par exemple, par un *get_integer* dans la tête de la clause appelée par le sous-but;
- SL4: à la continuation de la clause, mais seulement pour les variables globales, sous la condition précédente (existence de nœuds OU) (fig. 4.2-a).

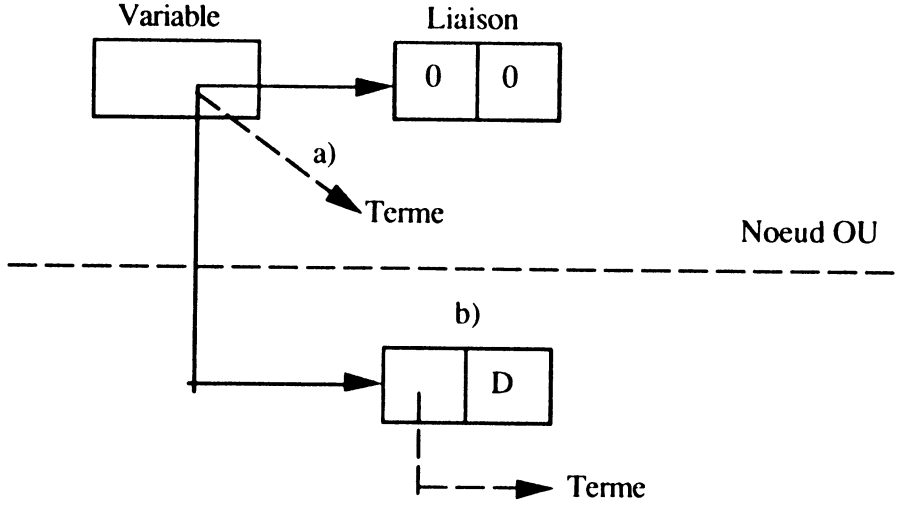

Fig. 4.2 - Liaison avec pile de variables et datation

Les situations SL3 et SL4 affectent les variables conditionnelles, ce qui oblige à remplacer la référence à la structure variable par la référence au terme. L'espace physique alloué à cette structure n'est pas récupérable à ce moment, sauf s'il correspond au sommet de la pile (ceci exige un test dynamique). L'espace physique sera normalement récupéré par le retour-arrière.

Les liaisons conditionnelles ne se produisent que dans les situations SL3 et SL4, mais ici à la condition inverse: il existe des nœuds OU entre l'allocation et la liaison de la variable. Dans

ce cas la référence à la structure variable est conservée, celle-ci reçoit la référence au terme et la date courante, et une référence à la structure variable est ajoutée à la pile traînée (fig. 4.2-b).

Le déréférencement est à-peu-près le même pour les deux types de liaison: il avance jusqu'à trouver un terme non-variable ou une référence à libre, celle-là pouvant apparaître dans une cellule globale ou dans une structure variable. En raison des liaisons inconditionnelles des variables conditionnelles, l'adresse précédente doit être sauvegardée de façon à permettre la liaison directe au terme. Ceci revient à dire que le déréférencement doit retourner l'adresse de la cellule locale ou globale, si la variable est libre. Pour la même raison, la liaison à une autre variable pointe vers la cellule locale ou globale.

La direction de la liaison entre deux variables peut être déterminée par une comparaison d'adresses (piles locale et globale), si les zones de piles sont allouées de façon statique à la même position dans chaque TWAM.

4.3.3.4. Déliaison Séquentielle

La déliaison séquentielle de la TWAM, effectuée lors d'un retour-arrière, remet la cellule valeur des piles de variables à libre. Il n'est pas nécessaire de remettre la date, celle-là n'étant pas utilisée en séquentiel.

4.3.3.5. Récupération de l'Espace dans les Piles de Variables

L'espace dans les piles de variables est récupéré de la même façon que dans les piles locale et globale. A la fin d'une clause, en retournant à l'environnement de la clause appelante, le pointeur de la section de la pile de variables locales (*E.VL*) est automatiquement récupéré. Comme pour la pile locale, cette récupération est effective si les structures ne sont pas protégées par un point de choix (*B.VL*). Les optimisations de réglage d'environnement et d'appel terminal sont réalisées par:

- l'allocation des structures (calcul de V_j à la compilation) est effectuée dans l'ordre inverse de la dernière occurrence, comme pour les cellules de la pile locale;
- l'instruction *put_unsafe_local* a été adaptée: elle alloue une structure dans la pile de variables globales si la variable est locale et libre.

Les structures de la pile de variables globales sont récupérées lors d'un retour-arrière par la mise à jour de *VG* par *B.VG*.

4.3.3.6. Datation

La date courante est donnée par la profondeur dans l'arbre OU. Elle est maintenue dans un nouveau registre Clock. Initialement, elle prend la valeur la plus basse, par exemple 0. A chaque création d'un nœud OU, elle est augmentée de 1, et à chaque retour au nœud précédent, c.a.d. après l'épuisement de toutes les alternatives du nœud courant, elle est diminuée de 1.

Cependant, la WAM détruit le nœud OU (par l'instruction *trust*, cf. l'annexe B) au début de l'exécution de la dernière branche, ce qui empêche la distinction de deux cas: retour-arrière à un nœud qui a eu, ou qui n'a pas eu, des nœuds descendants. La solution adoptée dans la TWAM est la sauvegarde de Clock dans le point de choix, après l'augmentation. Cette valeur est restaurée dans tous les retours-arrière.

4.3.3.7. Installation de Tâches

La description ci-dessous correspond à l'exportation d'un seul point de choix (le PPR), celui qui est au fond la pile. Elle ne considère pas la copie incrémentale de pile.

L'installation est constituée des étapes suivantes (voir schéma dans la fig. 4.3):

- d'abord le PPR est transféré, ce qui permet à la TWAM importatrice de connaître la date du nœud, et les positions et les tailles des piles à recevoir;
- ensuite les autres piles sont transférées dans l'ordre:

  - piles de variables;
  - pile globale;
  - pile locale.

- la TWAM importatrice, en parallèle à la réception de ces trois piles, effectue trois actions:
  - le chaînage du PPR dans la pile de points de choix doit être actualisé à cause d'une translation éventuelle de la position physique du point de choix. Le but de la translation est de récupérer l'espace des points de choix précédemment exportés par la TWAM exportatrice;
  - l'état de la TWAM, correspondant au moment initial d'un retour-arrière au PPR importé, est simulé;
  - la déliaison non-locale: les piles de variables sont parcourues, et la structure est initialisée à libre, si la date de la liaison est égale ou supérieure à la date du nœud. Evidemment, le début de cette action est synchronisé avec la fin du transfert des piles de variables.

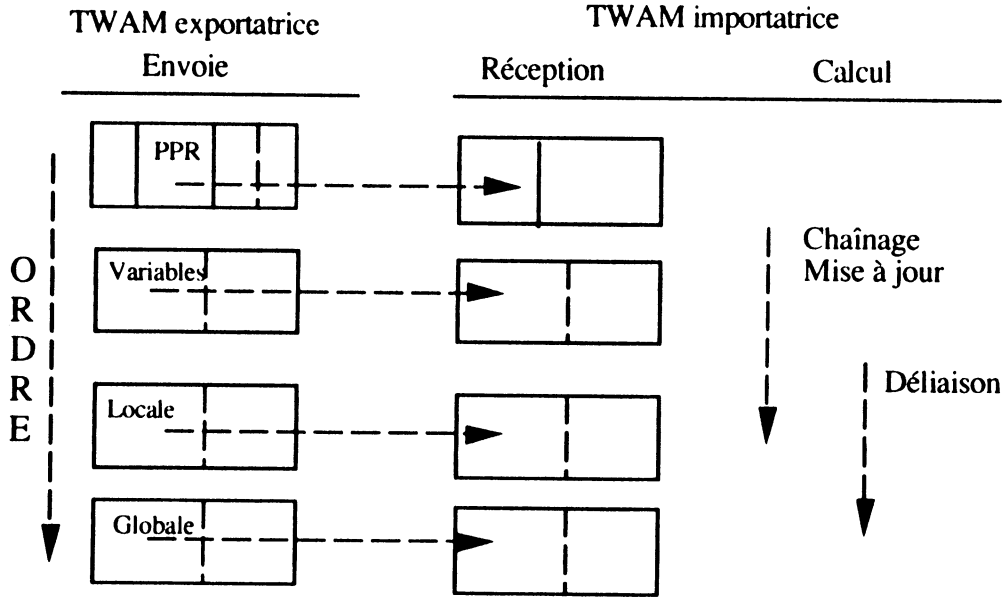

Fig. 4.3 - Schéma de l'installation VD

L'état de la TWAM importatrice sera complètement mis à jour par l'instruction de retour-arrière, liée à la première alternative suspendue de PPR.

Cette méthode doit être étendue si deux optimisations sont introduites:

- O1: plusieurs (plus que 1) points de choix sont exportés: la TWAM importatrice a besoin de la pile traînée, limitée au PPR, pour pouvoir effectuer les déliaisons locales au moment des retours-arrière aux points de choix plus vieux que PPR;
- O2: la copie incrémentale est implantée: les liaisons conditionnelles des variables communes, réalisées, par la TWAM exportatrice avant la création de PPR, doivent être transférées.

La façon la plus simple d'installer les liaisons conditionnelles de la section commune (O2) est le transfert total des piles de variables, c.a.d. la section entre la base et PPR. Ici, l'inefficacité provient du transfert des liaisons déjà existantes dans la TWAM importatrice, celles effectuées avant le point de choix commun le plus récent. Ce sur-coût augmente toujours avec l'augmentation de la partie commune, c.a.d. vers la fin de l'exécution, où les TWAM sont proches des feuilles et les tâches exportables sont plus réduites.

Une autre façon d'implanter l'optimisation O2 est la détection par l'exportateur de chaque liaison conditionnelle des variables communes:

- soit en parcourant la section commune des piles de variables: les liaisons à transférer sont celles dont la date de liaison est égale ou supérieure à la date du point de choix commun le plus récent et inférieure à la date de PPR;
- soit en parcourant la pile traînée: les liaisons à transférer sont celles dont la structure variable se situe dans la section commune.

L'envoi de ces liaisons peut se réaliser de deux façons:

- une par une: on transfère la liaison avec son adresse dans les piles de variables. Le désavantage est que le coût d'amorçage du transfert est multiplié par le nombre de liaisons;
- en paquet: toutes ces liaisons, et leurs adresses dans les piles de variables, sont d'abord copiées dans une zone tampon de l'exportateur, avant d'être transférées. L'utilisation de 2 tampons en alternance permet d'assurer un transfert pendant un remplissage diminuant ainsi le temps d'attente de la TWAM exportatrice.

Dans les deux cas, par liaison ou en paquet, l'installation dans les piles de variables de l'importatrice ne peut se réaliser qu'après la réception de ces informations.

Ces considérations montrent que, l'introduction des optimisations O1 et O2 dans la méthode VD induit un sur-coût non-négligeable. Cependant, les premières analyses des mesures de performance du prototype Opera indiquent la nécessité d'exporter plusieurs points de choix (plus de 1). Cette nécessité provient du coût trop élevé de l'installation d'une tâche, et principalement, de la recherche d'une tâche, dans une machine sans mémoire commune. L'implantation de la copie incrémentale semble être aussi très importante.

4.3.4. La Méthode Pile de Variables et Traînée plus Valeur (VV)

Cette section décrit la deuxième méthode pour l'installation de tâches (VV). Elle utilise aussi les piles de variables; la datation a été supprimée et une deuxième cellule a été ajoutée à la pile traînée ([WAR 84], [WAR 87b]). L'idée de base de cette méthode est le transfert exclusif des liaisons valides à l'importateur, cette validité étant déterminée par le nœud PPR. Les avantages sont:

- pile traînée plus valeur: le transfert de liaisons conditionnelles valides des variables communes aux deux TWAM devient plus efficace;
- piles de variables: la déliaison automatique des variables non-communes évite le transfert de la pile traînée entière.

Les sur-coûts introduits dans la TWAM sont à-peu-près équivalents à ceux introduits par la méthode précédente, affectant principalement les mécanismes de traitement des variables conditionnelles. On rappelle que les sur-coûts de la méthode VD sont plutôt dus aux piles de variables (allocation et initialisation plus lourdes, et indirection au déréférencement) qu'à la datation.

4.3.4.1. Organisation de la Mémoire

Les piles de variables ne contiennent que la cellule valeur. Le registre Clock n'est plus nécessaire. La pile traînée contient deux cellules par liaison conditionnelle: adresse et valeur.

4.3.4.2. Traitement des Variables

Les mécanismes d'allocation, d'initialisation, de récupération et de déliaison des variables conditionnelles ne sont pas modifiés, par rapport à la méthode précédente. Ils sont réalisés dans les piles de variables, mais sans datation. La différence essentielle est la liaison conditionnelle: la valeur de la liaison est sauvegardée à coté de l'adresse de la variable, dans la pile traînée (fig. 4.4).

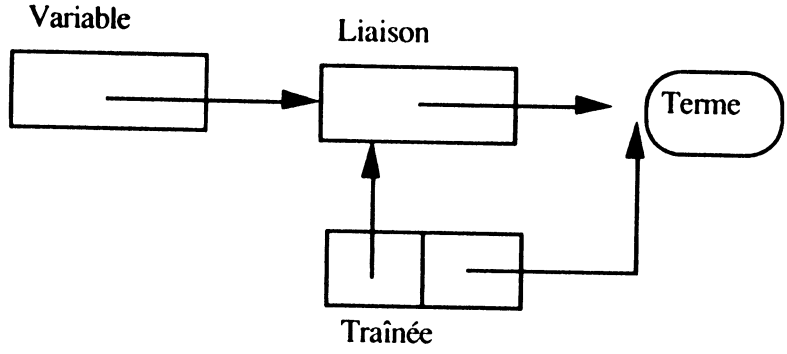

Fig. 4.4 - Liaison conditionnelle dans la méthode VV

4.3.4.3. Installation de Tâches

Les principes de cette méthode sont:

- les tableaux de variables ne sont pas transférés;
- l'importatrice effectue la déliaison locale sur la section commune et met entièrement à libre la section non-commune;
- les liaisons conditionnelles valides sont transférées par la pile traînée et installées sur les sections commune et non-commune par la TWAM importatrice.

On remarque que la pile traînée ne contient, au-dessous du nœud PPR, que des liaisons valides.

Le procédé décrit ci-dessous permet:

- l'exportation de plusieurs points de choix;
- la copie incrémentale.

On observe qu'une TWAM peut avoir plusieurs points de choix communs, à cause de plusieurs exportations/importations avec plusieurs TWAM. PCC est le plus vieux des points de choix communs aux deux TWAM. Cependant, il peut ne pas être le plus vieux de tous les points de choix communs d'une TWAM (voir fig.4.5). La méthode utilisée pour la détermination de PCC sera décrite plus loin.

Installation:

- initialement, deux étapes sont exécutées en parallèle (voir schéma dans la fig. 4.6):

- les points de choix sont transférés; le PPR et le PCC sont utilisés par les deux TWAM pour déterminer la position et la longueur des sections des piles à transférer;
- la TWAM importatrice effectue la déliaison locale des liaisons parvenues après le PCC, en utilisant sa pile traînée;

- ensuite, la section non-commune de la pile traînée est transférée;

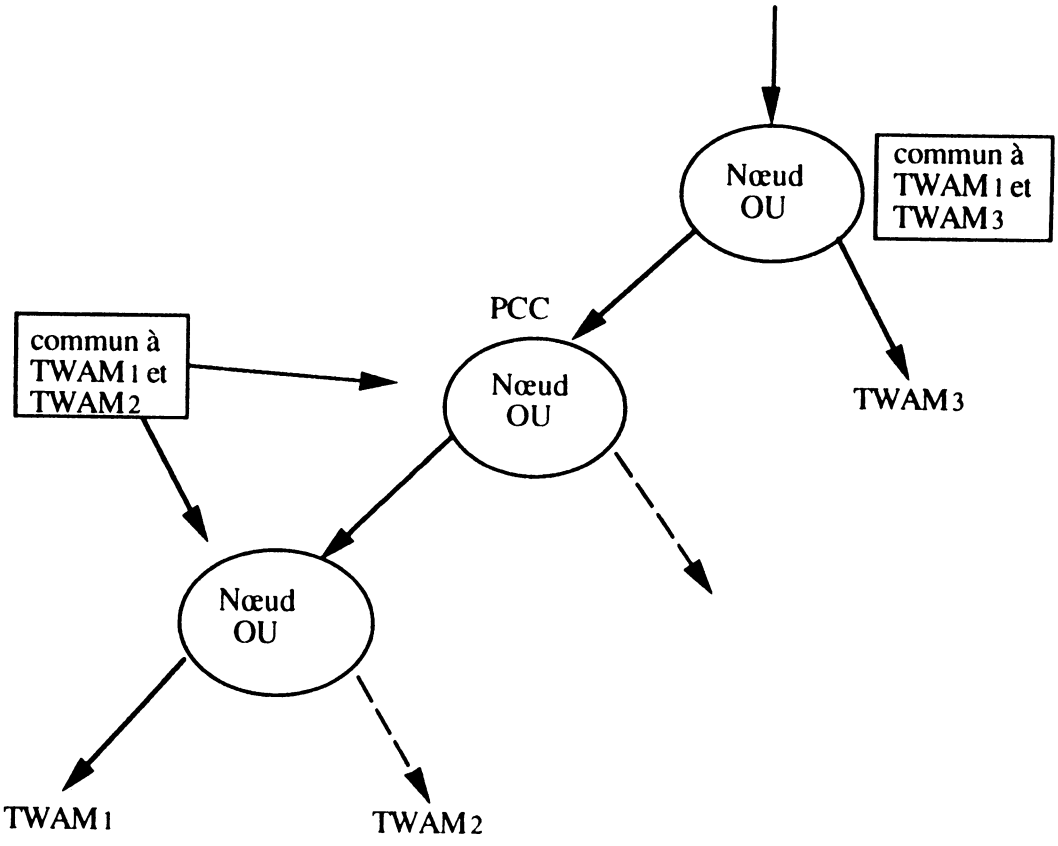

Fig. 4.5 - Points de choix communs

- à la fin du transfert de la pile traînée, deux étapes sont exécutées en parallèle:

  - les sections non-communes des piles globale et locale sont transférées;
  - la TWAM importatrice installe dans les piles de variables les liaisons existantes dans la section non-commune de la pile traînée. On observe que cette étape peut se réaliser en parallèle à la précédente (DMA).

Cette méthode présente un synchronisme entre réception et travail d'installation effectué par la TWAM importatrice: la réception de la pile traînée ne peut commencer qu'après la fin de la déliaison locale. Le temps d'attente correspond à la différence entre les durées de la déliaison locale et du transfert des points de choix. Cependant rappelons que la TWAM importatrice a effectué automatiquement, au moment où elle est devenue inactive, la déliaison relative à son point de choix commun le plus récent, donc la durée de la déliaison locale sera nulle, si ce point est le PCC, c.a.d. le point en commun avec la TWAM exportatrice.

Une sous-étape d'initialisation à libre de la section non-commune des piles de variables est nécessaire, si l'initialisation n'est pas statique (voir la section 4.3.3.2, paragraphes sur

l'initialisation des structures variables). Cette étape doit se réaliser avant l'installation des liaisons de la section non-commune de la pile traînée.

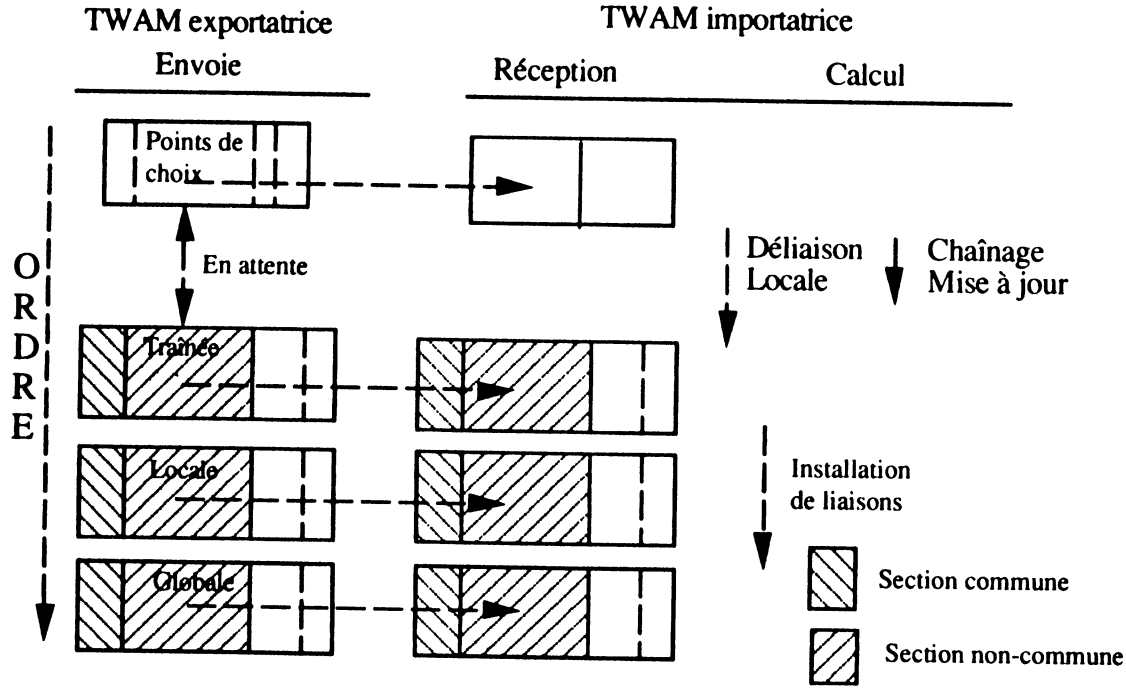

Fig. 4.6 - Schéma de la méthode VV

4.3.5. Bilan Datation ou Traînée plus Valeur (VD/VV)

Cette section présente une comparaison préliminaire entre les deux méthodes. D'abord, le coût d'installation sera examiné, en supposant que, dans la méthode datation, il est préférable de parcourir la section commune des piles de variables pour détecter les liaisons à transférer, et en omettant la déliaison locale, nécessaire aux deux méthodes:

"Datation":

- transfert: section non-commune des piles locale, globale, de variables et traînée; liaisons sur la section commune, en sous-ensembles contigus;
- calcul: parcours des deux sections des piles de variables.

"Traînée plus valeur":

- transfert: sections non-communes des piles locale, globale et traînée, celle-là ayant deux cellules par liaison;
- calcul: parcours de la pile traînée entre le PCC et le PPR; parcours de la section non-commune des piles de variables.

Différence: ("datation" moins "traînée plus valeur"):

- transfert: 2 (2 cellules) fois le nombre de variables conditionnelles de la section non-commune moins le nombre de liaisons conditionnelles de la section non-commune;
- calcul: le nombre de variables conditionnelles de la section commune moins le nombre de liaisons conditionnelles de la section non-commune.

La conclusion définitive dépend de la différence entre le nombre de variables conditionnelles appartenant à la section non-commune et le nombre de liaisons conditionnelles effectuées pendant le calcul de la section (branche) non-commune. Des mesures expérimentales

sur un nombre représentatif de programmes Prolog seront nécessaires. Néanmoins, le facteur 2 plus les liaisons sur la section commune, de la méthode VD, semble indiquer un avantage pour l'autre méthode. En outre, la méthode pile traînée plus valeur reporte tout le calcul d'installation sur le processeur (TWAM) importateur, ce qui est préférable à coûts égaux, du fait que la TWAM exportatrice peut continuer à exécuter sa branche courante.

Du point de vue de l'espace mémoire, la deuxième méthode est encore préférable, parce que sa deuxième cellule, la valeur, est allouée si une liaison se produit. Dans la méthode VD, la date est allouée presque de façon statique (variables conditionnelles), même si la liaison ne se réalise pas ou si la liaison devient inconditionnelle.

Par la suite, l'expression *pile de variables* sera utilisée pour référencer les deux piles de variables.

4.3.6. Contrôle des Points de Choix Communs

L'implantation de la copie incrémentale exige la connaissance des diverses copies de parties communes entre deux TWAM. Cette connaissance peut être divisée en deux problèmes:

- PSC1: l'enregistrement de liaisons effectuées, sur la section commune, après le point de choix lié à cette section commune, par une TWAM qui ne possède qu'une branche de ce point de choix;
- PSC2: le maintien des informations qui permettent la détermination du point de choix commun (PCC) entre deux TWAM (exportatrice et importatrice).

Les solutions Opera à ces deux problèmes reposent sur un principe de base: une TWAM ne détruit sa section commune avec une autre TWAM, qu'au moment où elle importe des tâches dérivées des points de choix plus vieux que le point de choix lié à la section commune (voir fig. 4.7).

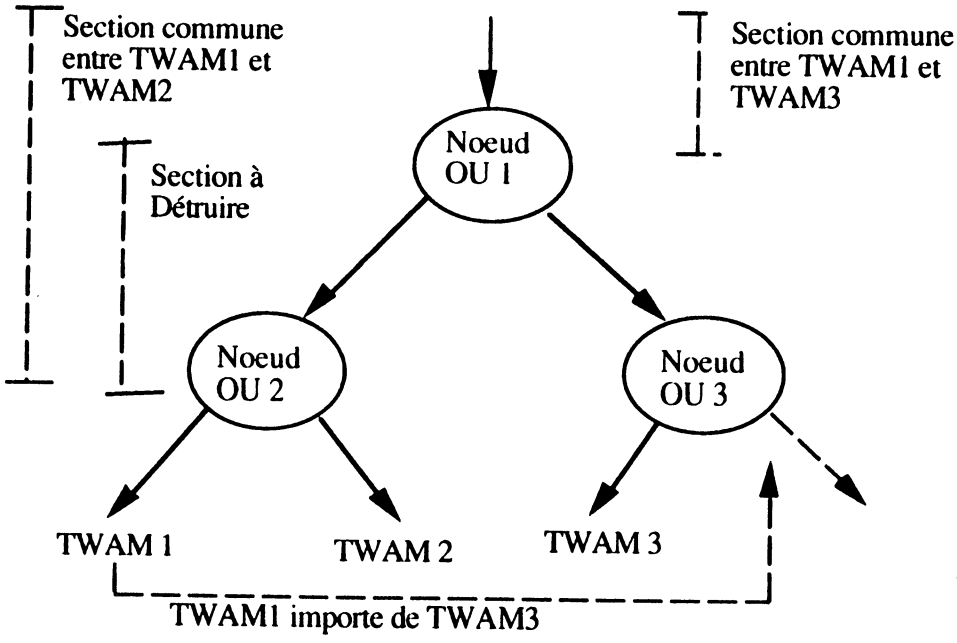

Fig. 4.7 - Destruction de section commune

Ceci exige deux règles d'ordonnancement:

- RO1: l'exportation d'une alternative est possible, si toutes les alternatives situées dans des points de choix plus vieux sont de même exportées;

- RO2: dans le cas d'une exportation, dont les alternatives appartiennent à plusieurs points de choix, la section commune est celle qui est définie par le point de choix le plus vieux, donc la section commune la plus petite (voir fig. 4.8).

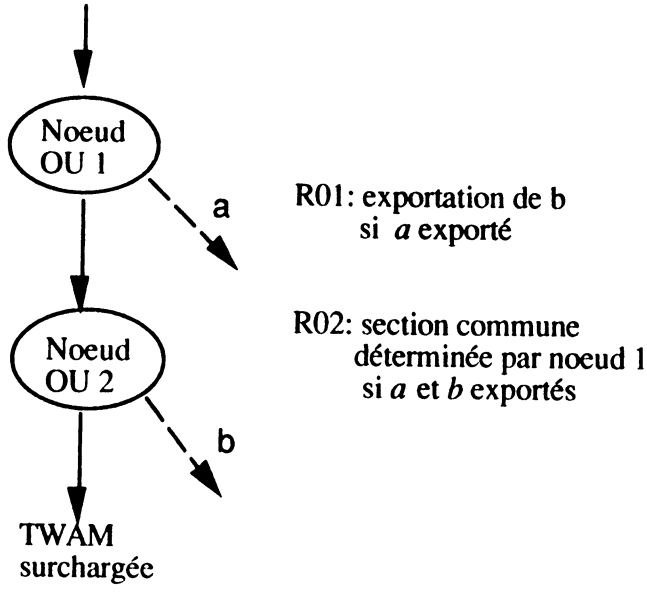

Fig. 4.8 - Règles d'ordonnancement

Le non-respect à ces règles implique des modifications dans la section commune, provoquées par les retours-arrières. Dans ce cas, le contrôle aurait besoin:

- soit d'une synchronisation entre les TWAM, ce qui ajoute un sur-coût élevé;
- soit d'une vérification et un ajustement éventuel de la section commune au moment du transfert. Ceci augmente la complexité du contrôle et du protocole de transfert, et peut annuler les gains obtenus par des transferts comportant des sections communes plus grandes.

La deuxième solution, c.a.d. augmentation des sections communes et contrôle plus complexe, pourrait être examinée dans une deuxième phase d'études, à partir de l'analyse de mesures, comme par exemple la taille et la fréquence de ces sections. Cette solution est voisine de la méthode Muse ([ALI 90]). Dans celle-ci les points de choix communs ont une seule copie dans une mémoire globale (physique) et les TWAM y accèdent normalement au retour-arrière. Nous reviendrons sur ces points par la suite (sections 4.3.6.1 et 4.3.6.2 sur les problèmes PSC1 et PSC2, respectivement).

4.3.6.1. Enregistrement des Liaisons de la Dernière Branche

Dans la WAM, les nœuds OU sont détruits au début de l'exécution de la dernière branche (instruction *trust*). Les liaisons, effectuées dans cette branche sur les variables créées entre le nœud OU précédent et le nœud détruit, et avant la création d'un autre nœud, deviennent inconditionnelles. Dans le cas séquentiel, ces liaisons ne sont pas traînées. En parallèle, elles ne seraient ni traînées ni enregistrées dans la pile de variables (liaison profonde). Cependant, si ces liaisons appartiennent à une section commune, elles doivent être connues par:

- la TWAM importatrice du fait que celle-ci doit effectuer une déliaison locale sur sa copie de la section commune;
- la TWAM exportatrice, parce que ses liaisons valides sont transférées vers la TWAM importatrice.

Il est donc nécessaire d'effectuer de façon superficielle ces liaisons, en les traînant et en les enregistrant dans la pile de variables.

Ceci se traduit aussi par la conservation d'un point de choix commun, jusqu'à ce que toutes ses branches soient épuisées (ceci est utile à d'autres fonctions comme l'ordonnancement des tâches). En présence d'une mémoire commune, cette conservation est facile.

Dans le cas d'Opera, les copies d'un point de choix étant dispersés entre différentes TWAM, la même solution exige une synchronisation très couteuse (algorithme de cohérence de copies multiples). Une autre solution est un contrôle individuel par TWAM, la destruction d'une copie ne devenant effective qu'au moment où la TWAM importe des branches plus proches de la racine. L'inconvénient est la sur-consommation de mémoire, due à la conservation de points de choix vides. Cette solution exige de plus soit un point de retour (alternative) spécial, soit un drapeau à tester dans tous les retours-arrière.

La solution actuelle d'Opera consiste à:

- ne détruire les points de choix communs que dans les cas suivants:
  - pour une TWAM exportatrice: lorsque toutes les alternatives sont exportées;
  - pour une TWAM importatrice ou exportatrice: lorsque un retour-arrière à la dernière alternative est effectué (action normale de la WAM).
- sauvegarder, au moment d'une installation de tâches, les sommets des piles locale et globale du point de choix commun le plus vieux, dans des registres spéciaux des deux TWAM (exportatrice et importatrice). Rappelons que la WAM utilise les sommets de ces piles pour déterminer si une liaison est conditionnelle ou inconditionnelle. Les valeurs sauvegardées sont utilisées à partir du moment où le point de choix le plus vieux est détruit (pile de choix vide).

Il est important d'observer que, dans le cas d'exportations successives par la même TWAM, la section commune d'une exportation est un sous-ensemble de la section commune de la prochaine exportation (par la règle RO1). La sauvegarde des sommets des piles locale et globale de la prochaine exportation garantit l'enregistrement des liaisons relatives à l'exportation précédente.

4.3.6.2. Maintien des Points de Choix Communs

Comme pour l'enregistrement des liaisons de la dernière branche, l'existence de mémoire commune permettrait d'enregistrer les points de choix communs, en étendant la pile de points de choix pour représenter l'arbre OU (pile cactus). Ce n'est pas nécessaire car il suffit de représenter les relations entre les points de choix communs des TWAMs. Cette représentation ne se modifie qu'à chaque installation de tâches, à condition de respecter les règles d'ordonnancement RO1 et RO2.

Une solution naïve consiste à représenter le point de choix commun à chaque couple de TWAM. La structure peut être une matrice M, de NxN éléments, où N est le nombre de TWAM. Chaque élément $E_{i,j}$, où $i \neq j$, contient le point de choix commun aux deux TWAM i et j et des données limitées à ce qui intéresse le contrôle: date, et tailles des piles locale, globale, traînée et de variables. La matrice est diagonale ($E_{i,j}$ est égal à $E_{j,i}$). $E_{i,i}$ porte l'état courant d'une TWAM.

Initialement, toutes les dates de la matrice sont égales à la date d'initialisation du registre Clock, et toutes les tailles de pile sont égales à zéro.

La matrice est mise à jour à chaque installation de tâches, de la façon suivante, en supposant que la TWAM *a* exporte à la TWAM *b*, et que le point de choix commun le plus vieux (parmi les exportés) soit daté par D:

- $E_{a,b} \leftarrow D$;
- $E_{b,k} \leftarrow E_{a,k}$, pour tout $k \neq a,b$.

La mise à jour de $E_{b,k}$ par $E_{a,k}$ signifie que, la TWAM importatrice hérite des points de choix communs de la TWAM exportatrice avec les autres TWAM.

L'implantation distribuée de la matrice impose, en général, un coût de synchronisation trop élevé, du fait que, pour le transfert entre *a* et *b*, il faut mettre à jour tous les *k*. Une simple implantation centralisée sera décrite dans la suite (section 4.6).

La représentation par la matrice a deux handicaps:

- elle ne s'adapte pas à des modifications dans les méthodes d'ordonnancement;
- les coûts, de mise à jour et de recherche de la plus grande section commune, sont en O(N);
- des informations sont dupliquées, c.a.d. plusieurs TWAM peuvent avoir le même point de choix en commun;
- en outre, la taille de la structure est en $N^2/2$ (matrice diagonale), et elle croît vite avec le nombre de processeurs.

Pour les grandes configurations du Supernode, il est possible d'utiliser une structure en forme d'arbre binaire, qui représente les points de choix communs, chaque TWAM étant attachée au point de choix qui est à l'origine de la dernière importation. La recherche du point de choix commun à deux TWAM données, et la mise à jour, sont plus complexes, mais leurs coûts sont, en moyenne, plus petits, et bornés à O(N). La taille de la structure est limitée au nombre de points de choix communs (N-1) plus le nombre de TWAM (N).

4.3.7. Cohérence des Données

L'accès concurrent aux données de la TWAM, dû à l'installation de tâches, pose un problème de cohérence des données.

Pour la TWAM importatrice, ce problème a déjà été examiné dans les sections précédentes. Dans le cas de la méthode pile traînée plus valeur, la déliaison locale doit attendre la fin du transfert de la pile traînée.

Du coté de la TWAM exportatrice, les problèmes de cohérence sont plus complexes. Dans un but d'efficacité, la TWAM continue son calcul pendant le transfert des piles. Ceci exige que les modifications effectuées par la TWAM ne perturbent pas la cohérence des données transférées, du point de vue de la TWAM importatrice.

4.3.7.1. Les Sources d'Incohérence

Le transfert d'une tâche implique:

- le retrait des points de choix exportés de la pile des points de choix de la TWAM exportatrice. Ceci entraîne la modification de registres d'état de la TWAM;
- le transfert de données d'état et de portions de piles. Ces données sont:
  - un nombre NPC de points de choix contigus, à partir du plus vieux: le plus jeune d'entre eux est le PPR. Il sont tous plus

jeunes que le point commun aux deux TWAM (PCC), et situés dans la même branche de l'arbre OU que le PCC;
- les sections non-communes des piles locale, globale et traînée, situées entre les point de choix PCC et PPR, pour les deux méthodes (VD et VV);
- la section non-commune de la pile de variables, dans le cas de la méthode VD;
- les liaisons conditionnelles valides de la section commune de la TWAM exportatrice, dans le cas de la méthode VD.

a) Cohérence des piles:

Si l'on admet que les transferts ont lieu en parallèle (DMA) du fonctionnement de la TWAM exportatrice, il faut vérifier que les opérations que réalise celle-ci sur ces données ne risquent pas de les altérer au cours de leur transfert.

Les opérations de la TWAM sur les piles peuvent être de quatre types:

- allocation: par exemple, l'allocation d'un environnement;
- création: par exemple, l'initialisation d'une variable, ou l'écriture d'une structure;
- modification: la liaison d'une variable libre et la déliaison correspondante (si liaison conditionnelle);
- destruction: la récupération d'un environnement à la fin d'une clause ou d'une cellule après la dernière occurrence de la variable respective, et la récupération des sections de toutes les piles lors d'un retour-arrière.

L'allocation des cellules transférées a été effectuée avant la création des points de choix transférés. Dans la WAM, les opérations de création sont effectuées au même moment que l'allocation, à l'exception de l'initialisation des variables locales. Dans la TWAM, les variables locales sont initialisées avant l'appel du premier sous-but (modifications introduites à cause de la pile de variables), donc avant la création d'un point de choix (cf. les sections 4.3.2.1 et 4.3.3.2). Il n'existe aucun risque d'incohérence.

Les cellules des piles locale et globale ne sont modifiées que par la liaison inconditionnelle de variables, donc avant la création de tout point de choix postérieur à la création de la variable. Par contre, les piles de variables sont justement modifiées par les liaisons conditionnelles et par les déliaisons respectives, donc peut-être après la création des points de choix transférés. Dans la méthode VD, il faut donc éviter le transfert d'une valeur (non-libre) couplée à une date libre de la section non-commune, et le transfert d'une liaison invalide de la section commune (voir fig.4.9).

La libération des environnements n'est effective que s'il n'existe aucun point de choix créé postérieurement. La pile traînée et la pile globale ne sont dépilées qu'au retour-arrière.

Ainsi, les seules possibilités de production d'incohérences sur les piles sont la liaison/déliaison conditionnelle (ceci affecte la méthode VD et les piles des variables) et le retour-arrière à un point de choix plus vieux que PPR (ceci affecte les deux méthodes et toutes les piles).

b) Cohérence du retrait de points de choix:

Le maintien de la cohérence de la pile de points de choix est plus délicat. D'abord, il faut éviter l'exécution de la même branche par les TWAM importatrice et exportatrice, parce que c'est un calcul inutile. Dans une machine à mémoire commune, où en général les points de choix communs sont physiquement partagés et une seule clause est exportée à chaque fois, il suffit d'accéder en exclusion mutuelle au point de choix lors des instructions de retour-arrière.

Dans Opera, les points de choix exportés, ou encore leurs clauses exportées, doivent être marqués ou enlevés de la pile de l'exportatrice. Ainsi, une exclusion mutuelle est nécessaire entre le retour-arrière effectué par la TWAM exportatrice et le marquage ou enlèvement des points de choix, nécessaire à l'exportation.

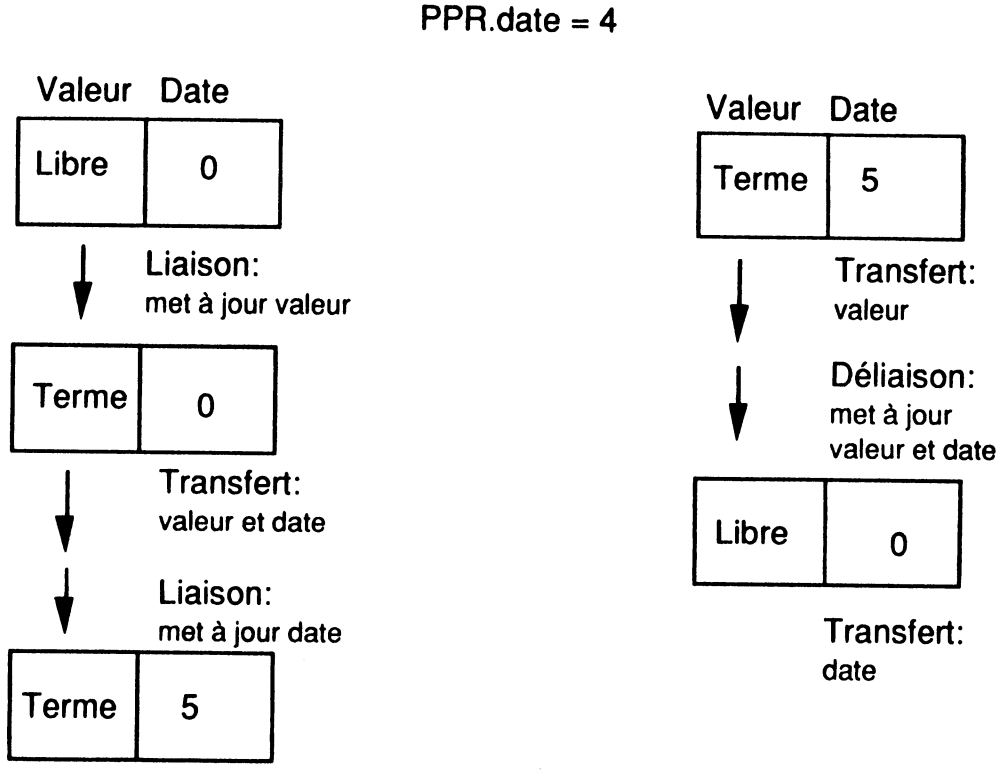

Fig. 4.9 - Transfert de liaison incohérente

4.3.7.2. Les Remèdes

a) Cohérence du retrait de points de choix:

Le principe de l'implantation actuelle est la manipulation de points de choix en bloc:

- toutes les alternatives d'un point de choix sont exportées (point de choix entier): ceci simplifie la mise à jour des points de choix[1];
- les points de choix exportés sont enlevés en bloc contigu de la pile;
- le point de choix courant de la TWAM exportatrice n'est jamais exporté;
- une TWAM effectue la mise à jour du sommet de la pile de points de choix (registre B) après la création et avant la destruction d'un point de choix. On évite que la TWAM importatrice accède à des données incohérentes.

Cette solution restreint l'exclusion mutuelle à:

- mise à jour du registre B par la TWAM exportatrice;
- l'enlèvement d'un point de choix par l'exportation: cet enlèvement consiste en la comparaison du point de choix à exporter avec le registre B, suivie par la mise à jour des pointeurs de chaînage.

---

[1] Les mesures montrent l'intérêt d'exporter plusieurs points de choix et de conserver sur un processeur le point de choix entier.

La durée de l'exclusion mutuelle dépend du calcul du bloc de points de choix à exporter. Ce calcul correspond à la vérification de conditions d'exportation (cf. chapitre 3, section 3.6).

b) Cohérence des piles

Les incohérences liées à la liaison/déliaison conditionnelle dans la méthode VD (cf. section 4.3.3) peuvent être éliminées par les règles suivantes (voir fig.4.10):

- la valeur d'une liaison est transférée avant sa date et la liaison met à jour la date avant la valeur;
- la valeur d'initialisation de la date ("date-libre"), utilisée par la déliaison, est "infinie" (supérieure à toutes les dates des points de choix).

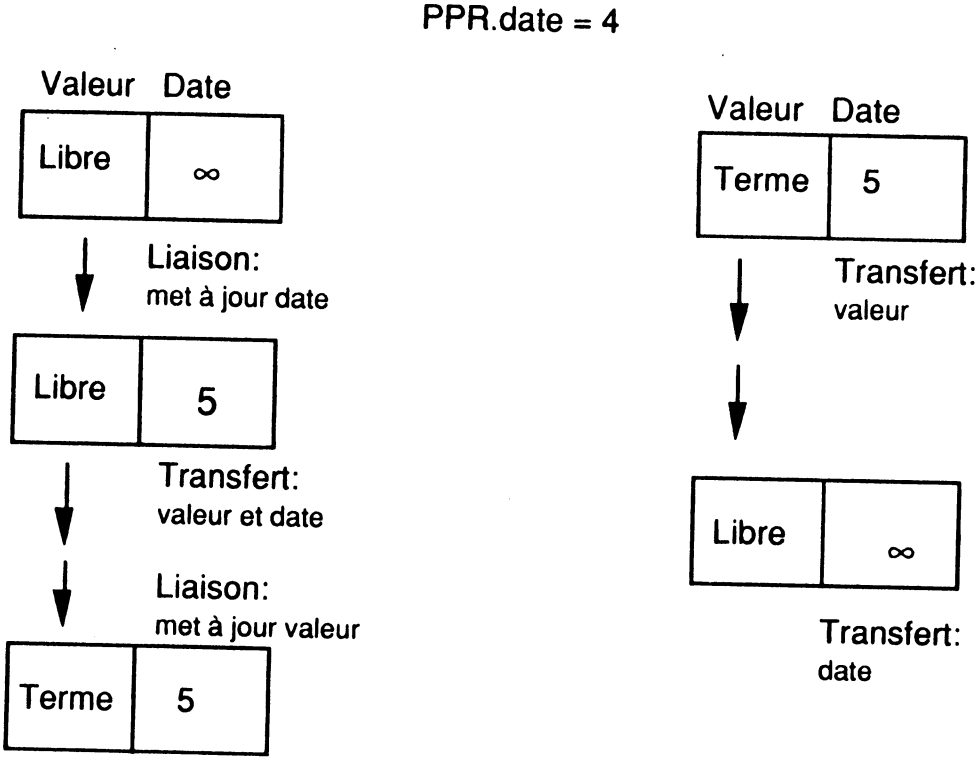

Fig. 4.10 - Transfert de liaison cohérente

Pour éviter de détruire une partie de pile en cours de transfert lors d'un retour-arrière, il faut interdire à la TWAM exportatrice un retour-arrière sur les points de choix exportés.

L'exclusion mutuelle sur la pile de points de choix, lors du retrait des points de choix exportés, fait que ce phénomène correspond à un retour-arrière sur une pile de points de choix vide, c'est à dire, le retour à l'état inactif de la TWAM exportatrice après parcours de tous les points de choix non-exportés. Dans ce cas deux possibilités extrêmes se présentent à la TWAM exportatrice:

- demander une tâche à une troisième TWAM: dans ce cas, l'importation doit se synchroniser avec l'exportation, afin d'éviter la destruction des données à transférer;
- annuler l'exportation en cours: dans ce cas, elle avertit la TWAM importatrice de l'échec de l'exportation, afin que celle-ci puisse chercher du travail auprès d'une autre TWAM. Des points de synchronisation sont nécessaires entre les différentes phases du transfert des piles. Ensuite, elle restaure l'état de la pile de points de choix et procède à un retour-arrière normal.

Si on exporte une seule alternative, la deuxième solution (annulation de l'exportation) est préférable, puisque la TWAM exportatrice peut relancer l'exécution avant la TWAM importatrice.

Dans le cas où plusieurs points de choix sont exportés, il semble plus intéressant d'avoir un compromis entre les deux solutions, c.a.d. un partage équitable entre les 2 TWAM, la TWAM importatrice récupérant les points de choix les plus jeunes, la TWAM exportatrice les plus vieux.

4.4. La Coupure

Jusqu'à présent, tout point de choix pouvait être choisi pour une exécution parallèle, que cette exécution soit ou ne soit pas une source d'efficacité (cf. le chapitre 3, section 3.6). Ceci est vrai en Prolog réduit à la logique des clauses de Horn. Ce n'est pas le cas dès qu'il s'agit de préserver la sémantique du langage Prolog, qui présente des opérateurs dont l'effet dépend de l'ordre d'exécution:

- la coupure;
- les entrées-sorties;
- la lecture et l'écriture dans les bases de faits et de clauses.

Parmi ces prédicats, nous nous sommes intéressés à la coupure avec l'objectif principal de préserver la sémantique séquentielle de celle-ci. La méthode utilisée est d'inhiber le parallélisme en présence d'une coupure ([YAS 84], [ALI 90]).

4.4.1. Le Problème

Dans les méthodes qui interdisent le parallélisme en présence d'une coupure, l'idée de base est le marquage des points de choix qui contiennent des coupures, particulièrement le premier (nommé **Bcut** dans la suite). Les points de choix créés avant Bcut peuvent être exportés, Bcut et ceux postérieurs à Bcut ne le peuvent pas. Le parallélisme potentiel est inhibé. Néanmoins, il faut distinguer deux cas (voir fig. 4.11):

- inhibition totale: la clause courante de Bcut a des coupures à franchir;
- inhibition partielle: les clauses suspendues de Bcut contiennent des coupures, ce qui n'est pas le cas pour la clause courante.

Dans le deuxième cas, tous les points de choix existants dans la branche courante, au-dessus de Bcut, ne sont pas soumis aux coupures de Bcut. En effet, les règles de contrôle de Prolog font que tous ces points de choix seront épuisés avant l'exécution des autres clauses de Bcut. Donc ces points de choix peuvent être exportés, à condition que les autres coupures éventuelles dans ces points de choix, soient respectées. L'inhibition partielle vient du fait que le point de choix Bcut peut être exporté, mais à condition de transférer à la même TWAM importatrice toutes les clauses à droite de la première qui contient des coupures.

De plus, toute méthode doit prendre en compte les cas particuliers suivants:

- un échec apparaissant avant la première coupure de la clause: l'inhibition doit être réévaluée en fonction de la prochaine clause: par exemple, l'échec de *q* dans l'inhibition totale de la fig. 4.11a;
- dernière coupure d'une clause: la dernière coupure de la clause est la seule qui libère les prochains points de choix de la clause: par exemple, $!_2$ libère les alternatives de *m* dans la fig. 4.11a;
- une coupure sans point de choix associé au prédicat contenant la coupure (ceci pouvant être dû à un prédicat à une seule clause, par exemple *n* dans la fig. 4.11, ou à un accès direct à la clause par indexation, ou encore à la destruction anticipée du point de choix au début de la dernière branche): dans ce cas les prochains points de choix, créés avant l'exécution de la coupure,

doivent être inhibés. Ce cas devient plus complexe si on considère plusieurs coupures dans des clauses différentes;

- la combinaison du cas précédent avec des points de choix qui contiennent des coupures (*n* et *p* dans la fig. 4.11a);
- l'indexation par valeur: l'instruction de retour-arrière, qui contrôle le sous-ensemble de clauses à indexer, ne peut pas déterminer l'état de l'inhibition, du fait que la clause choisie n'est pas encore connue à ce moment: par exemple, l'instruction de retour-arrière, après la clause p_6, ne connait pas la prochaine clause (p_7 ou p_8) (fig. 4.11b);

Totale (a)

```
n :- p, !.
p1 :- q, !1, l, !2, m.
p2 :- r.
p3 :- s.
```

Partielle (b)

```
n :- p.
p4(a) :- q.          clause courante
p5(b) :- r, !.
p6(X) :- s.
p7(c).
p8(d).
```

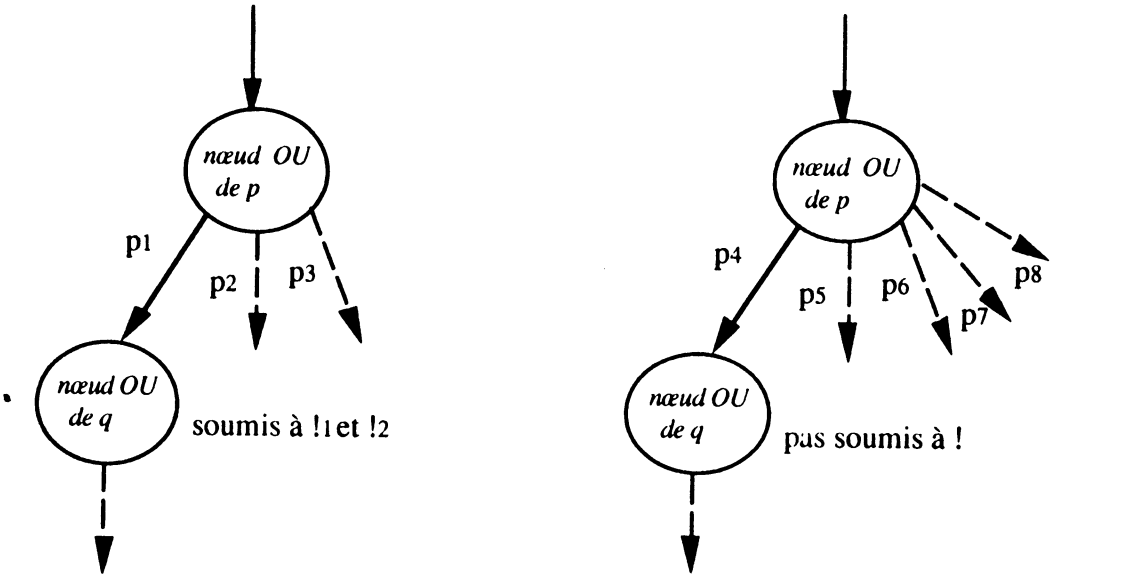

Fig. 4.11 - Inhibition totale et inhibition partielle

- les instructions de retour-arrière sont obligées de contrôler l'inhibition dans tous les cas si l'exportation du point de choix courant est permise, afin d'éviter le vol d'un point de choix qui ne sera inhibé que plus tard, au début de la clause: ce cas s'oppose au précédent;
- coupure globale dans une disjonction (cf. l'annexe C sur la compilation de la coupure): il n'est pas possible d'inhiber et libérer à chaque alternative de la disjonction, mais seulement après la dernière coupure. On risque d'exporter des clauses qui pourraient être coupées par une coupure globale existante dans les prochaines alternatives de la disjonction. Par exemple (dans la fig. 4.12), $!_2$ peut couper p_2 et p_3, donc l'alternative *s* ne peut pas libérer le point de choix de *p*. En outre, il faut réexaminer l'inhibition de la disjonction à cause de coupures locales (*!!*, cf. l'annexe C sur la compilation de la coupure), si la libération globale est effectuée pendant l'exécution de la disjonction. Par exemple (fig. 4.12), le début de *v* peut libérer p_2 et p_3, mais *u* doit être inhibée à cause de !!.

```
p1 :- q, (r, !1; s; t, !2; v, !!; u).
p2 :- ...
p3 :- ...
```

Fig. 4.12 - Coupure globale dans une disjonction

En principe, une méthode efficace devrait éviter un sur-coût sur les clause n'ayant pas de coupures. Ceci est difficile à cause des coupures qui n'ont pas un point de choix associé et à cause des échecs se produisant avant la dernière coupure. Dans la suite, le mot alternative est utilisé indifféremment pour une clause et pour une alternative de disjonction.

4.4.2. Les Principes de la Méthode Opera

Le point de choix courant, au moment où la première alternative avec coupure apparait, est marqué. Il est le dernier point de choix non-inhibé, nommé BFREE. Les autres points de choix, postérieurs à BFREE, ne seront pas marqués.

Les alternatives (clause ou branche de disjonction) contenant des coupures ("coupée"), sans point de choix associé, sont comptées jusqu'à la création d'un nouveau point de choix. On compte aussi l'alternative courante de ce point de choix, si elle contient une coupure.

L'exportation d'un point de choix donné est permise, s'il est BFREE ou s'il est plus vieux que BFREE. L'exportation doit transférer toutes les alternatives du point de choix, ce qui résout l'inhibition partielle.

Le marquage est effectué par les instructions de retour-arrière et par une nouvelle instruction, placée au début d'une clause si celle-ci contient au moins une coupure globale. Le démarquage est effectué par les instructions de retour-arrière ou par la dernière coupure dans la clause. Le comptage, limité jusqu'au premier point de choix inhibé, évite la sauvegarde du compteur dans les points de choix; en contre-partie les fonctions de marquage et de démarquage deviennent plus complexes.

4.4.3. Registres de la TWAM

La méthode emploie trois nouveaux registres:

- NCC: compteur d'alternatives ayant des coupures à franchir, jusqu'à la création de BCUT (inclus);
- BFREE: pointeur vers le point de choix non-inhibé le plus jeune;
- SNCC: sauvegarde de NCC, effectuée si le premier point de choix inhibé contient des coupures.

NO_BFREE est une constante qui indique qu'il n'existe pas de point de choix inhibé. Sa valeur est supérieure au sommet limite de la pile de choix.

NCC et SNCC sont initialisées à 0, et BFREE à NO_BFREE.

4.4.4. Instructions de la TWAM et Compilation

La compilation de la coupure parallèle est basée sur la compilation de la coupure séquentielle. Celle-ci est présentée dans l'annexe C. Cette section décrit brièvement la compilation de la coupure parallèle, dans le cas général d'une coupure globale. Toutes les nouvelles instructions, et les modifications des anciennes, sont détaillées dans l'annexe D. Ceci inclut les cas particuliers de la disjonction, de la coupure locale et de l'indexation de clauses.

Une nouvelle instruction, nommée *inc_ncc*, est engendrée au début de chaque clause qui contient une coupure globale. Elle inhibe le parallélisme, en faisant pointer le registre BFREE

vers le point de choix précédent (plus vieux). Cette instruction contrôle aussi le nombre de coupures sans point de choix, en incrémentant le registre NCC.

Une version de l'instruction *cut*, nommée *last_cut*, se substitue au *cut* pour la dernière coupure de la clause. Elle autorise le parallélisme, en re-initialisant le registre BFREE à NO_BFREE. L'instruction *last_cut* décrémente aussi le registre NCC.

Des versions "coupées" des instructions de retour-arrière (*try's*, *retry's* et *trust's*), sont utilisées dans un prédicat dont une clause au moins contient une coupure globale. Les versions *try's* inhibent le parallélisme (BFREE pointe vers le point de choix précédent), et sauvegardent le nombre de clauses "coupées" sans point de choix (copie du registre NCC dans le registre SNCC). Les versions *retry's* et *trust's* inhibent, ou libèrent, le parallélisme au retour-arrière, en mettant à jour les registres BFREE, NCC et SNCC.

Comme exemple, la compilation des clauses suivantes

```
p1 :- ...
p2 :- r, !, s, !.
p3 :- t.
p4 :- u, !.
```

engendre le code suivant (le squelette), où *YES/NO* indique l'occurrence de coupures dans la clause:

```
        saveB(V)
        try_me_else(p2); code de p1;
p2:     retry_me_else_cutted(p3, YES); inc_ncc(V); tête de p2;
        call(r); cut(V); call(s); last_cut(V); ...;
p3:     retry_me_else_cutted(p4, NO); tête de p3; call(t); ...
p4:     trust_me_cutted(YES); inc_ncc(V), ...
```

Un schéma de l'exécution est montré dans la figure 4.13.

4.4.5. La Synchronisation avec l'Exportation

L'inhibition d'un point de choix, en présence d'une coupure, est effectuée soit au moment où il est le point de choix courant (retour-arrière), soit avant sa création (instruction *inc_ncc*). Il n'est pas possible d'exporter un point de choix, dont la branche courante est inhibée. Par contre, le registre BFREE peut devenir incohérent si le point de choix correspondant est exporté. Une solution contraignante à ce problème est constituée par:

- la mise à jour de BFREE pendant l'exportation;
- l'exclusion mutuelle entre la mise à jour de BFREE par la TWAM (calcul), et la mise à jour de BFREE et du chaînage des points de choix pendant l'exportation.

Une solution moins contraignante est:

- l'exportation ne met pas à jour BFREE (BFREE peut devenir incohérent);
- les instructions de retour-arrière vérifient si le premier point de choix non-inihibé a été exporté (BFREE est incohérent).

La vérification d'incohérence de BFREE est simple, parce qu'il les instructions de retour-arrière vérifient (déjà) si le point de choix précédent (B.B) est le plus jeune non-inhibé (BFREE), c.a.d. (cf.annexe D)

$$BFREE = B.B$$

Dans le cas d'exportation de BFREE, le pointeur *point_de_choix_précédent* (B.B) du premier point de choix inhibé pointe vers le fond de la pile. Alors, la vérification des instructions de retour-arrière devra inclure cette possibilité, c.a.d. BFREE est plus vieux que B.B

BFREE ≥ B.B

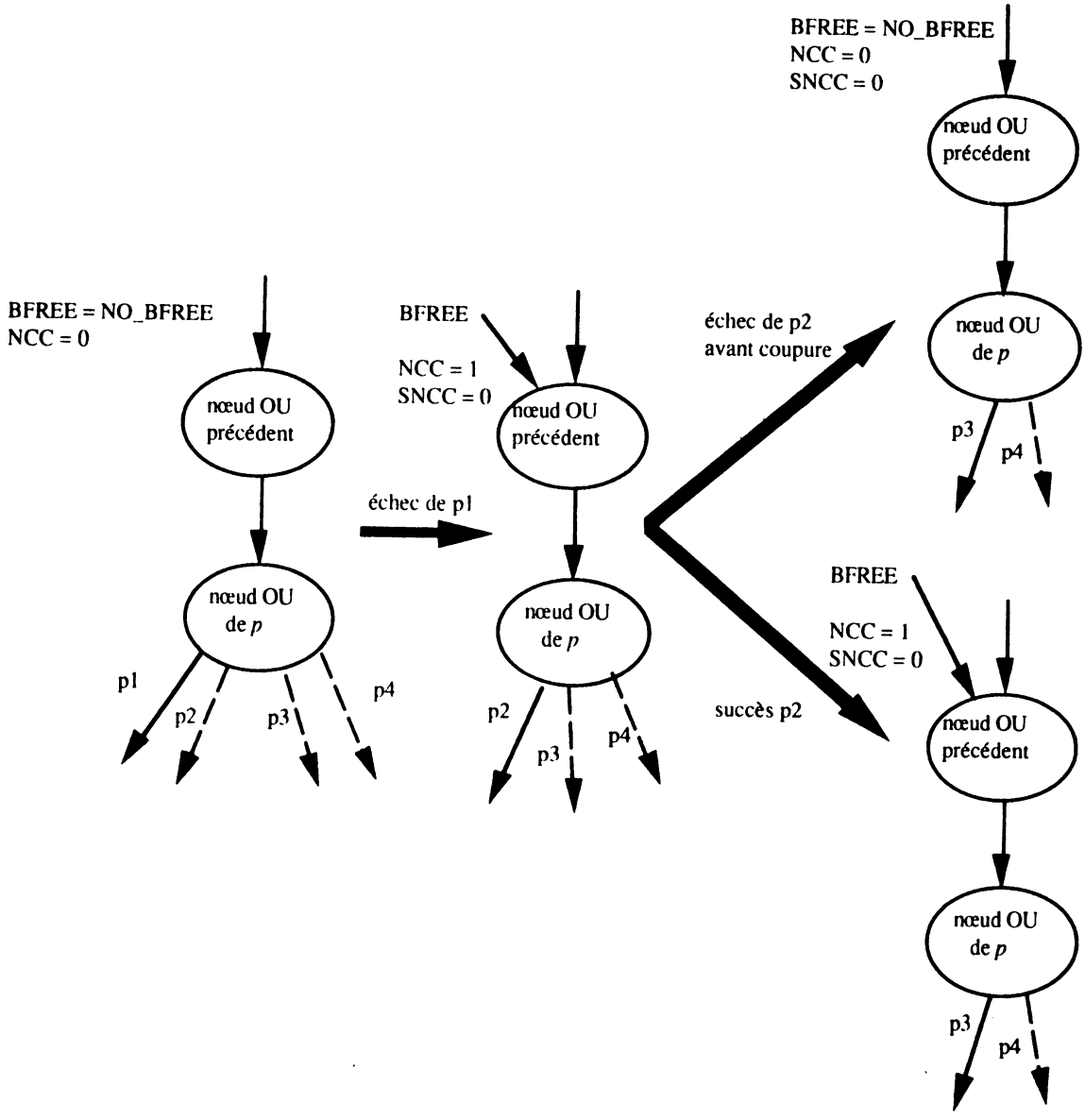

Fig. 4.13 - Exécution de la coupure en parallèle

4.4.6. Autres Méthodes

La méthode [ALI 87] utilise une nouvelle structure dans la pile de points de choix. Cette structure marque les prédicats qui contiennent des coupures. Ces structures sont enchaînées, de la plus récente à la plus vieille; un nouveau registre pointe vers la plus récente. Un autre registre sert à marquer le premier prédicat, dont la branche courante doit franchir des coupures; il pointe vers la structure correspondante. En cas d'échec, la mise à jour de ces registres est complexe. Un drapeau est ajouté à chaque point de choix, indiquant l'existence ou la non-existence des coupures dans le prédicat associé. Ce drapeau permet l'exportation partielle des points de choix dont les prédicats n'ont pas de coupures. La méthode [ALI 87] ne considère pas les coupures

globales dans les disjonctions, mais ceci est plutôt lié à la façon dont la disjonction y est compilée (comme un prédicat obtenu par pliage de la disjonction). En résumé, la comparaison de deux méthodes, [ALI 87] et Opera, indique:

- la consommation de mémoire est plus grande dans [ALI 87];
- les échecs sont plus complexes dans la méthode [ALI 87];
- la méthode [ALI 87] permet l'exportation partielle des points de choix sans coupures. Ceci est dû au drapeau constant, ajouté à chaque point de choix.

Une autre variante a été étudiée dans le cas d'Opera. Le but principal de cette variante est d'assouplir la complexité du traitement des échecs, en utilisant le mécanisme normal de sauvegarde et de mise à jour de l'état de la WAM, par les instructions de retour-arrière. Elle permet l'exportation partielle des prédicats avec des coupures dans certains conditions. Par exemple, les clauses à gauche de la première clause avec des coupures, ou encore, les clauses à droite des coupures, dont les clauses respectives ont échoué, peuvent être exportées partiellement. Cette variante ajoute un seul nouveau registre, employé comme compteur de clauses avec des coupures. Par contre, elle a besoin de deux cellules additionnelles dans tous les points de choix.

4.5. L'ordonnancement des Tâches

Cette section présente l'ordonnancement provisoire de tâches d'Opera. Cet ordonnancement, destiné à une évaluation d'Opera, a pour simple objectif de permettre la prise de mesures de fonctionnement, préliminaire obligatoire à une analyse du comportement de programmes parallèles. Cette analyse permettra, peut-être, la conception d'un ordonnanceur plus élaboré.

Certaines abréviations, introduites dans le chapitre 3, sont rappelées:

- S: tâche exportée;
- A: tâche restante;
- TC_k: durée d'exécution de la tâche *k*;
- TE_k: coût d'exportation de la tâche *k*;
- TI_k: coût d'importation d'une tâche *k*.

Les conditions garantissant un accroissement de performance par mise en parallèle de 2 tâches sont:

- CP_e: condition minimale d'exportation: $TC_s > TE_s$;
- CP_i: condition minimale d'importation: $TC_a > TI_s$.

Le principe de base de l'ordonnanceur provisoire est l'application d'une heuristique simple d'évaluation de la charge de chaque TWAM. Faisant abstraction de l'implantation, considérons qu'un processus unique, nommé Ordonnanceur, est responsable de toutes les fonctions.

Les règles ou contraintes du contrôle initial sont:

- C_1: exportation d'un seul point de choix (nœud OU): le plus vieux;
- C_2: interdiction d'exportation si la TWAM exportatrice n'a pas un nombre minimum de nœuds (NMN);
- C_3: choix de la TWAM exportatrice en fonction de la charge la plus grande;
- C_4: choix de la TWAM importatrice en fonction de la taille la plus petite de données à transférer.

La règle C_1 garantit le moindre coût d'installation. Par contre, la charge peut être faible, donc la vérification de CP_e est incertaine. La contrainte C_2 est une tentative de garantir CP_i en

essayant de maintenir une charge résiduelle suffisante (TC_a). On oublie les coûts d'installation. En outre, le seuil "idéal" NMN peut varier substantiellement d'un programme à l'autre.

4.5.1. Classification selon la Charge

Chaque TWAM est classée suivant sa charge:

- inactive: la TWAM n'a aucune tâche à exécuter;
- muette: la TWAM a des tâches à exécuter, mais pas en nombre suffisant pour pouvoir en exporter (contrainte C_2);
- surchargée: la TWAM a trop de tâches à exécuter; une exportation est possible.

La différence précise entre muette et surchargée est déterminée par une valeur de charge, nommée SEUIL_SURCHARGE.

4.5.2. Evaluation de la Charge

Deux heuristiques d'évaluation de charge ont été proposées, prenant comme base soit le nombre de points de choix, soit la date du point de choix le plus vieux.

a) Nombre de points de choix:

Dans une même branche (TWAM), la charge (durée d'exécution) croît avec le nombre de points de choix. La durée d'exécution d'un point de choix, est comptée dans la durée d'exécution du point de choix précédent (plus précisément de l'alternative courante du point de choix précédent).

Cependant la relation d'ordre qu'on peut établir en nombre de points de choix de deux TWAM, ne peut être étendue à la charge. La charge d'un point de choix varie en fonction:

- du nombre d'alternatives;
- de la taille, du type et du contenu, des arguments de l'appel (sous-but);
- de la complexité de chaque alternative;
- de la continuation de la résolvante.

Par exemple, certains prédicats itératifs, où la clause de terminaison est la dernière, créent un grand nombre de points. Ils n'ont aucun parallélisme, si la clause de terminaison échoue toujours à l'exception du dernier appel. Au retour-arrière, la TWAM les détruit rapidement. On risque donc de considérer comme surchargée une TWAM qui redevient très vite inactive. Un exemple est le prédicat *itere/1* suivant, qui exécute la tâche *tache/1* N fois:

```
itere(N) :- N > 0, tache(N), N1 is N - 1, itere(N1).
itere(0).
```

Dans un appel comme *itere(10)*, 11 points de choix sont créés, mais la deuxième clause échoue les 10 premières fois.

Au début de l'exécution, un petit nombre de points de choix peut représenter une grande charge; par contre, à la fin il peut se produire la situation opposée. En conséquence, prendre comme charge le nombre de points de choix ne peut être qu'une approximation.

Le comptage des points de choix est réalisé par la TWAM, à chaque création et à chaque destruction d'un point de choix. Ceci introduit un sur-coût faible, constant et indépendant du parallélisme. Un nouveau registre, nommé LOAD, maintient le nombre courant de points de choix.

Remarque:

Ce registre est sauvegardé avec le registre B dans la structure représentant une coupure à exécuter(*SaveB*), au début des prédicats avec coupures (instruction *saveB*). Ceci est nécessaire afin de permettre la mise à jour correcte du nombre de points choix au moment de l'exécution de la coupure (instructions *cut* et *lastCut*).

b) Date du point de choix le plus vieux:

Cette heuristique utilise le nombre de points de choix et la date du plus vieux (plus proche de la racine) point de choix. Le nombre de points de choix est encore utilisé pour distinguer les états "muette" et "surchargée". Cependant, les TWAM surchargées seront classées en fonction de la date de leur point de choix le plus vieux. La charge est autant plus grande que la date est plus petite. Ici, l'heuristique considère que les branches, les plus proches de la racine de l'arbre OU, sont les plus chargées. Ceci n'est pas vrai dans tous les cas; c'est aussi une approximation de la charge réelle. Une propriété intéressante de cette heuristique est que, sur une même branche, les coûts d'installation croissent avec la date, donc, en choisissant la date plus petite, on peut espérer minimiser les coûts d'installation.

4.5.3. Conditions Minimales du Parallélisme

Les conditions minimales (CP_i et CP_e) ne sont pas directement appliquées, du fait qu'il faut transformer l'unité de charge, par exemple la date, en unité temps. Opera utilise une variante imprécise de la condition d'importation (CP_i): c'est un palliatif. Cette variante classe les TWAM actives en muettes et surchargées. Basée sur le nombre de points de choix, elle peut échouer en choisissant les exportations qui violent CP_i, ou, au contraire, en rejetant les exportations accroissant l'efficacité. Par conséquence, elle ne peut pas garantir une augmentation de la performance.

En outre, il est important de contrôler la constante SEUIL_SURCHARGE. Si elle est trop élevée, il peut se produire artificiellement une situation où le nombre de TWAM surchargées n'est pas suffisant pour satisfaire les TWAM inactives. Dans ce cas, des possibilités réelles d'augmentation de performance sont gaspillées. Si elle est trop petite, on risque de diminuer l'efficacité par des exportations qui violent CP_i et CP_e.

La valeur de la constante SEUIL_SURCHARGE sera fixée à partir d'une série de mesures sur des programmes de test de façon à trouver un compromis satisfaisant, compte tenu du comportement des programmes parallèles.

Une solution intéressante à évaluer serait l'ajustement dynamique de ce seuil aux programmes. Le principe pourrait être d'augmenter la valeur du seuil à partir d'un certain taux de violation de la condition CP_i (la TWAM exportatrice devient inactive avant la fin du transfert). Ce taux devient à son tour une constante de système, à évaluer sur une campagne de mesures.

4.5.4. Régulation de Charge

Cette section se préoccupe de l'appariement de deux TWAM, de façon à assurer l'exécution la plus efficace (cf. l'étude initiale dans le chapitre 3, section 3.6.2). Dans la situation où n TWAM ($n > 1$) sont surchargées et une seule est inactive, on choisit la TWAM la plus chargée. Ce critère réduit la durée d'exécution du programme, en réduisant la durée d'exécution de la plus chargée des TWAM actives. Le problème est que la relation d'ordre en nombre de points de choix (ou en âge) ne correspond pas à celle établie en durée d'exécution.

Dans le cas où n TWAM ($n > 1$) sont inactives et une seule est surchargée, on choisit en fonction de la plus petite longueur de piles (section non-commune) c.a.d. du plus petit coût d'installation. Ce critère est en accord avec l'étude initiale (dans le chapitre 3, section 3.6.2). Connaissant les points de choix communs, on choisit la TWAM inactive dont la date du point

de choix commun avec la TWAM surchargée est la plus grande. Dans une même branche, la relation d'ordre établie par la date correspond à celle établie par la longueur de piles.

Il est possible de calculer la tâche *S* en appliquant l'équation suivante:

$$TC_a + TE_S = TC_S + TI_S$$

Rappelons les difficultés de l'application de cette formule: le calcul de *S* se réalise par itérations, en augmentant *S* (c.a.d. le nombre d'alternatives) à chaque étape; les durées d'exécution (TC_a, TC_S) ne sont pas connues. L'ordonnanceur actuel utilise donc une simplification extrême. Les coûts TE_S et TI_S sont considérés comme constants, TI_S étant plus grand que TE_S. La charge de chaque point de choix est considérée comme constante. La TWAM exportatrice coupe la pile de points de choix en deux, de façon à exporter une fraction *fc* du nombre de points de choix. *fc* est un paramètre du système, qui doit être évalué par une campagne de mesures.

Initialement, nous proposons *fc* égal à 0,3. La différence en faveur de l'exportatrice est due au fait que TE_S est plus petit que TI_S, et que les points de choix plus proches de la racine (ceux exportés, *S*) sont en principe plus chargés.

4.6. L'architecture du Logiciel: Processus et Protocole

Nous terminons ce chapitre par une brève présentation de l'architecture du logiciel du système Opera, et de sa projection sur l'architecture du Supernode (cet aspect est détaillé dans la thèse de Michel Favre).

Nous présentons tout d'abord l'organisation en processus du modèle Opera, et ensuite le protocole de communication entre les processus. L'architecture choisie pour l'ordonnanceur impose des optimisations spécifiques à la méthode décrite dans la section précédente.

4.6.1. Organisation en Processus

Le modèle Opera peut être vu comme l'interaction de 4 parties:

- le calcul: cette partie est composée par les TWAM, exécutant le code du programme Prolog;
- l'installation de tâches entre deux TWAM;
- la prise et l'enregistrement de mesures sur le comportement d'une TWAM, nécessaires à l'ordonnancement;
- l'ordonnancement.

L'installation de tâches a deux facettes:

- l'exportation, c.a.d. le choix des points de choix à exporter, leur extraction de la pile de points de choix et l'envoi des piles: cela est effectué en parallèle au calcul (voir la section 4.3.7 sur la cohérence des données);
- l'importation, c.a.d. la réception des piles, la déliaison locale, l'initialisation de la section non-commune de la pile de variables, l'installation des liaisons conditionnelles et la mise à jour préliminaire de l'état de la TWAM.

Les mesures à prendre et à enregistrer sont le nombre de points de choix, la date du premier point de choix et les points de choix communs aux deux TWAM. L'ordonnancement comprend la vérification des conditions minimales, la régulation de charge et le maintien des points de choix communs.

Dans une machine à mémoire commune, la réalisation de l'ordonnanceur est simple et évidente. La mémoire commune permet à chaque TWAM d'accéder efficacement aux données de toutes les autres, et peut supporter des structures globales de contrôle. Chaque TWAM est

responsable du maintien de sa partie, mais peut accéder à la structure complète. Une TWAM, devenant inactive, peut assumer entièrement les fonctions de contrôle. La synchronisation se ramène à un problème d'exclusion mutuelle à ces données communes.

Dans une machine sans mémoire commune, la conception d'un ordonnanceur distribué est plus complexe, et il faut assurer que l'efficacité ne se dégrade pas à cause du flux de communication nécessaire à la mise en œuvre de cet algorithme ([BOS 90]).

A côté d'une architecture uniformément distribuée d'un ordonnanceur, il existe deux architectures plus simples:

- l'architecture centralisée;
- l'architecture hiérarchisée.

Dans l'organisation centralisée, un seul processeur est responsable de l'ordonnancement, et toutes les communications de mesure et de contrôle sont effectuées vers/depuis ce processeur. Ceci peut devenir un goulot d'étranglement. Dans de petites configurations de machines parallèles (par exemple à une dizaine de processeurs), ce goulot est négligeable. Par contre, dans des configurations de l'ordre de la centaine de processeurs, les conflits d'accès pourront dégrader les performances générales, si le support des communications des TWAM vers l'ordonnanceur, ou l'ordonnanceur lui-même, ne sont pas assez efficaces.

La conception d'un algorithme hiérarchisé est plus complexe que celle d'un algorithme centralisé, mais il réduit l'importance du goulot pour de grandes configurations. Un premier niveau de processeur se partage le contrôle des TWAM. Les processeurs d'un niveau donné pilotent des groupes du niveau inférieur et le dernier est le processeur maître.

Pour le prototype d'Opera, nous avons choisi une organisation centralisée, pour des raisons de:

- facilité de conception et de mise au point;
- bonne adaptation aux caractéristiques architecturales de la configuration de base de la machine cible (le Supernode).

En effet, l'architecture du Supernode comporte un processeur maître spécialisé pour le contrôle des autres processeurs, ainsi que du réseau d'interconnexion. Etant donné que le contrôle est centralisé, il faut considérer l'envoi des mesures de comportement et des données de contrôle, de chaque TWAM vers le processeur de contrôle. Un "accès" aux données locales de toutes les TWAM, effectué par le processeur de contrôle, au moment d'une décision, serait inefficace.

De plus, le maintien d'un état global et exact, par le processeur de contrôle, exige une synchronisation à chaque modification partielle de cet état par une TWAM. Une communication asynchrone aura un coût considérablement plus faible que le précédent, mais l'état d'exécution est imprécis. Il faut donc prévoir que l'ordonnanceur puisse échouer, c.a.d. élire une TWAM surchargée dans le passé, qui ne l'est plus au moment de son élection. En précisant, la TWAM élue comme exportatrice peut être dans l'état inactive, ou dans l'état muette, au moment où elle est mise au courant de la décision.

La prise et l'envoi de mesures pourrait être réalisée par la TWAM. Cependant, il est préférable de donner une certaine indépendance à cette prise de mesures par rapport au calcul. Par exemple, par un réglage du pas d'échantillonnage ou du seuil de déclenchement de la mesure, il est possible de masquer des fluctuations microscopiques, qui seraient inutiles à l'ordonnancement et surchargeraient les voies de communication et l'ordonnanceur.

Ceci conduit à une organisation en processus où les fonctions précédentes sont assurées par:

- un processus **Ordonnanceur** unique, responsable de l'ordonnancement;
- trois processus par TWAM:

  - le **Calculateur**: responsable du calcul (Prolog) et de l'importation de tâches;
  - l'**Exportateur**: responsable de l'exportation de tâches;
  - l'**Espion**: responsable de la prise, de la classification et du filtrage de mesures, et de leur envoi à l'Ordonnanceur.

La figure 4.14 présente un schéma de cette organisation et du protocole de messages entre les processus. Les prochaines sections présentent la structure algorithmique de chaque processus, en relevant le protocole de messages.

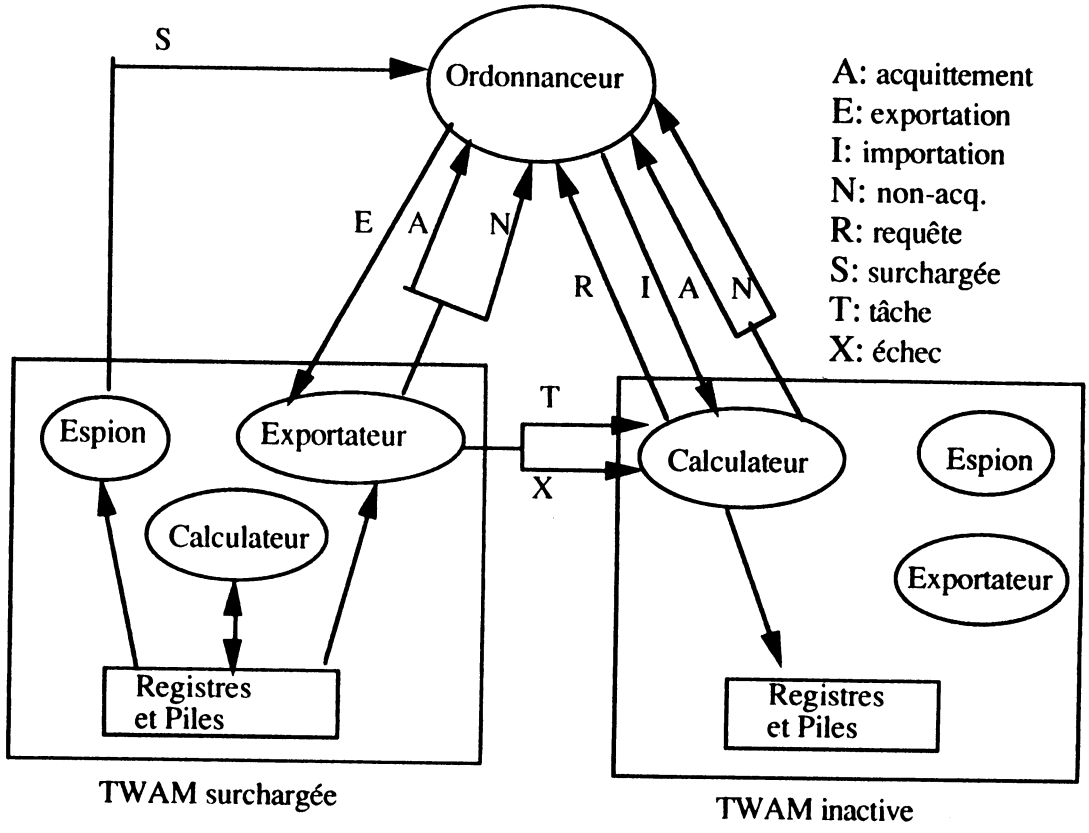

Fig. 4.14 - Organisation en processus d'Opera

4.6.2. L'Ordonnanceur

L'Ordonnanceur a trois fonctions:

- l'initialisation et la terminaison de l'exécution;
- l'appariement de deux TWAM, l'une surchargée et l'autre inactive;
- le contrôle de points de choix communs.

Afin d'accomplir ces fonctions, il maintient deux structures principales:

- un état approché de la charge de chaque TWAM, en les classant en trois catégories: inactive, muette, surchargée;
- une matrice dont chaque élément décrit le point de choix commun à deux TWAM.

A l'état initial, l'Ordonnanceur envoie un message d'initialisation à une seule TWAM, et des messages d'attente à toutes les autres. Ensuite il devient un processus itératif, qui ne travaille que lors de la réception d'un message envoyé par une TWAM. Ces messages, et les actions correspondantes, sont:

- sur une requête de tâche (qui correspond à une mesure de charge nulle):
  - l'état de la TWAM émettrice est mis à jour;
  - s'il existe des TWAM surchargées, il effectue un appariement: l'Ordonnanceur choisit une TWAM surchargée, lui envoie un ordre d'exportation, et envoie un ordre d'importation à la TWAM inactive. Les deux messages contiennent le nom de l'autre TWAM, et la date du point de choix commun. Les deux TWAM sont provisoirement considérées comme muettes, en attendant un acquittement de ces deux TWAM.
- une mesure de charge: l'état de la TWAM est mis à jour. Si l'état devient "surchargée" et s'il existe des TWAM inactives, l'Ordonnanceur effectue un appariement, comme décrit ci-dessus (il choisit une TWAM inactive):
- un acquittement de la TWAM importatrice, qui contient le nouveau point de choix commun avec l'exportatrice, et sa nouvelle charge: l'état (charge et points de choix communs) de la TWAM est mis à jour;
- un échec (non-acquittement) de la TWAM importatrice: action identique à celle de la requête de tâche;
- un acquittement, ou échec, de la TWAM exportatrice, en contenant sa nouvelle charge: ce message correspond à une mesure de charge.

L'Ordonnanceur termine l'exécution au moment où toutes les TWAM sont inactives.

4.6.3. L'Espion

L'Espion travaille périodiquement, la fréquence étant un paramètre du système. Il observe la charge de la TWAM, et il envoie une mesure de charge à l'Ordonnanceur, si la charge actuelle est "significativement" différente de celle qui est connue de l'Ordonnanceur. Le mot significativement équivaut à un autre paramètre du système, qui définit les seuils que la mesure doit franchir pour être transmise à l'Ordonnanceur.

Dans le cas de l'évaluation par le nombre de points de choix, la charge transmise (si "surchargée") est le nombre de points de choix divisé par un paramètre, nommé TRANCHE_SURCHARGE. L'Ordonnanceur reçoit donc une nouvelle mesure de charge, si soit la TWAM a changé d'état, soit elle a changé de tranche de surcharge.

Dans le cas de l'évaluation par date, la charge "surchargée" est transmise à la moindre variation, ce qui correspond à des tranches de valeur 1.

L'Espion n'envoie pas la charge nulle, celle-ci correspondant à une requête de tâche par le Calculateur.

L'Ordonnanceur écarte des futurs choix les deux TWAM concernées par un transfert en cours. En conséquence, les espions respectifs n'ont pas besoin de travailler durant le transfert; ceci réduit le flux de communication. Dans le cas de l'importateur, il faut soit sérialiser les messages de l'Espion avec la requête de tâche émise par le Calculateur, soit que l'Ordonnanceur ignore les messages de l'Espion, reçus après la requête de tâche jusqu'à l'acquittement (ou le non-acquittement).

4.6.4. Le Calculateur

A l'état initial, le Calculateur attend une directive de l'Ordonnanceur; elle peut être de deux types:

- exécuter le but (tâche) principal;
- attendre un ordre d'importation: ce message, qui semble inutile, est utilisé, de même que le premier, pour envoyer des données d'initialisation.

Dans les deux cas, le Calculateur devient un processus itératif (dans le premier cas, il se remet tout suite à l'étape B ci-dessous):

- A: réception d'un ordre d'importation depuis l'Ordonnanceur, qui indique le nom de la TWAM exportatrice;
- réception d'un message de confirmation ou d'échec, de la TWAM exportatrice;
- si le message confirme l'appariement:
  - importation de la tâche (suivant la méthode VD ou VV): la TWAM exportatrice envoie le point de choix commun aux deux TWAM;
  - envoi d'un message d'acquittement à l'Ordonnanceur;
  - B: exécution de la tâche: cette étape se termine au moment où toutes les alternatives de la tâche ont été exécutées ou exportées (la pile de points de choix est vide);
  - envoi d'une requête de tâche à l'Ordonnanceur: ce pas ne peut s'effectuer que si l'Exportateur de cette TWAM est en attente. Dans le cas contraire, le Calculateur se bloque en attendant un signal de réveil de l'Exportateur;
  - répéter (première étape A ci-dessus).
- si le message de la TWAM exportatrice est un échec:
  - envoi d'un message de non-acquittement à l'Ordonnanceur;
  - répéter (première étape A ci-dessus).

4.6.5. L'Exportateur

L'Exportateur répète les actions suivantes:

- réception d'un ordre d'exportation depuis l'Ordonnanceur, en indiquant le nom de la TWAM importatrice et les tailles des sections communes des piles;
- (re) vérification de la condition d'importation (LOAD ≥ SEUIL_SURCHARGE), ce qui peut produire un échec;
- exportation de la tâche, ou un message d'échec, à la TWAM importatrice;
- envoi d'un message d'acquittement, ou d'échec, vers l'Ordonnanceur;
- envoi d'un signal de réveil au Calculateur de sa TWAM (si celui-ci est bloqué).

Entre le début de la vérification de la condition d'importation et la fin du transfert des points de choix (premier pas de l'exportation), le Calculateur ne peut pas modifier les points de choix exportés (voir section 4.3.7 sur la cohérence des données transférées).

4.7. Conclusions

Ce chapitre a présenté le modèle Opera, sujet principal de cette thèse. Opera a été conçu pour le parallélisme OU multi-séquentiel en Prolog, et pour les machines sans mémoire commune.

Le but d'Opera est l'exploitation automatique du parallélisme OU d'un programme Prolog: le programmeur n'en intervient pas et la sémantique du langage n'est pas modifiée.

L'implantation du langage se réalise par compilation vers la TWAM, une machine abstraite efficace basée sur la WAM. Ceci évite des fausses interprétations des gains de performance du système parallèle.

La duplication des sections communes aux TWAM est proposée pour la gestion des contextes multiples. Le problème principal de ce type de gestion est l'installation de tâches, qui, pour être efficace, doit :

- ne transférer que les portions de piles non-dupliquées (encore) dans les deux TWAM, exportatrice et importatrice (E1);
- réduire les synchronisations entre le calcul (Prolog) et l'installation de tâches (E2);
- éviter le transfert des données non-nécessaires à la TWAM importatrice, comme les liaisons conditionnelles plus récentes que le nœud OU exporté le plus jeune (PPR) (E3);
- profiter de la force de calcul de la TWAM importatrice (processeur) pendant la réception des sections communes (E4).

La copie incrémentale et la pile de variables sont respectivement des solutions aux condition E2 et E1.

Deux méthodes de gestion des contextes multiples ont été proposées, les deux utilisant la pile de variables. La première méthode, nommée "pile de variables plus datation" (VD), fait supporter à la TWAM importatrice la déliaison non-locale (condition E4), mais elle transfère les liaisons conditionnelles non-valides (violation de E3) et rend difficile la copie incrémentale.

La deuxième méthode, dite "pile de variables et traîné plus valeur" (VV), n'a pas ces désavantages et semble plus efficace.

Des mesures expérimentales sont nécessaires afin de confirmer la meilleure efficacité de la méthode VV.

Le contrôle des sections communes, exigé par la copie incrémentale, n'est pas trivial dans une machine sans mémoire commune. Une proposition simple le résout dans Opera, en échange de restrictions imposées sur:

- les règles d'ordonnancement: un point de choix ne peut être exporté que si ont été exportés ou épuisés tous les points de choix plus vieux que celui-là;
- la section commune: elle est limitée par le point de choix le plus vieux entre ceux qui sont exportés.

Une représentation indépendante des points de choix communs à deux TWAM est utilisée dans le contrôle des sections communes.

Des règles d'ordonnancement, en accord avec celles employées dans la copie incrémentale, simplifient la préservation de la cohérence de la pile de points de choix: les points de choix exportés sont contigus, le premier étant toujours le plus vieux.

Une méthode efficace a été proposée pour l'implantation de la coupure en parallèle. Elle préserve la sémantique classique de la coupure, en inhibant le parallélisme si la branche courante a des coupures à franchir.

L'ordonnancement, qui est le problème principal d'Opera, a été provisoirement résolu, de façon à permettre la prise de mesures expérimentales. Ces mesures seront utilisées dans l'étude

d'un ordonnancement plus efficace, qui devra garantir un accroissement de performance par parallélisation.

Cette solution provisoire évalue la charge par deux heuristiques: nombre de points de choix et date du point de choix le plus vieux. On classe les TWAM actives en muettes et surchargées: on cherche à éviter l'exportation d'un point de choix qui serait immédiatement nécessaire à la TWAM exportatrice (tentative de non-violation de CP_i).

On exporte plusieurs points de choix (un point de choix au moins) afin d'avoir une tâche exportée significative (tentative de non-violation de CP_e).

La TWAM la plus chargée est choisie par une TWAM inactive. En revanche, pour une TWAM surchargée, on minimise les transferts de piles, en élisant la TWAM inactive par le volume de la section commune.

Un processus centralisée effectue l'ordonnancement, en recevant des demandes d'importation de tâches des TWAM inactives, et des mesures de charge des TWAM surchargées. Trois processus composent une TWAM:

- le Calculateur: responsable du calcul séquentiel (Prolog) et de l'importation de tâches;
- l'Exportateur: il exporte les tâches;
- l'Espion: il prend des mesures de charge et les envoie à l'Ordonnanceur.

5. Chapitre 5
Le Prototype d'Opera et ses Résultats

Ce chapitre présente l'implantation d'un prototype du modèle Opera sur le Supernode, une machine parallèle sans mémoire commune. D'abord, le Supernode est brièvement décrit. Deuxièmement, nous détaillons le pliage de l'architecture d'Opera sur la configuration de base du Supernode, nommée Tnode. Ensuite, les différences principales entre la définition du modèle Opera et ce prototype sont présentées, de façon à pouvoir évaluer précisément les résultats obtenus sur les quelques programmes Prolog de test. Après présentation de ces résultats, le chapitre s'achève par la liste des outils de développement, conçus et implantés pour Opera.

5.1. Le Supernode

Le Supernode est une machine parallèle, du type MIMD (Multiple Instruction, Multiple Data), sans mémoire commune, développée dans le cadre du projet Esprit 1085 ([HAR 86], [MUN 89], [WAI 90]). Ses caractéristiques les plus importantes sont:

- la communication inter processeurs, intégrée à son processeur de base (le Transputer), est inspirée du modèle de processus communicants CSP ([HOA 78]);
- le réseau d'interconnexion (graphe de degré 4) est dynamiquement configurable;
- l'architecture est modulaire et hiérarchisée;
- une voie de contrôle offre une communication additionnelle entre les *n* processeurs de travail (les nœuds) et 1 processeur de contrôle.

Nous décrirons la machine telle qu'elle est fabriquée par la société Telmat.

Le processeur de base (nœud):

Le processeur de base du Supernode appartient à la famille de processeurs conçue et produite par Inmos: le Transputer.

Un premier objectif ayant présidé à la conception du Transputer est de pouvoir programmer systématiquement en terme de processus. Le modèle de processus choisi est le modèle de processus communicants de Hoare (CSP [HOA 78]). Le rendez-vous est utilisé comme protocole de communication.

Le Transputer possède un noyau microprogrammé de gestion de processus. Des instructions spécifiques existent pour la création et la destruction de processus, un nombre arbitraire de processus pouvant être exécuté de façon concurrente sur un même Transputer. Quelques microsecondes sont nécessaires à la commutation de contexte de processus.

L'efficacité de cette implantation des processus permet de les utiliser systématiquement dans la programmation.

Les processus d'un Transputer peuvent communiquer entre eux par des canaux logiques, lesquels sont programmés de la même façon que les 4 canaux physiques de communication (lien).

Un second objectif est de faire du Transputer la brique de base pour la construction de machines fortement parallèles sans mémoire commune.

Cet objectif est atteint:

- par l'intégration dans le Transputer de 4 processeurs de communication, que les processus invoquent selon le protocole de communication normal;
- par la simplicité de la liaison physique entre Transputers: on connecte un lien d'un Transputer à un lien du deuxième Transputer, sans utiliser des dispositifs de communication additifs.

Le Transputer T800, le plus puissant à l'époque, possède les caractéristiques suivantes:

- 4 liens de communication série bi-directionnelles à 20 Mbits/s chacun;
- une unité de calcul de 32 bits à 20 Mhz (10 Mips);
- une unité de calcul de virgule flottant de 64 bits (2,25 Mflops à 30 Mhz);
- une mémoire interne d'accès rapide, de 4 Koctets;
- une interface de mémoire (externe), pouvant accéder jusqu'à 4 Gigaoctets.

La configuration de base:

La configuration de base du Supernode est constituée de 16 ou 32 Transputers de travail (nœuds), un Transputer de contrôle, un commutateur de liens et une voie de contrôle (fig. 5.1). La connexion de deux liens de deux nœuds quelconques se réalise par le commutateur de liens. Il est commandé par le processeur de contrôle, et tous les graphes de degré 4, de 16 ou 32 nœuds, sont réalisables. Les 4 liens d'un Transputer sont nommés Nord, Sud, Est et Ouest, et certaines liaisons physiques, par exemple Sud-Ouest, ne sont pas possibles du fait que le commutateur est composé de deux commutateurs 72 X 72. Cette restriction augmente la complexité de l'algorithme de configuration (cf. Waille [WAI 90]). Le commutateur 72 x 72, étant conçu par le **R**oyal **S**ignal and **R**adar **E**tablishment dans le cadre du projet Esprit, est nommé commutateur RSRE.

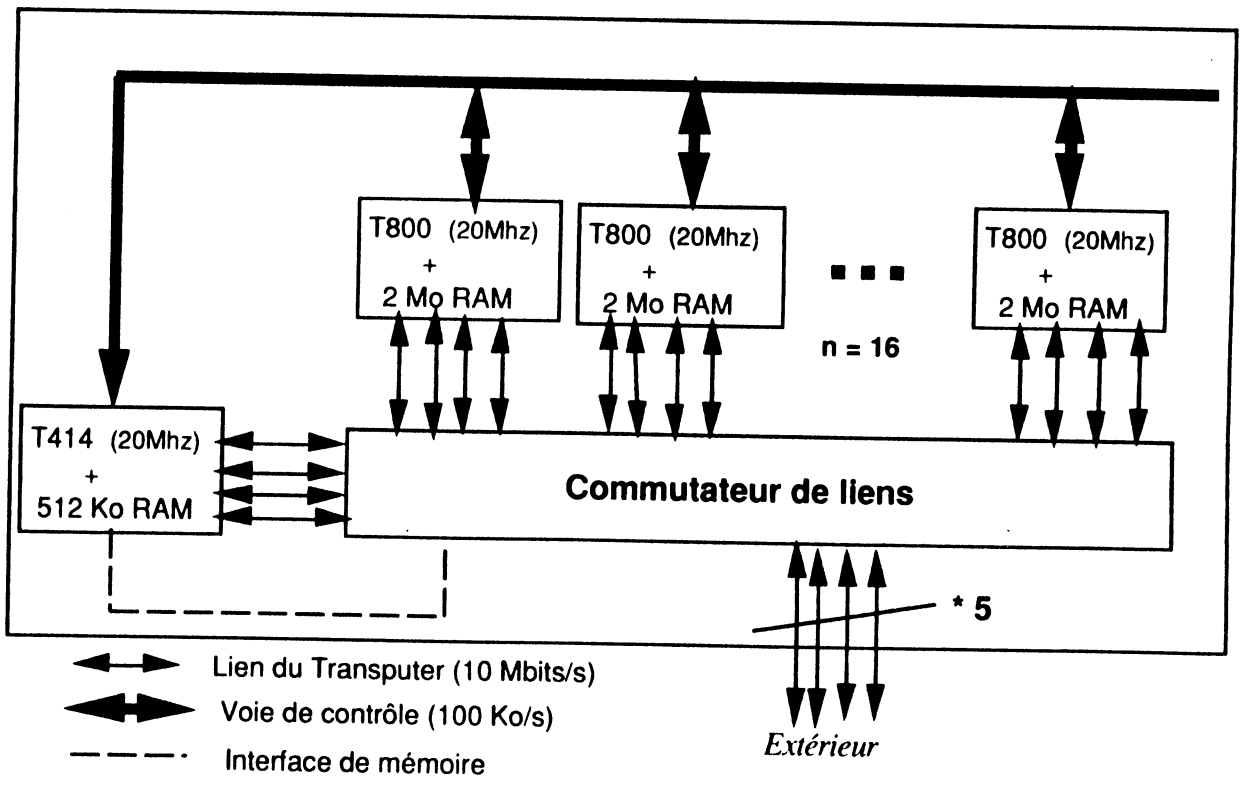

Fig. 5.1 - Configuration du Tnode à 16 Transputers

La voie de contrôle permet la communication entre les nœuds et le Transputer de contrôle. C'est une voie parallèle (8 bits), du type maître-esclave (contrôle-nœuds), à 100 Koctets/s. Cette voie offre deux primitives de communication:

- le transfert d'un octet depuis un nœud donné vers le Transputer de contrôle;
- la diffusion d'un octet depuis le Transputer de contrôle vers un ensemble donné de nœuds.

deux primitives de synchronisation:

- la requête OU: un nœud quelconque d'un ensemble donné de nœuds émet un signal vers le Transputer de contrôle. Celui-ci peut envoyer, en acquittement, un signal de retour vers le nœud;
- l'acquittement Et: le Transputer de contrôle diffuse un signal vers un ensemble donné de nœuds. Il reçoit un signal d'acquittement lorsque tous les nœuds acquittent son signal.

et des primitives de sélection de nœuds, commandées par le Transputer de contrôle et nécessaires à l'exécution des primitives précédentes.

Tous ces primitives consomment un cycle de la voie, de durée 3,2 microsecondes. Cependant, en fonction de la disposition physique des nœuds de l'ensemble, une sélection ou une requête OU peuvent exiger l'exécution de 1 à 4 commandes de primitives. De plus, des fonctions plus élaborées sont implantées par l'action combinée de plusieurs primitives. Par exemple, un appel du Transputer de contrôle par un nœud quelconque, avec transfert d'un octet, consomme en moyenne 5,5 cycles de la voie ([WAI 90]).

Le lecteur, intéressé par plus des détails sur le commutateur de liens et la voie de contrôle, peut consulter [WAI 90].

Le module de base (Tandem):

La configuration suivante du Supernode, nommée Tandem, est le module de base pour la construction de machines plus performantes. Un Tandem possède 2 Transputers de contrôle et 4 commutateurs de liens RSRE. Ces derniers permettent d'interconnecter jusqu'à 64 nœuds. Dans le Tandem, les liaisons réalisables ne sont que Nord-Sud et Est-Ouest, toutes les 4 (Sud-Nord et Ouest-Est) bi-directionnelles. Chaque Transputer de contrôle ne commande que deux RSRE. Ceci augmente la complexité de la programmation de l'algorithme de configuration, parce que l'un des Transputers de contrôle est le contrôleur de la machine et le maître du deuxième Transputer de contrôle. Celui-ci devient responsable seulement de l'exécution de commandes de commutation vers ses 2 RSRE, le maître étant la source de ces commandes.

Les machines hiérarchisées (Méganode):

Une machine hiérarchisée, nommée Méganode, est constituée de plusieurs Tandems, interconnectés par une batterie de cartes de commutation, une pour chaque Tandem. Chaque carte possède 4 commutateurs de liens 32 X 32, de type C004, et deux Transputers de contrôle, chacun commandant deux C004 (une direction: Nord-Sud ou Est-Ouest). La machine contient encore deux cartes de contrôle, chacune avec un Transputer, et responsables du pilotage de l'ensemble et de la communication inter modules, via une voie de contrôle additionnelle.

Il est évident que la programmation d'une fonction de contrôle et de configuration de la machine hiérarchisée devient très complexe, à cause du nombre élevé de Transputers de contrôle et de commutateurs de liens.

Conclusion:

Répétons quelques points importants du Supernode:

- tous les graphes de connexion de degré 4 étant possibles, la machine peut s'adapter à chaque application;

- la configuration peut s'effectuer de façon dynamique, donc l'architecture du réseau peut se modifier pendant l'exécution d'une application, dans le but de s'adapter aux exigences de communication;
- la communication entre une fonction de contrôle et les nœuds peut être supportée par une voie indépendante des liens de chaque nœud. Rappelons que la vitesse de cette voie est environ 20 fois plus lente qu'un lien du Transputer.

Deux approches s'opposent pour la communication inter nœuds d'une application:

- une configuration statique, réalisée à l'état initial de l'exécution;
- une configuration dynamique, et partielle, effectuée à chaque demande de communication entre un ensemble de *n* nœuds, où à la limite *n* peut être égal à 2.

Les caractéristiques comparatives de deux approches sont:

- le coût total de configuration est plus élevé dans la deuxième approche;
- la première approche exige un noyau de routage de messages, pendant que la deuxième a besoin d'un protocole de communication entre le Transputer de contrôle et les nœuds, ainsi qu'un protocole d'établissement des connexions bi-point;
- dans le premier cas, la durée d'une communication est dépendante de la distance entre les deux nœuds. De plus, le routage ralentit les autres nœuds qui sont sur le chemin. Dans le deuxième cas, les deux Transputers sont toujours en connexion directe (distance 1). Par contre, dans ce cas, il faut ajouter le coût de la connexion à chaque communication, si les deux nœuds ne sont pas déjà connectés. Le coût de cette connexion est très élevé (plusieurs microsecondes), même si elle ne se réalise que de temps en temps (à chaque installation): dans le coût on doit considérer globalement la requête: soumission par les nœuds, commandes de connexion et acquittement du Transputer de contrôle vers les nœuds;
- le Transputer de contrôle est un processeur équivalent à ceux utilisés pour les nœuds, bien que de performance plus faible. Cette caractéristique le rend capable d'exécuter des fonctions spécifiques de certaines applications.

5.2. Le Placement de l'Architecture Opera sur le Supernode

La cible de l'expérience d'Opera est un Tnode de 16 Transputers (T800) de travail, avec 2 Moctets de mémoire chacun.

L'architecture du logiciel du prototype a été conçue pour que le placement soit facile, c.a.d. qu'on trouve des équivalences naturelles entre les processus et les processeurs. Le processus est une abstraction naturelle de la notion de processeur, et les interactions entre processus sont le partage de données ou les communications.

Le placement de la TWAM:

Les processus associés à une TWAM (Calculateur, Exportateur, et Espion) ont des interactions fréquentes, correspondant à l'observation et à la modification de l'état interne d'une TWAM. Il a été choisi de les regrouper sur un processeur. Chaque nœud représente une TWAM.

Propriétés du réseau logique:

On peut distinguer dans Opera deux sous-réseaux logiques: le sous-réseau de contrôle, correspondant aux communications entre les TWAMs et l'Ordonnanceur, et le sous-réseau opératoire correspondant aux interactions inter TWAMs.

- sous-réseau de contrôle: toutes les TWAMs communiquent avec l'Ordonnanceur, ce qui constitue un réseau arborescent (N * 1). Ce réseau trouve une correspondance immédiate avec le réseau physique de contrôle d'un Tnode. Cette correspondance est un argument favorable au choix de l'architecture centralisée pour le prototype d'Opera.
- sous-réseau opératoire: deux TWAMs quelconques se communiquent afin d'installer une tâche. Ce sont des interactions bi-point au sein d'une paire de TWAMs, mais l'appariement peut être quelconque. Le réseau logique de communication est un réseau de liaisons bi-points complètement maillé.

Le placement de l'Ordonnanceur et du sous-réseau de contrôle:

Dans ce cadre, le placement du processus Ordonnanceur est immédiat: sur le Transputer de contrôle, l'Ordonnanceur s'exécutant en parallèle aux TWAMs. Le sous-réseau de contrôle est supporté par la voie de contrôle.

Ce choix permet d'envisager une évolution simple du prototype sur une machine Méganode à plus haut degré de parallélisme. Il faut alors définir une implantation de la fonction d'ordonnancement en deux niveaux. Chaque Transputer de contrôle de premier niveau porterait un ordonnanceur de premier niveau, et le Transputer de contrôle de l'ensemble de la machine exécuterait l'ordonnanceur maître.

Le sous-réseau opératoire:

Ce sous-réseau doit être implanté par les liaisons bi-points entre les nœuds, offertes par le commutateur de liens. Sa propriété de maillage complet peut être assurée de deux manières:

- par un graphe d'interconnexion physique de degré 4 assurant l'existence d'un chemin de communication entre toutes paires de TWAMs (nœuds);
- par l'établissement à la demande (dynamique) d'une liaison directe entre deux TWAMs, affectées par une installation de tâche.

Les supports nécessaires à chaque solution, et leurs avantages, ont été discutés à la fin de la section sur le Supernode. On remarquera que ces problèmes sont ignorés des processus d'Opera, et ne sont qu'un choix d'implantation des protocoles de communication entre ces processus.

Nous avons choisi de démarrer par la seconde solution (liaison dynamique) pour l'implantation du prototype. Les raisons en sont:

- une efficacité garantie sur le Tnode;
- une factorisation du travail entre plusieurs projets ([BOT 89]);
- une implantation plus simple.

Dans le cas d'un Méganode, une expérimentation ultérieure de la première solution sera toujours possible avec un noyau de communication en développement dans le même laboratoire (LGI [BRI 89]).

5.3. L'adaptation du Modèle Opera au Supernode

Le prototype Opera, implanté sur le Tnode, se différencie du modèle Opera, présenté dans la chapitre précédent, par quelques points. Ces points concernent:

- des simplifications, dans le but de diminuer le coût de développement du prototype;
- des adaptations dues au matériel (Transputer et Supernode);
- des décisions, nécessaires à cause de points encore ouverts dans la définition du modèle Opera (installation de tâches, ordonnancement).

L'annexe E présente les traits principaux de l'émulation de la TWAM séquentielle.

5.3.1. La Gestion des Contextes Multiples

Le prototype a été implanté avec la méthode d'installation de tâches "pile de variables plus datation" (VD, cf. chapitre 4, section 4.3.3).

La compilation ne traite pas le cas des variables conditionnelles. Celles-là sont initialisées à la première référence:

- variables locales: par le *put_variable*;
- variables globales: par le *unify_variable*.

Cette simplification interdit l'optimisation de la copie incrémentale.

La compilation ne distingue pas les variables non-sûres (*put_unsafe_value*) et les variables à globaliser (*unify_local_value*), car les variables locales sont systématiquement "globalisées" (allouées dans la pile de variables). Les conséquences sont que:

- l'espace de la pile de variables locales n'est récupérée qu'au retour-arrière;
- les cellules des piles locale et globale ne contiennent que des indirections vers les pile de variables;
- la liaison entre deux variables est (toujours) réalisée par les piles de variables;
- le déréférencement ne maintient que l'adresse finale dans les piles de variables.

Une telle décision invalide les avantages de l'utilisation de deux piles de variables, et interdit la liaison inconditionnelle pour les variables conditionnelles, dans les cas où celle-ci serait possible (pas de nœuds OU entre la création et la liaison de la variable).

Par rapport au modèle Opera, les coûts de ces modifications sont:

- une consommation de mémoire plus élevée dans la pile de variables;
- un déréférencement plus coûteux à cause du sur-coût constant de l'indirection vers la pile de variables.

Installation de tâches:

L'optimisation de la copie incrémentale n'est pas appliquée, et en conséquence le contrôle des sections communes n'est pas nécessaire. Du coté de la TWAM importatrice, toutes les étapes, soit de calcul, soit de réception, sont réalisées de façon séquentielle (pas de parallélisme entre la mise à jour des liaisons et le transfert des piles).

Cohérence des données:

L'exclusion mutuelle sur la pile de points de choix de la TWAM exportatrice, dans l'accès par le Calculateur et par l'Exportateur, est résolue en employant les points suivants (outre ce qui a été décrit dans le chapitre précédent):

- les processus Calculateur et Exportateur travaillent en basse priorité, et, pendant l'exclusion mutuelle, ils n'utilisent pas d'instructions provoquant la réallocation de l'unité de calcul;
- un nouveau registre, nommé BF, qui pointe vers le point de choix le plus vieux, protège le point de choix exporté (un seul) jusqu'à l'envoi de toutes ses données à la TWAM importatrice.

Coupure:

La coupure en parallèle n'a pas été implantée. La plupart des programmes de test ont été modifiés, afin d'enlever les coupures. En général, ceci se traduit par une augmentation de la durée de l'exécution séquentielle. Par contre, aucun sur-coût n'a été introduit dans la TWAM séquentielle.

5.3.2. L'ordonnancement

Evaluation de la charge:

Deux batteries de test ont été effectuées, chacune utilisant l'une de deux heuristiques d'évaluation de charge, proposées dans le chapitre précédent: le nombre de points de choix et la date du point de choix le plus vieux. Les meilleurs résultats ont été obtenus avec la date du point de choix le plus vieux.

La fréquence d'échantillonnage de l'espion a variée durant les tests. Certaines fréquences ont produit des meilleurs résultats que les autres, mais cette étude doit être refaite avec d'autres programmes (voir chapitre 6, section 6.2).

Le paramètre TRANCHE_SURCHARGE détermine des tranches de charge des TWAM surchargées, l'espion envoyant des mesures de charge si celle-ci change de tranche. Les valeurs utilisées dans le prototype sont 10 dans le cas de l'heuristique "nombre de points de choix", et 1 dans le cas "date du point de choix le plus vieux".

Théoriquement, l'espion ne devrait se réveiller qu'au moment où la charge change de tranche surchargée. Dans le cas de la charge évaluée par nombre de points de choix, on peut estimer approximativement ce moment, à partir de la vitesse de la TWAM sur le Transputer, et d'une relation moyenne de création de nœuds par rapport aux appels (inférences). Pour une vitesse moyenne de 30 Klips (voir plus loin les résultats en séquentiel), et un pourcentage (estimé) d'un tiers de prédicats non-déterministes ([TOU 87]), 10 nœuds seraient créés chaque milliseconde. Evidemment, cette estimation ne considère qu'un régime stable de création de nœuds, alors que la TWAM crée et détruit des nœuds.

Conditions minimales du parallélisme:

Le paramètre SEUIL_SURCHARGE distingue les états "muette" et "surchargée". Les mesures ont été effectuées avec deux valeurs de SEUIL_SURCHARGE: 2 et 10, la deuxième valeur produisant de meilleurs résultats.

Régulation de charge:

Dans le cas où *n* TWAMs sont inactives et une seule est surchargée, on choisit la TWAM importatrice au hasard. Un seul point de choix est exporté.

5.3.3. Le Format des Messages inter Processus

Le message d'"initialisation", envoyé par l'Ordonnanceur à toutes les TWAMs, possède 4 octets:

- numéro d'identification de la TWAM;
- argument de mise au point;
- argument de travail initial: exécution séquentielle, but principal ou initialement muette.

Les messages "requête de tâche", "mesure de charge" et les messages d'"acquittement" et de "non-acquittement", envoyés par les TWAMs à l'Ordonnanceur, ont deux octets:

- un code de message: mesure, acquittement et non-acquittement;
- la charge: un entier.

Les messages d'"exportation" et d'"importation", envoyés par l'Ordonnanceur aux TWAMs, ont 4 octets:

- un code de fonction: exportation ou importation;
- les liens du Transputer, jusqu'à 3, disponibles pour le transfert de la tâche (3 octets).

A l'installation d'une tâche, un message de contrôle, dont la longueur est de 12 octets, est envoyé avant le transfert de chacune des 4 piles. Ces messages ont été utiles seulement pour la mise au point du prototype, mais les mesures ont été effectuées avec ce sur-coût.

5.4. Les Programmes de Test

Les programmes de test ont deux origines:

- News: des programmes de d'évaluation de performances de Prolog (séquentiel), copiés du News[1];
- PEPSys[2] : des programmes d'évaluation des performances de Prolog parallèle, utilisés dans le projet PEPSys (ECRC, [BAR 88]) et Gigalips ([LUS 88]).

Tous les programmes ont été légèrement modifiés, du fait des différences de syntaxe des prédicats prédéfinis. La principale modification concerne les opérateurs arithmétiques, qui sont implantés par des prédicats préfixés dans Opera. Par exemple, l'addition est réalisée par *add(X, Y, R)*.

Les programmes du deuxième groupe, employés dans les tests en parallèle, ont été plus fortement modifiés, à cause de l'inexistence de la coupure parallèle. En général, la version Opera parallèle est moins performante que la version originale exécutée en séquentiel (mono-processeur) dans Opera.

Les programmes sont:

- deriv (News): ce programme effectue plusieurs dérivations de formules mathématiques;
- farmer (ECRC): ce programme résout le problème "du fermier, du chou, du loup et de la chèvre". Le nombre d'inférences étant très petit (300 inférences), chaque exécution répète 100 fois la solution;
- fib (News): les 15 premiers éléments de la suite de Fibonacci sont calculés;
- hamN (News): un chemin hamiltonien est cherché dans un graphe. Le nombre d'inférences est de l'ordre de 460.000 pour un graphe de 20 nœuds et de 60 arètes.
- hamP (PEPSys): le même problème et la même solution de *hamN*, mais en nouvelle version;
- map (ECRC): le problème est le coloriage d'un petit graphe. La solution effectue 22.800 inférences;
- nrev (News): une liste de 30 éléments est inversée;
- queens (News): le problème consiste à placer *n* reines dans un échiquier *n * n*, à la condition que chaque ligne, chaque colonne et chaque diagonale ne contiennent qu'une reine. La solution de *queens* essaye toutes les lignes pour chaque colonne, mais la vérification est effectuée pour chaque placement de reine;

---

[1] Un message émis par des gens de l'ECRC.

[2] Fournis de façon gracieuse par J.C. de Kergommeaux.

- queens-1 (PEPSys): le même problème de *queens*, mais ici la solution évite les lignes et les colonnes déjà utilisées pour les reines précédentes. Pour 8 reines, 103.000 inférences sont réalisées;
- queens-2 (PEPSys): le même problème et la même solution de *queens*, mais dans une nouvelle version. Cette version effectue 233.000 inférences pour 8 reines;
- query (News): le programme consulte une base de données géographique;
- quicksort (News): l'algorithme quicksort est utilisé pour trier les 50 éléments (entiers) d'une liste;
- quickdiff (News): une autre solution au problème précédent, en utilisant la technique de programmation "différence de listes".

5.5. Les Résultats en Séquentiel

Tous les programmes précédents ont été exécutés en séquentiel sur une carte contenant un Transputer T800. Le logiciel se résume à une version séquentielle de la TWAM, les processus Ordonnanceur, Espion et Exportateur étant omis. Cette version de la TWAM ne présente pas de grandes différences par rapport à la TWAM du prototype d'Opera sur le Tnode. Elle contient déjà la pile de variables et la datation.

Trois types de mesures ont été effectués:

- T1: le code complet et les données sont placés dans la mémoire externe du Transputer (plus lente que l'interne);
- T2: la mémoire interne contient une partie du code compilé du programme Prolog;
- T3: les registres de la TWAM et une partie du code de la TWAM (routine d'unification, instructions de retour-arrière) sont placés dans la mémoire interne.

La table 5.1 présente les meilleurs résultats, les unités étant la millesecondes (ms) et le Kiloinférences par second (Klips). Ces résultats indiquent une efficacité raisonnable de l'émulation de la TWAM. Suivant Inmos, le Transputer offre 10 Mips. Dans [WAR 87b], Warren considère qu'une implantation efficace de Prolog peut atteindre 20 Klips par Mips, donc 200 Klips sur le Transputer. Cependant, d'autres tests sur le Transputer montrent qu'il très difficile d'obtenir plus de 4 Mips, dans le cas où les données et le code sont placés dans la mémoire externe. En appliquant encore la règle de Warren, le Transputer devait offrir 80 Klips. Les 34 Klips obtenus par le programme *nrev* sont satisfaisants, compte tenu des décisions simplificatrices prises dans l'implantation de la TWAM.

5.6. Les Résultats en Parallèle

Les programmes, adaptés au prototype d'Opera, ont été exécutés sur le Tnode. La table 5.2 présente leurs résultats. Les diverses colonnes par programme sont dues au nombre de processeurs (TWAMs) utilisés, lequel a varié de 1 à 16. Les deux dernières colonnes présentent les temps obtenus par le système Quintus version 2.4.2 (compilateur), considéré l'un des plus efficaces, et exécuté sur les machines Sun 3/50 (1,5 Mips) et Sun 4/60 (12,5 Mips). Les trois lignes pour chaque programme contiennent, respectivement, le temps en millesecondes, le facteur d'accélération et l'efficacité. Celle-ci est calculée en divisant le facteur par le nombre de processeurs.

Dans tous les cas, le prototype d'Opera a obtenu des accélérations effectives, c.a.d. le temps d'exécution diminue toujours en augmentant le nombre de processeurs. Les facteurs d'accélération croissent de façon presque linéaire pour les gros problèmes (*queens-1* pour 10 reines). Pour les petits problèmes (*map*, *queens-1* pour 8 reines), la dérivée du facteur d'accélération diminue rapidement, c.a.d. l'efficacité à 16 processeurs est bien plus basse qu'à 2 processeurs.

Programme	T1		T2		T3	
	Ms	Klips	Ms	Klips	Ms	Klips
deriv	209	23	143	34	143	34
fib (15)	1353	7	1285	8	813	13
hamN	35593	13	34116	14	20335	24
nrev (30) * 100	39979	19	26953	21	23123	34
queens (4)	284	24	269	25	179	38
query	155	14	148	15	104	22
quickdiff	1933	15	1872	16	1139	26
quicksort	346	17	327	18	204	29
farmer * 100	3729	8	3542	8	2112	14
map	1795	13	1693	13	1009	22
queens-1 (8)	5787	18	5440	19	3663	28

Table 5.1 - Résultats en séquentiel

5.7. Le Logiciel de Développement

Au début des travaux, le logiciel de développement pour le Transputer, existant sur le marché ou le milieu académique, était inadéquat ou inexistant pour l'implémentation du prototype d'Opera. De même, il n'existait pas de système d'exploitation pour le Supernode.

En collaboration avec l'équipe Sympa (sous la direction de T. Muntean), l'équipe Flop du laboratoire LGI, à Grenoble, a donc implanté un ensemble de logiciels de développement et de service pour le Supernode. Ce travail a été considérable et a consommé une forte partie des ressources de l'équipe Flop, soit en développement soit en mise au point. Il a été alourdi par le manque total de logiciel de démarrage et par la complexité intrinsèque du Supernode.

Parmi ces outils, il faut mentionner:

- un compilateur C avec des fonctions pour la programmation parallèle (gestion de processus, par exemple);
- un assembleur, un éditeur de liens et un programme de mise au point pour le Transputer;
- des logiciels de service permettant, depuis une station de travail frontale au Supernode, de contrôler la configuration, le chargement et l'exécution de celui-ci, et, symétriquement, permettant au Supernode d'accéder aux fichiers et à la console de la station de travail;
- un noyau de système assurant les communications entre les processus, et le pilotage du commutateur de liens.

Programme	1	2	4	8	12	16	Quintus Sun 3/50	Quintus Sun 4/60
hamP	31271	16937	8122	4320	3260	2791	27100	6000
	1	1,85	3,85	7,24	9,59	11,2	1,15	5,2
	1	0,92	0,96	0,9	0,8	0,7		
map	1568	893	515	378			1350	300
	1	1,76	3,04	4,15			1,16	5,2
	1	0,88	0,76	0,52				
queens-1 (8)	3890	2060	1112	682	592	540	4350	930
	1	1,89	3,50	5,70	6,57	7,20	0,89	4,2
	1	0,94	0,87	0,71	0,55	0,45		
queens-1 (10)	90926	45628	23029	11663	8275	6433	103300	21900
	1	1,99	3,94	7,79	10,98	14,13	0,88	4,15
	1	0,99	0,98	0,93	0,91	0,88		
queens-2 (8)	8571	4397	2380	1319	1085	933	8650	1867
	1	1,95	3,60	6,50	7,90	9,18	0,99	4,6
	1	0,97	0,9	0,81	0,66	0,57		

Table 5.2 - Les résultats sur le Tnode

5.8. Conclusions

Les résultats obtenus par le prototype parallèle, montrent la capacité du modèle Opera à augmenter l'efficacité de Prolog, sur de machines parallèles sans mémoire commune. Il ne faut cependant pas oublier que ces résultats peuvent être encore améliorés par l'inclusion de plusieurs optimisations, dont la plupart sont déjà prévues dans la définition du modèle Opera. Citons les plus importantes:

- l'emploi de deux piles de variables;
- la copie incrémentale;
- la liaison profonde pour les liaisons inconditionnelles des variables conditionnelles;
- le parallélisme du calcul (déliaison, etc...) et de la réception des piles, à l'installation de tâches, du coté de la TWAM importatrice;
- l'élimination des messages de contrôle dans le transfert des piles;
- l'exportation de plusieurs points de choix;
- l'utilisation de plus d'un lien du Transputer dans le transfert des piles;
- la coupure parallèle.

6. Chapitre 6. Conclusions

Le sujet principal de cette thèse a été l'étude du parallélisme OU multi-séquentiel en Prolog, sur des machines sans mémoire commune. Cette étude s'est concrétisée par la conception et l'implantation du modèle Opera.

6.1. Résumé

6.1.1. Prolog et Parallélisme

L'étude s'est concentrée sur l'exploitation automatique du parallélisme intrinsèque au langage Prolog. Ce parallélisme trouve ses sources dans les degrés de liberté existants pour la conception d'une stratégie de résolution, c.a.d. une stratégie de parcours de l'arbre ET/OU associé à tout programme Prolog.

Le parallélisme ET exige une synchronisation entre les sous-buts parallèles. Le parallélisme OU est plus facile à implanter que le parallélisme ET, du fait de l'absence de synchronisation. De plus, son grain est plus gros, si on considère toute une branche de l'arbre OU (clause plus continuation de la résolvante courante) comme une tâche parallèle. Cependant, le parallélisme OU est faible ou absent dans les programmes déterministes.

La combinaison de ces deux types de parallélisme, le ET et le OU, est nécessaire à un système qui se veut efficace pour tous les programmes. Il est encore difficile de concevoir et implanter efficacement un tel modèle.

6.1.2. Le Modèle OU Multi-séquentiel

La thèse porte plus précisément sur le parallélisme OU multi-séquentiel.

Ses caractéristiques sont les suivantes:

- parallélisme à gros grain: une tâche est une branche de l'arbre OU;
- un processus est une machine abstraite Prolog (MAP), séquentielle et complète;
- un processus est créé si les ressources en processeur/mémoire le permettent.

La gestion des multiples contextes est le principal problème d'implantation d'un système multi-séquentiel OU. Elle doit permettre:

- un accès parallèle efficace aux sections communes des piles;
- la coexistence des multiples liaisons conditionnelles.

Deux types de solution sont possibles: la duplication des sections communes ou le partage de ces sections.

Le partage, adopté dans la plupart des projets, exige une architecture parallèle permettant un accès efficace au support physique des données communes. Actuellement, ceci n'est assuré que dans les machines parallèles à mémoire commune. Un mécanisme spécial de liaison (profonde) est nécessaire pour les liaisons conditionnelles, ce qui ajoute des sur-coûts aux opérations de liaison, déréférencement et déliaison.

La duplication est plus coûteuse en consommation mémoire. Par contre, l'emploi d'un mécanisme de liaison n'est pas nécessaire. Dans les deux cas (duplication et partage), la gestion implique un travail initial d'installation de tâches.

Les modèles proposés dans la littérature choisissent de maintenir constant et faible l'un ou l'autre de ces sur-coûts (liaisons ou installation de tâches).

6.1.3. Gain d'Efficacité

L'exploitation d'une opportunité de parallélisme n'implique pas toujours un gain d'efficacité. En effet, ce gain n'est acquis que si deux conditions minimales de parallélisme efficace sont vérifiées:

- $TC_s > TE_s$ (CP_e);
- $TC_a > TI_s$ (CP_i)

où

- TC_a: est la durée de la tâche restante à la MAP exportatrice;
- TC_s: est la durée de la tâche exportée;
- TE_s/TI_s: sont les temps d'exportation/importation de la tâche exportée.

La régulation de charge, vérifiant en permanence les conditions minimales (CP_e et CP_i), est complexe à cause:

- de la difficulté de la prévision du temps de calcul d'une tâche;
- du nombre éventuellement grand de tâches à considérer.

Les méthodes de régulation de charge, appliquées à des systèmes multi-séquentiels OU, sont encore basées sur des heuristiques, qui parfois ne produisent pas les gains de performance espérés.

Par ailleurs, les prédicats séquentiels de Prolog exigent une implantation spéciale, qui dans le plus mauvais des cas conduit à l'interdiction du parallélisme dans les sous-arbres, qui contiennent des appels à de tels prédicats.

Cependant, les résultats des prototypes actuels montrent que le parallélisme OU multi-séquentiel produit des augmentations de performance sensibles, dans le cas des machines parallèles à mémoire commune.

6.1.4. Le Modèle Opera

L'essentiel de notre travail de thèse a porté sur la conception d'un modèle d'implantation du parallélisme OU multi-séquentiel en Prolog, pour des machines sans mémoire commune: Opera.

L'objectif d'Opera est d'exploiter automatiquement le parallélisme OU d'un programme Prolog, sans intervention du programmeur: la sémantique classique est maintenue.

Cette implantation est réaliste; elle est basée sur un compilateur de Prolog vers une machine abstraite du type WAM ([WAR 83]): la TWAM. La méthode de gestion des contextes multiples est une simple duplication des parties communes entre processeurs.

Dans ce contexte, le problème principal est l'efficacité de l'installation de tâches. Une installation efficace doit éviter:

- des synchronisations entre le calcul de la TWAM exportatrice et le transfert de piles (E1);
- une sous-utilisation de la force de calcul de la TWAM importatrice, pendant le transfert;
- le transfert de liaisons conditionnelles non-valides pour la TWAM importatrice;

- le transfert des sections communes déjà dupliquées dans les deux TWAM, exportatrice et importatrice (E2).

La copie incrémentale des piles résout la condition E2; des piles de variables pour les liaisons conditionnelles, apportant une solution à la condition E1.

De plus, l'installation de tâches doit préserver les optimisations liées à l'exécution séquentielle (WAM).

Nous avons proposées deux méthodes de gestion des contextes multiples. La première, dite "pile de variables plus datation" (VD), transfère les liaisons conditionnelles non-valides et rend difficile la copie incrémentale. La seconde méthode, dite "pile de variables et traînée plus valeur" (VV), n'a pas ces inconvénients et semble plus efficace.

La copie incrémentale exige un contrôle des portions dupliquées entre les TWAM. Ce contrôle est complexe dans une machine parallèle sans mémoire commune. Opera propose une solution simple au prix de contraintes sur:

- les choix d'ordonnancement: une TWAM exporte d'abord les points de choix les plus vieux;
- la section commune: elle est définie par le point de choix le plus vieux.

Opera effectue ce contrôle par une représentation indépendante des TWAM: on maintient le point de choix commun à deux TWAM.

Le problème des liaisons conditionnelles de la dernière branche d'une TWAM est résolu par une sauvegarde spéciale des sommets des piles de la section commune.

La cohérence de la pile de points de choix de la TWAM exportatrice (accès concurrent par le calcul et par l'exportation) est préservée d'une façon simple par des règles d'ordonnancement: exportation en bloc des points de choix contigus, à partir du point de choix le plus vieux. Ces règles sont en accord avec les règles imposées par la copie incrémentale.

La coupure, nécessaire à l'évaluation des résultats, est implantée par une méthode simple et efficace, qui inhibe le parallélisme en cas de coupures à franchir dans la branche courante.

L'ordonnancement de tâches nécessite une évaluation de la charge d'une TWAM. La charge est évaluée par deux heuristiques: nombre de points de choix et profondeur (inversement proportionnelle) du point de choix le plus vieux. Les TWAM actives sont classées en muettes et surchargées, afin d'éviter d'exporter un point de choix qui serait immédiatement nécessaire à la TWAM exportatrice (tentative de non-violation de CP_i).

Plusieurs branches de l'arbre OU sont exportées (un point de choix complet au moins), afin d'avoir une tâche exportée significative (tentative de non-violation de CP_e).

Une TWAM inactive choisit la TWAM la plus surchargée. Par contre, une TWAM surchargée choisira la TWAM inactive minimisant les transferts nécessaires.

L'ordonnancement est réalisé par un processus centralisé. Il reçoit des demandes d'importation et d'exportation, et des mesures sur l'état de chaque TWAM.

Une TWAM est composée de 3 processus:

- le Calculateur: il effectue le calcul séquentiel (Prolog) et les importations;
- l'Exportateur: il réalise les exportations;
- l'Espion: responsable de la prise de mesures.

6.1.5. Le Prototype Opera

Un prototype du modèle Opera a été implanté sur le Supernode, une machine sans mémoire commune. Le Supernode offre:

- la réconfiguration dynamique par des commutateurs de liens;
- des processeurs de contrôle centralisés/hiérarchisés, responsables par la commutation des liens;
- des graphes de connexion de degré 4;
- des voies de contrôle pour la communication entre les processeurs de contrôle et les nœuds.

Sur une petite configuration (le Tnode à 16 processeurs), les processus du logiciel Opera sont placées d'une façon simple:

- l'ordonnanceur sur le processeur de contrôle;
- chaque TWAM sur un nœud.

Le prototype est une implantation partielle du modèle Opera. La gestion des contextes multiples est réalisée selon une variante simple de la méthode pile de variables plus datation:

- pas de copie incrémentale;
- pas de traitement spécial pour les variables conditionnelles;
- pas de parallélisme entre mise à jour des liaisons et réception de piles.

Des tests élémentaires ont été effectués; leurs résultats montrent:

- une bonne efficacité de la TWAM séquentielle;
- des gains de performance en parallèle identiques à ceux des prototypes implantés sur des machines à mémoire commune, mais pour des problèmes de plus grande taille (*queens-10*).

L'implantation de ce prototype a été une opération très longue, du fait du manque de logiciel de développement pour le Supernode. De gros efforts ont été consacrés à la réalisation de logiciels de développement et en mise au point du prototype.

6.2. Le Problème de la Mémoire et de la Complétude

Dans cette thèse, le problème de base a été l'accroissement de la performance en rapidité. Cependant, le parallélisme peut être utilisée pour augmenter la performance en taille. La plupart des implantations séquentielles actuelles de Prolog ne permettent pas de résoudre des problèmes de même taille que les langages impératifs. La capacité totale en mémoire d'une machine parallèle, principalement de la classe sans mémoire commune, étant supérieure à celle des machines séquentielles, il est envisageable d'utiliser le parallélisme pour augmenter la limite en taille d'une application en Prolog. Dans le cas de machines sans mémoire commune, la mémoire locale de chaque processeur tend à être plus petite que la mémoire unique d'une machine séquentielle. Donc dans ces machines la taille maximale du problème est égale ou plus petite que la taille dans des machines séquentielles, si chaque MAP n'accède qu'à la mémoire d'un seul processeur. Dans le but de supprimer cette limitation on peut imaginer des solutions du type:

- une mémoire virtuelle (globale) distribuée, ce qui dépasse le pur problème de l'implantation de Prolog;
- une division du programme Prolog: certaines MAP sont spécialisées dans l'exécution de certaines parties du programme, ce qui implique le transfert d'arguments et de résultats en échappant au modèle multi-séquentiel de base;
- une méthode qui permet l'utilisation de la mémoire d'une autre MAP (inactive ou dont la mémoire est peu utilisée), par la MAP qui a épuisé sa mémoire, ce qui exige des accès non-locaux.

En résumé, la solution à cette limitation est complexe et peut avoir un coût très élevé.

Le problème de la complétude en Prolog a été présenté dans un chapitre précédent (voir chapitre 2, section 2.3.1). Le parallélisme OU peut encore être imaginé comme une solution à ce problème, en profitant de la plus grande capacité en mémoire et processeur d'une machine parallèle. Cependant, dans la plupart des modèles de parallélisme OU, à partir du moment où toutes les MAP prennent une branche infinie, aucune solution ne sera produite. Cette situation survient certainement, si le nombre de branches infinies est supérieur ou égal à *N*, *N* étant le nombre de MAP. Par rapport à la complétude, le seul avantage de ces modèles est la possibilité de trouver un nombre plus grand de solutions que l'exécution séquentielle.

Dans Opera, des solutions à ces problèmes n'ont pas été envisagées.

6.3. Opera: Critiques et Perspectives

Il est possible d'améliorer le prototype actuel d'Opera, en implantant le modèle Opera complet:

- copie incrémentale;
- deux piles de variables;
- traitement spécial pour les variables conditionnelles;
- parallélisme entre mise à jour des liaisons et réception de piles, du coté de la TWAM importatrice;
- la coupure en parallèle.

Cependant, la conception d'une régulation de charge nécessite un ensemble de mesures significatives des comportements parallèles de programmes Prolog "réels" comme:

- *CHAT80*: un programme de grande taille, comportant un analyseur de requêtes (*parser*) en langue naturelle et des recherches dans une base de données géographiques ([CAL 89]);
- *TiNA*: un programme qui calcule des circuits touristiques ([CHA 89a]);
- *pundit*: un système de langue naturelle, développé à Unisys Paoli Research Center ([ALI 90]);
- *semigroup* et *satchmo*: des programmes pour la démonstration automatique des théorèmes ([ALI 90]);
- *proTwam*: le compilateur Prolog d'Opera.

Les mesures intéressantes concernent:

- taille des piles transférées;
- fréquence des échecs;
- oisiveté des TWAM (en attente d'une tâche à importer);
- temps de calcul des tâches;
- vitesse de création de points de choix;
- taille d'un point de choix (nombre de branches);
- temps d'installation de tâches, ceci comprenant le temps d'exportation, d'importation et de mise à jour des liaisons.

Une analyse de ces mesures permettrait d'établir:

- les relations entre les temps de calcul des tâches et les temps d'installation de tâches;
- la fréquence des exportations qui violent les conditions minimales par rapport à la taille des piles, à la date de la tâche, ...;
- la variation des gains de performance par rapport à la variation des paramètres du système Opera, par exemple le SEUIL_SURCHARGE;

- les performances de la TWAM séquentielle et de la TWAM parallèle sur mono-processeur (sur-coût dû à la gestion des contextes multiples).

Du point de vue du modèle Opera, il faut effectuer des études et concevoir:

- une méthode d'ordonnancement plus proche des études analytiques (cf. le chapitre 3, section 3.6), c.a.d. moins heuristique. Ceci inclut, par exemple, l'évaluation partielle de la charge à la compilation, en unité temps;
- une méthode d'implantation des prédicats séquentiels à effet de bord comme les entrées/sorties et les prédicats relatifs aux ensembles (*bagof*, *setof*, *oneof*, ...);
- une architecture de logiciel adaptée aux configurations hiérarchisées du Supernode (Méganode). Ceci affecte principalement la structure en processus de l'Ordonnanceur, le protocole de communication et le maintien d'un état global approché (sections communes, ...).

Les points suivants doivent au moins être évalués d'une façon approfondie:

- l'introduction de directives optionnelles qui permettent au programmeur de contrôler le parallélisme;
- l'implantation de la coupure par une méthode "activer/tuer" les branches de l'arbre OU soumises à une coupure, en utilisant des poids différents pour ces branches dans l'ordonnancement (tâches spéculatives);
- une solution particulière au problème de la limitation de la taille de l'application par la taille de la mémoire d'une TWAM (cf. section 6.2).

Enfin, le parallélisme ET doit compléter Opera pour fournir un système Prolog parallèle complet. On peut chercher à pallier le manque de complétude de Prolog (cf. section 6.2).

A. Annexe A
Opera: le Langage

Cette annexe décrit les différences principales entre le langage Prolog, implanté dans le prototype Opera, et C-Prolog. Ces différences sont en général liées à des problèmes d'implantation d'Opera, et non à des besoins du modèle parallèle.

A.1. Modules

Opera ne comprend qu'un compilateur plus un éditeur de liens, à la différence d'un environnement Prolog classique. L'introduction de modules en Opera permet la compilation de gros programmes et en particulier du compilateur Opera Prolog.

Trois types de déclarations sont relatives à la notion de modules:

- module: déclare le nom du module;
- extrn: définit les prédicats importés d'autres modules;
- entry: définit les prédicats exportés vers d'autres modules.

A chaque module correspond un fichier. La syntaxe de chaque déclaration est la suivante:

```
:- module(Nom_du_module)
:- extrn(Nom/Arité_du_prédicat)
:- entry(Nom/Arité_du_prédicat)
```

Nom_du_module est un atome et *Nom/Arité_du_prédicat* a la forme normale *nom/arité*, par exemple:

```
:- module(analyseur).
:- extrn(append/3).
:- entry(peephole/2).
```

A.2. Les Prédicats Prédéfinis

Cette section présente les prédicats prédéfinis existants dans le prototype Opera.

Expressions logiques:

X; Y	X ou Y
true/0	Vrai.
=/2	Unification de X et Y (X = Y).
X, Y	X et Y

Prédicats de contrôle:

!/0	Coupure.

fail/0	Faux (échec).
repeat/0	Boucle.
\+/1	Négation par échec.
not/1	Idem.
P → Q; R	Si P est vrai, alors Q, sinon R.
P → Q	Si P est vrai, alors Q, sinon échec.

Prédicats méta-logiques:

var/1	X (var(X)) est une variable libre.
nonvar/1	X (nonvar(X)) n'est pas une variable libre.
atom/1	X (atom(X)) est un atome.
integer/1	X (integer(X)) est un entier.
list/1	X (list(X)) est une liste.
structure/1	X (structure(X)) est une structure.
atomic/1	X (atomic(X)) est un atome ou un entier (pas de réels).
number/1	X (number(X)) est un entier (pas de réels).

Prédicats arithmétiques:

Ces prédicats ne vérifient pas si les arguments ont les types corrects. *X* et *Y* doivent être des entiers, et *V* doit être une variable libre.

add/3	V est X + Y, pour add(X, Y, V).
div/3	V est X / Y, pour div(X, Y, V).
mult/3	V est X * Y, pour mult(X, Y, V).
rem/3	V est le reste de X / Y, pour rem(X, Y, V).
sub/3	V est X - Y, pour sub(X, Y, V).
equal/2	X est égal à Y, pour equal(X, Y).
greater/2	X est plus grand que Y, pour greater(X, Y).
greaterorequal/2	X est plus grand ou égal à Y, pour greaterorequal(X, Y).

less/2	X est plus petit que Y, pour less(X, Y).
lessorequal/2	X est plus petit ou égal à Y, pour lessorequal(X, Y).
notequal/2	X n'est pas égal à Y, pour notequal(X, Y).
maxint/1	V est liée à l'entier maximum, pour maxint(V).
minint/1	V est liée à l'entier minimum, pour minint(V).

Prédicats d'entrée-sortie:

write/1	écrit le terme X (write(X)).
nl/0	écrit une nouvelle ligne.
time/1	V (time(V)) est lié à un entier, donné par l'horloge du Transputer.

A.3. La Coupure

La coupure implanté dans le prototype Opera a une seule sémantique (et une seule syntaxe), celle définie en [VAN 84]. La coupure dans les disjonctions possède donc une portée locale, c.a.d. elle ne coupe que les points de choix créés par la disjonction.

B. Annexe B
Un Résumé de la TWAM et de la Compilation

Cette annexe présente un résumé descriptif de la TWAM, la machine abstraite d'Opera. La TWAM est basée sur la Machine Abstraite de Warren (WAM), définie dans [WAR 83]. La WAM a été modifiée dans le but d'optimiser l'exécution séquentielle et parallèle sur le Transputer.

Les aspects suivants seront abordés:

- un rappel des éléments de la WAM;
- une introduction à la compilation de Prolog vers la WAM;
- les optimisations de l'exécution séquentielle.

B.1. Les Eléments de Base de la Machine Abstraite (WAM)

La WAM est constituée de:

- un ensemble de types de données (termes);
- un ensemble de registres;
- une organisation de la mémoire en piles;
- un ensemble d'instructions.

Les caractéristiques principales du fonctionnement de la WAM sont:

- l'unification est optimisée à partir de la connaissance des termes de la tête de la clause: une instruction spécialisée pour chaque type de terme (variable, constante, structure, liste);
- les arguments d'un appel (sous-but) sont chargés dans des registres spéciaux, nommés arguments, et non dans la pile comme il est usuel. Ils seront sauvegardés plus tard, si ceci est nécessaire lors d'un autre appel, ou pour la mise à jour lors d'un retour-arrière;
- un deuxième type de structure, allouée dans la pile, est introduit pour contrôler le non-déterminisme: nœud OU ou point de choix.

B.1.1. Les Termes de la WAM

La WAM supporte les types de termes suivants de Prolog:

- variable: une variable est soit libre, soit liée à un autre terme;
- constante: une constante est un entier relatif, un atome (identificateur) ou la liste vide ([] en Prolog);
- structure: une structure contient un foncteur plus n, $n > 1$, termes. Le foncteur est un atome et n est l'arité de la structure;
- liste: une liste est une structure d'arité 2 et le foncteur est implicitement défini. La liste d'un seul élément, par exemple [a], est représentée par cet élément et la liste vide. La liste de m éléments, $m > 2$, est représentée par le premier élément et la liste des $m-1$ éléments. La liste vide est un atome pré-défini (NIL ou []).

Tous les termes sont représentés par un mot qui contient un couple:

- étiquette: l'étiquette indique le type du terme;

- valeur: la valeur peut être la valeur effective du terme, une référence à la valeur effective du terme ou la description du codage de la valeur effective, qui suit ce mot.

L'étiquette de liste permet l'omission du foncteur de liste.

B.1.2. L'organisation de la Mémoire

La mémoire de la WAM comprend les zones suivantes:

- zone de code: cette zone contient le code du programme;
- pile globale: cette pile contient les structures et les listes construites pendant l'exécution du programme, de même que les variables globales et certaines variables nommées non-sûres (ce point sera traité dans la suite, section B.3.4);
- pile locale: cette pile contient deux types de structures de données:
  - environnement: un environnement contient les variables locales (nommées Y_i) d'une clause en exécution et les sauvegardes du point de retour à la clause appelante (c.a.d. le prochain sous-but de la résolvante actuelle) et de l'environnement de la clause appelante;
  - nœud OU: un nœud OU, ou point de choix, contient la sauvegarde de l'état de la WAM: les registres de piles et du code, et les registres arguments (décrits en B.1.3);
- pile traînée: cette pile contient les enregistrements des liaisons conditionnelles. Elle est nécessaire au retour-arrière pour défaire ces liaisons.

B.1.3. Les Registres

La WAM possède trois types de registres: code, piles et arguments. Les registres principaux sont:

Code:

- P: compteur ordinal;
- CP: point de retour: sauvegarde de P à l'appel d'un sous-but.

Pile locale:

- E: environnement de la clause courante;
- B: dernier point de choix;
- A: sommet de la pile locale, c.a.d. le plus grand entre E et B.

Pile globale:

- G: sommet de la pile globale;
- S: pointeur sur l'élément courant d'un terme structuré en unification.

Pile traînée:

- T: sommet de la pile traînée.

Arguments:

- X_1 à X_n: ces registres contiennent soit les arguments de l'appel (sous-but), soit les variables temporaires de la clause.

B.1.4. Les Instructions

Les instructions de la WAM peuvent être classées de la façon suivante:

- indexation: ces instructions sélectionnent une clause ou un sous-ensemble de clauses du même prédicat, selon le type ou la valeur du premier argument.
- contrôle de l'avancement: ces instructions sont utilisées dans l'appel de clauses, dans l'allocation et la libération des environnements et dans le retour à la clause appelante;
- contrôle du retour-arrière: ces instructions réalisent la création et la destruction des points de choix, et restaurent l'état de la machine en cas de retour-arrière après un succès ou un échec;
- lecture de registres X_i: ces instructions lisent le contenu d'un registre X_i et l'unifient avec un terme spécifié par l'instruction. Dans les cas de termes structurés, une copie du même terme est construite dans la pile globale, si l'argument est une variable. La construction est réalisée en mode **écriture** (*write* en anglais), par opposition au mode **lecture** (*read*) si l'argument X_i est un terme structuré. Le format général est:

  get_terme(X_i)

- écriture de registres X_i: ces instructions affectent le registre X_i à un terme construit dans la pile globale. Le format général est:

  put_terme(X_i)

- éléments de termes structurés: ces instructions soit unifient le champ courant (registre S) d'un terme structuré avec le terme décrit par l'instruction (mode lecture), soit copient ce terme dans le champ courant du terme structuré (mode écriture). Le format général est:

  unify_terme

Il existe un *get* et un *put* différent pour chaque type de terme; ceci est aussi le cas pour les *unify* (à l'exception des termes structurés).

B.2. La Compilation vers la WAM

La compilation de Prolog vers la WAM est effectuée en deux phases:

- prédicats: des instructions de contrôle de retour-arrière et d'indexation sont engendrées, en considérant toutes les clauses du prédicat (procédure);
- clauses: chaque clause est compilée de façon indépendante.

B.2.1. La Compilation de Prédicats

La compilation de prédicats n'utilise que des instructions de contrôle du retour-arrière et d'indexation (nommées instructions de prédicat dans la suite).

Une instruction de contrôle du retour-arrière est produite au début de chaque clause, s'il y a plus d'une clause. Ceci est nécessaire pour tous les prédicats à cause des appels dont le premier argument est une variable: toutes les clauses doivent être essayées. Par exemple, un prédicat *p* de la forme

$p_1(\ldots)$:- ...
$p_2(\ldots)$:- ...
...
$p_n(\ldots)$:- ...

produit le squelette de code suivant

```
p:
  try_me_else(p_2)
  code_de_la_clause_p1
p_2:
  retry_me_else(p_3)
  code_de_la_clause_p2
p_3:
...
pn:
  trust_me_else_fail
  code_de_la_clause_pn
```

Des instructions d'indexation sont produites après un examen du type du premier terme de la tête de toutes les clauses. Les types discriminants sont: variable, constante (atome), liste et structure.

D'abord, une instruction d'indexation sur le type, dite de premier niveau, est ajoutée au début de la procédure, si, pour certains types du premier argument (registre X_1), il est avantageux d'avoir des procédés différents du procédé précédent (essai de toutes les clauses). Cet avantage est déterminé par certaines combinaisons des types du premier terme des clauses. Par exemple, si toutes les clauses sont du type liste (au niveau clause on se réfère toujours au type du premier terme) et si le premier argument est de type constante, on peut échouer tout de suite.

L'instruction d'indexation sur le type (*switch_on_term*) teste le type du registre X_1 et transfère le contrôle vers:

- soit la première clause (son instruction de contrôle de retour-arrière), si le type du registre X_1 est variable;
- soit le groupe de clauses dont le type du premier argument est celui du registre X_1, ou le type variable;
- soit une routine d'échec, si aucune clause ne possède le type du registre X_1, ni le type variable.

Par exemple, pour un prédicat *p* qui est défini par les clauses

```
p1(abc)      :- ...      % 1° terme est une constante
p2(fon(X))   :- ...      % 1° terme est une structure
p3([X|Y]     :- ...      % 1° terme est une liste
```

on produit le code

```
p:
  switch_on_term(var, constant, liste, structure)
var:
  try_me_else(p_2)
constant:
  code_de_la_clause_p1
p_2:
  retry_me_else(p_3)
structure:
  code_de_la_clause_p2
p_3:
  trust_me_else_fail
liste:
  code_de_la_clause_p3
```

Ensuite, les groupes de clauses de même type sont traitées. Un groupe de clauses dit de type T (constante, structure, liste) comprend:

- toutes les clauses dont le premier terme de tête est variable;
- toutes les clauses dont ce premier terme est de type T.

Ces groupes peuvent être constitués par:

- une seule clause: aucune autre instruction de prédicat n'est produite;
- deux clauses ou plus: des instructions de prédicat sont ajoutées, en fonction du type et de la contiguïté de clauses (ordre de déclaration).

Le cas des listes est le plus simple: toutes les clauses du groupe sont reliées par une suite d'instructions de retour-arrière engendrées à part. L'indexation de premier niveau pointe vers cette suite.

Par exemple, si les clauses suivantes définissent un prédicat *p*

```
p1([a,b])    :- ...
p2(X)        :- ...
p3(1)        :- ...
p4([c,d])    :- ...
```

le code produit est (compilation similaire pour les clauses de type constante)

```
p:
  switch_on_term(bloc_var, bloc_constante, bloc_liste, var2)
bloc_liste:
  try(liste1)
  retry(var2)
  trust(liste4)
bloc_constante:
  try(var2)
  trust(constante)
bloc_var:
  try_me_else(p_2)
liste1:
  code_de_la_clause_p1
p_2:
  retry_me_else(p_3)
var2:
  code_de_la_clause_p2
p_3:
  retry_me_else(p_4)
constante:
  code_de_la_clause_p3
p_4:
  trust_me_else_fail
liste4:
  code_de_la_clause_p4
```

Pour les constantes et les structures, une instruction d'indexation par la valeur (indexation de deuxième niveau) est engendrée pour chaque sous-ensemble de clauses contiguës, si le nombre de clauses du sous-ensemble est supérieur à 1. La contiguïté est définie par la non-existence de clauses de type variable ou de répétition de la constante dans cette suite. Ces sous-ensembles, les sous-ensembles unitaires et les clauses de type variable sont reliées par une suite d'instructions de retour-arrière, cette suite étant engendrée à part. Dans le cas de structures, la valeur est le foncteur, arité incluse. Par exemple, un prédicat de la forme

p_1(a) :- ...
p_2(b) :- ...
p_3(b) :- ...

produit le code

```
p:
  switch_on_term(var, constante, fail, fail)
constante:
  try(groupe1)
  trust(const3)
groupe1:
  switch_on_constant(a:const1, b:const2)
var:
  try_me_else(p_2)
const1:
  code_de_la_clause_p1
p_2:
  retry_me_else(p_3)
const2:
  code_de_la_clause_p2
p_3:
  trust_me_else_fail
const3:
  code_de_la_clause_p3
```

B.2.2. La Compilation de Clauses

Une instruction d'allocation d'environnement, dont la taille est dépendante du nombre de variables locales de la clause, est produite au début du code de la clause.

Une instruction *get* est engendrée pour chaque terme de la tête, chacune étant suivie par des *unify* de sous-termes, si le terme est une structure ou une liste.

Chaque sous-but est compilé de la façon suivante:

- une instruction *put* pour chaque argument, chacune étant suivie par des *unify*, si le terme est structuré;
- une instruction d'appel de procédure.

A la fin de la clause, il est ajouté une instruction pour libérer l'environnement et une autre pour retourner à la clause appelante.

B.2.3. Résumé de l'Exécution d'un Prédicat

Appel:

Certains arguments sont créés et initialisés juste avant l'appel (structures, listes et variables), si le sous-but contient leur première occurrence. Les références aux arguments sont chargées dans les registres X_1 à X_n, *n* étant l'arité du prédicat (*n* peut être 0). Le contrôle est transmis à la procédure, en sauvegardant le point de retour dans CP.

Indexation de premier niveau:

Le contrôle est transmis en fonction du type du registre X_1.

Retour-arrière et indexation de deuxième niveau:

Un point de choix est créé s'il y a plusieurs clauses à essayer. Une indexation par la valeur du registre X_1 est éventuellement effectuée (voir B.2.1).

Tête de la clause:

Un environnement est alloué dans la pile locale, au-dessus de la dernière structure qui est soit un environnement, soit un point de choix.

Une unification spécialisée est exécutée pour chaque terme, et pour chaque élément d'un terme structuré. Le contrôle est transmis à l'instruction de retour-arrière du prochain sous-ensemble de clauses de la procédure (ou du prédicat précédemment appelé), si une unification échoue.

A la fin des unifications de la tête, les cellules de l'environnement (Y_i), qui y apparaissent, sont déjà liées soit à un argument (élément) du sous-but, soit à un élément d'un terme structuré construit pendant l'unification, cet élément pouvant être une variable libre. Les variables, dont la première occurrence est dans une structure, sont nommées globales parce qu'elles sont soit créées comme un élément de la structure soit y sont unifiées à un élément de la structure.

Corps de la clause:

Chaque sous-but est exécuté de la même façon que l'appel du prédicat: construction de termes structurés, chargement de registres X_i et l'appel, dans l'ordre de déclaration. A la fin de la clause, en libérant l'environnement, les variables disparaissent, mais les termes structurés, construits dans la pile globale, y restent. Ceci exige que toute unification entre deux variables lie soit la variable locale à la globale, soit la locale plus jeune à la plus âgée.

Le contrôle revient à l'instruction de retour-arrière du prochain sous-ensemble de clauses, si un sous-but échoue, et si, ni celui-ci, ni les précédents, ont des alternatives suspendues. Dans ce cas, les liaisons conditionnelles effectuées après l'appel du prédicat sont défaites (à partir de la pile traînée), et l'état de la WAM est restauré, ce qui provoque une récupération d'espace dans toutes les piles.

Dernier sous-ensemble de clauses:

Au début du dernier sous-ensemble de clauses, le point de choix est détruit après la restauration de l'état de la machine.

B.3. Les Optimisations de la WAM

Cette section présente des optimisations classiques, déjà décrites dans [WAR 83].

B.3.1. Occurrences de Variables

La première occurrence d'une variable, nommée **variable**, correspond à son initialisation soit à libre (*put* ou *unify* en mode écriture), soit à un argument de la clause (*get*). Les autres occurrences, nommées **value**, correspondent soit à une unification (*get* ou *unify* en mode lecture), soit à l'écriture d'un registre (*put*) ou d'un élément (*unify* en mode écriture). Ceci permet de spécialiser encore plus les instructions *get*, *put* et *unify*.

B.3.2. Optimisation d'Appel Terminal

L'environnement de la clause peut être libéré avant l'appel du dernier sous-but, après le chargement des registres arguments avec le contenu des variables. Ceci permet l'utilisation de la récurrence pour la programmation des itérations sans risque de débordement de la mémoire, à

condition que l'appel récurrent soit le dernier. La clause appelée doit retourner directement à la clause mère, ce qui exige une instruction d'appel qui ne modifie pas le registre CP (instruction *execute*). Les variables locales disparaissent et doivent avoir été préalablement globalisées (copiées dans la pile globale), si elles peuvent être encore utilisées (variables non-sûres).

B.3.3. Allocation de Variables

La taille de l'environnement peut être réduite en allouant certaines variables, dites **temporaires**, dans les registres X_i, en opposition aux variables **permanentes**. Une variable permanente ne peut pas être allouée dans un registre X_i dès qu'elle présente deux occurrences au moins séparées par un appel de sous-but. En effet, les registres Xi ne sont pas préservés avant utilisation dans une clause.

Les variables temporaires n'ont pas besoin ni d'être sauvegardées ni d'être crées dans la pile locale, parce qu'elles sont référencées dans un seul sous-but, et parce que soit elles sont crées dans la pile globale, soit elles ont été créées par les clauses appelantes.

L'optimisation d'appel terminal peut être étendue à toutes les variables permanentes: elles sont libérées après la dernière occurrence (instruction *put* ou *unify*). Il est suffisant de les allouer dans l'ordre inverse de leur dernière occurrence, et de les globaliser lors de cette dernière occurrence, si elles pointent encore vers l'environnement courant. Une variante de l'instruction *put_value*, nommé *put_unsafe_value*, est utilisée pour ces occurrences. Elle évite le sur-coût de la vérification dynamique de déréférencement local pour toutes les occurrences.

B.3.4. Globalisation de Variables Locales

La liaison entre une variable globale et une variable locale doit être "locale vers globale". Hors l'unification générale (*get_value* et *unify_value* en mode lecture), le seul cas qui peut conduire à une liaison inverse est la première occurrence d'une variable dans un terme structuré (*unify_value*), si elle est potentiellement locale (première occurrence de la variable dans un *get_variable* ou *put_variable*) et si le mode est écriture. La première condition étant statique, l'instruction *unify_value* possède deux versions:

- unify_local_value: pour la première occurrence d'une variable locale dans une structure: en mode écriture, la liaison est "locale vers champ" avec l'enregistrement éventuel dans la traînée, si la variable déréférence vers une variable locale (libre);
- unify_value: pour les autres occurrences: en mode écriture, le champ est initialisé avec la valeur de la variable.

B.4. Un Exemple

Soit un squelette de programme:

```
q(U) :- s(U, T), p(V, T, 3, c), r(V).

p(W, f(W), Z, c)    :- ...
p(W, f(b), Z, d)    :- ...
```

Pour le sous-but *p* de la clause *q/1* et pour la première clause du prédicat *p/4*, la compilation produit le code WAM de la figure B.1, en considérant toutes les variables comme permanentes et en désignant les variables par leurs noms.

L'état des piles et des registres, après l'exécution du sous-but *p* et de la tête de la première clause de *p/1*, est présenté dans la fig. B.2, en considérant que le sous-but *s* n'a pas lié la variable *T* (libre).

On note l'action de quelques instructions:

- put_unsafe_value(T,X_2): globalisation de *T*, en créant *T'*;
- get_variable(W,X_1): liaison inconditionnelle de *W* à *V*;
- get_structure(f/1,X_2): construction de *f/1* dans la pile globale; liaison de *T'* à *f/1*;
- unify_local_value(W): liaison de *V* et *W* au premier argument de *f/1* (une variable globale), en traînant *V*;
- get_variable(Z,X_3): liaison de *Z* à 3;
- get_constant(c,X_4): vérification d'égalité entre *c* et *c* (X_4).

```
clause q, sous-but p/4:
        ...
        put_variable(V,X1)          % première occurrence de V
        put_unsafe_value(T,X2)      % dernière occurrence de T, non-sûre
        put_constant(3,X3)
        put_constant(c,X4)
        call(p/4,1)                 % argument 1: 1 variable locale (V)
        ...

prédicat p:
p:
        try_me_else(p_2)
p_1:
        allocation de l'environnement
        get_variable(W,X1)
        get_structure(f/1,X2)
        unify_local_value(W)        % 1° occurrence de W en structure
        get_variable(Z,X3)
        get_constant(c,X4)
        ...
p_2:
        trust_me_else_fail
        ...
```

Fig. B.1 - Code TWAM pour sous-but et clause

B.5. Les Optimisations d'Opera (TWAM)

Cette section décrit des modifications introduites dans la spécification de la WAM et dans la compilation de Prolog, avec pour but l'optimisation de l'exécution séquentielle. Diverses d'entre elles ont été proposées par ailleurs, dont quelques-unes dans la spécification de la machine PLM, encore une variation de la WAM. Ceci provient du fait que le compilateur Opera est une extension du compilateur développé pour la PLM ([VAN 84]).

B.5.1. Les Termes

Les constantes ont été séparées en deux types: entiers et atomes. Ceci permet d'optimiser certaines instructions, à savoir:

- les instructions *get*, *put* et *unify* pour les constantes ont été dupliquées. Cette optimisation n'est pas importante;
- l'indexation de clauses: les sous-ensembles de clauses dont le premier argument est une constante sont séparés en sous-ensembles d'entiers et d'atomes, ce qui diminue le nombre de clauses à essayer dans l'un cas ou l'autre.

B.5.2. Listes

Dans [WAR 83] les listes sont compilées de la même façon que les structures, simplement en considérant la liste comme une structure dont le foncteur est reconnu par le compilateur. Cette technique exige une *get_list*, ou une *put_list*, pour chaque cellule de liste. Les *unify* ont une seule sémantique et ne peuvent pas traiter de sous-listes.

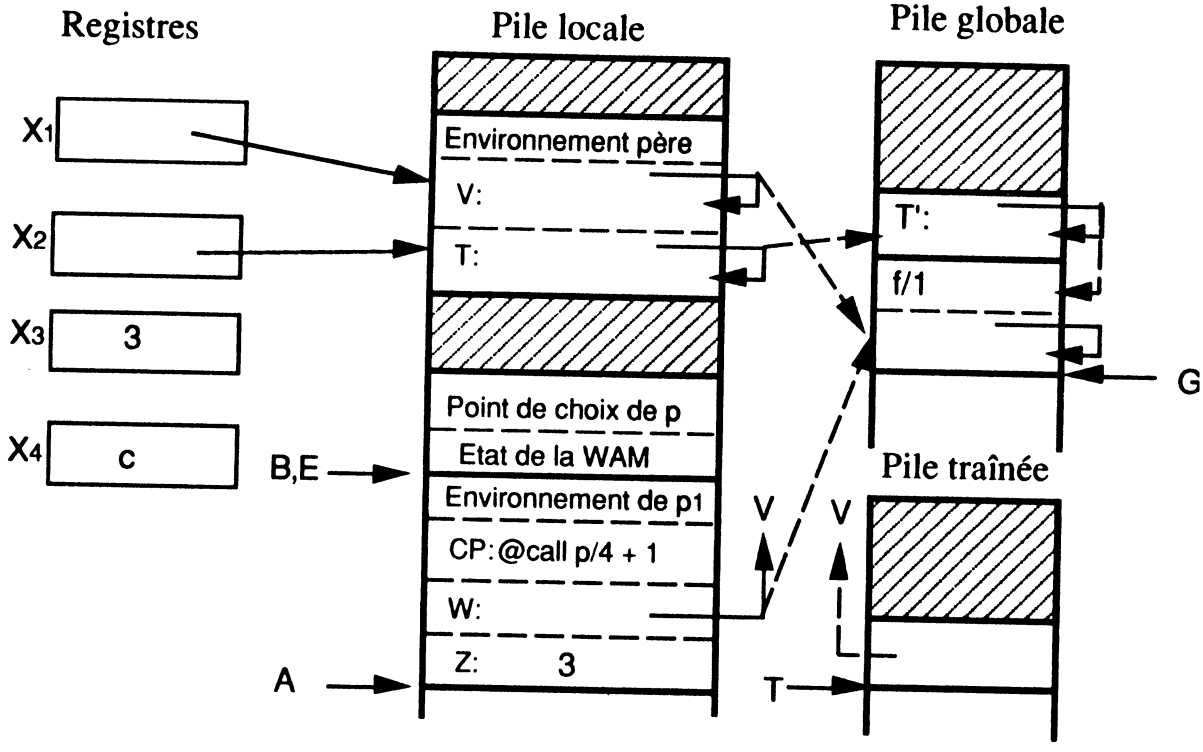

Fig. B.2 - Etat des piles et des registres

Dans la TWAM, des instructions *unify* spécialisées sont utilisées pour la compilation des éléments de listes, ce qui réduit la taille du code. Une instruction *unify_car_terme*, ou *unify_cdr_terme*, est produite pour le premier élément, ou le dernier, respectivement. Les *unify* originaux de la WAM sont utilisées dans la compilation des élément internes. Dans ce cas, elles doivent unifier le *cdr* de la liste précédente avec la nouvelle sous-liste, dont le *car* unifie avec le terme explicité dans l'instruction. Une technique similaire est utilisée par Van-Roy dans son compilateur ([VAN 84]).

B.5.3. La Disjonction

La disjonction est compilée en ligne, c.a.d. elle n'est pas transformée dans un prédicat équivalent, comme souvent pratiqué. Un point de choix spécial est créé pour chaque disjonction: les registres arguments et le point de retour n'y sont pas sauvegardés. Des variantes des instructions de retour-arrière, adaptées au point de choix spécial, ont été définies. Les variables, dont la première occurrence est dans la disjonction, sont initialisées avant la disjonction. A la fin de la disjonction, le contrôle saute vers la suite de la clause.

Par exemple, la clause

... :- ..., (a(X); b; c), d, ...

produit le code

```
...
 code_de_la_tête
 code_des_sous-buts_avant_disjonction
 init(X)                                  % 1°occurrence de X
 ltry_me_else(dis2)
 code_du_sous-but_a
 execute(cont)
disj2:
 lretry_me_else(disj3)
 code_du_sous-but_b
 execute(cont)
disj3:
 ltrust_me_else_fail
 code_du_sous-but_c
cont:
 code_du_reste_de_la_clause
```

B.5.4. Mise à Jour de l'Etat sur Echec

La mise à jour de l'état de la TWAM, au moment d'un échec, est effectuée par les instructions de retour-arrière (*retry* et *trust*), et non par la routine d'échec comme en [WAM 83].

B.5.5. La Coupure

L'implantation de la coupure n'est pas décrite dans [WAR 83]. Celle-ci doit effacer tous les points de choix créés après l'appel du prédicat qui contient la coupure. Plusieurs techniques proposent la sauvegarde du point de choix créé par le prédicat, par exemple dans l'environnement de la clause. La difficulté majeure de cette méthode provient du fait que, à cause de l'indexation de clauses, le point de choix peut ne pas avoir été créé.

La TWAM utilise une méthode similaire à celle décrite dans [ALI 87] et [CAR 87]. Au début du prédicat, avant l'indexation, l'adresse du point de choix courant (le dernier) est sauvegardée dans le registre X_{i+1}, où i est l'arité du prédicat. Le prédicat est compilé en ayant $i+1$ arguments. L'exécution de la coupure est une simple modification du contenu du registre B, à partir du registre X_{i+1}, éventuellement sauvegardé dans une variable permanente. Des registres cachés auxiliaires, par exemple les sommets de piles du point de choix courant, doivent être restaurés.

Une étude approfondie a été réalisée pour la définition d'une méthode d'implantation, valable pour diverses variantes de la coupure. Cette étude est présentée dans l'annexe C.

B.5.6. Indexation

Des nouvelles instructions ont été ajoutées pour les entiers et les structures, en mélangeant les fonctions de retour-arrière et d'indexation par valeur. Les avantages sont doubles:

- l'exécution intégrale est plus efficace: moins de branchements et de mise à jour de registres;
- la création du point de choix est différée jusqu'au moment où on arrive à une clause, dont la valeur du premier argument est égale à la valeur du registre X_1.

Par exemple, un prédicat de la forme

p_1(a) :- ...
p_2(b) :- ...
p_3(X) :- ...

$p_4(c)$:- ...
$p_5(d)$:- ...

produit l'indexation de deuxième niveau suivante (on omet la suite normale de contrôle du retour-arrière)

```
p:
  switch_on_term(var, atome, p_3c, p_3c, p_3c)
atome:
  try_switch_on_atom(a:p_1c, b:p_2c)
  retry(p_3c)
  trust_switch_on_atom(c:p_4c, d:p_5c)
var:
  try_me_else(p_2)
p_1c:
  code_de_la_clause_p1
p_2:
  retry_me_else(p_3)
p_2c:
  code_de_la_clause_p2
p_3:
  retry_me_else(p_4)
p_3c:
  code_de_la_clause_p3
p_4:
  retry_me_else(p_5)
p_4c:
  code_de_la_clause_p4
p_5:
  trust_me_else_fail
p_5c:
  code_de_la_clause_p5
```

C. Annexe C
Une Méthode d'Implantation de la Coupure en Prolog

Une méthode de compilation et d'implantation de la coupure est présentée. Cette méthode réalise efficacement les différentes formes classiques de coupure. La coexistence cohérente de ces formes est garantie à la compilation. Cette méthode peut être étendue à des formes de coupures non-classiques.

C.1. Introduction

La coupure est un opérateur de contrôle de la stratégie "en profondeur d'abord", utilisée dans l'évaluation du langage Prolog. Dans un langage du type C-Prolog, la coupure apparait sous deux formes:

- la coupure explicite;
- la coupure implicite.

a) Coupure explicite

La forme explicite se traduit par l'occurrence d'un opérateur (but) *!* dans le corps d'une clause. La sémantique opératoire de cet opérateur est claire: toutes les alternatives en attente, créées depuis l'entrée dans le prédicat, sont coupées.

La sémantique est moins claire lorsque le *!* apparait dans une disjonction au sein d'une clause:

- la famille C-Prolog traite la coupure dans une disjonction comme la coupure d'une clause (**portée globale**);
- une autre sémantique opératoire possible est de ne couper que les alternatives issues de la disjonction dans laquelle le *!* apparait [VAN 84] (**portée locale**).

L'avantage de la portée globale est de permettre l'expression de la coupure conditionnelle:

..., (but(...), ! ; true), ...

La portée locale préserve la sémantique d'un programme soumis aux opérations de pliage/dépliage. Par exemple, le sous-but

..., p(arg_p), ...

où

```
p(parm1) :- q(arg_q), !, r(arg_r)
p(parm2) :- s(arg_s).
```

devient :

```
..., (
   arg_p = renommage(parm1), q(renommage(arg_q)), !,
                r(renommage(arg_r))
   ;
   arg_p = renommage(parm2), s(renommage(arg_s))
   ), ...
```

b) Coupure implicite

Cette coupure est induite par la transformation des prédicats de test conditionnel et de négation, en des disjonctions avec coupure à portée lexicale locale (notée ici !!):

- not(but) ==> (but, !!, fail ; true)
- (test -> cas_1 ; cas_2) ==> (test, !!, cas_1 ; cas_2)

Il s'ensuit qu'une implantation Prolog, utilisant cette technique et voulant suivre C-Prolog, devra implanter ces deux formes de coupures: la coupure globale et la coupure lexicale.

Le problème se pose alors de faire cohabiter coupure locale et coupure globale sans perdre aucune des propriétés précédentes:

- équivalence par pliage/dépliage;
- coupure conditionnelle.

En effet, on voit de façon évidente que le dépliage d'une coupure globale conditionnelle ne peut se traduire par un simple remplacement du *!* par un *!!*, comme on pourrait naïvement le penser.

La résolvante

..., p(arg_p), ...

où

```
p(parm1) :- q(arg_q),!,r (arg_r)
p(parm2) :- s(arg_s).
```

est correctement transformée en:

```
..., (
arg_p = renommage(parm1), q(renommage(arg_q)), !!,
        r(renommage(arg_r))
;
arg_p = renommage(parm2), s(renommage(arg_s))
), ...
```

Mais la transformation de la même résolvante avec coupure globale conditionnelle:

```
p(parm1) :- (q(arg_q), ! ; true ), r(arg_r)
p(parm2) :- s(arg_s).
```

en

```
..., (arg_p = renommage(parm1),
            (
            q(renommage(arg_q)),!!
            ;
            true
            ),
      r(renommage(arg_r))
      ;
      arg_p = renommage(parm2), s(renommage(arg_s))
      ), ...
```

ne préserve pas l'équivalence.

De même, un problème de coexistence apparait dès que des coupures locales et globales apparaissent dans une disjonction. Par exemple, soit la clause suivante:

pred(...) :- but_1, ($test_2$, !, but_3, !!, but_4; *suitedisj*), *suitebut*.

Dans cet exemple, le *!* restaure le point de choix courant lors de l'appel à *pred*, et le *!!* restaure le point de choix courant après but_1 (avant l'appel de *test*). Ce point de choix a disparu à la suite du *!* antérieur. Les points de choix n'étant que des pointeurs dans la pile choix, nous avons une référence illégale (*dangling reference*) désignant des données non-significatives et au mieux un point de choix créé par l'exécution de but_3.

Après la présentation d'un mécanisme d'implantation efficace et unique pour ces deux formes de coupure, nous décrirons comment un compilateur peut garantir la coexistence de ces deux formes, et comment ce mécanisme peut être étendu à d'autres structures de contrôle de la stratégie.

C.2. Implantation

C.2.1. Le mécanisme de coupure de la WAM

La WAM utilisée ici est la TWAM (cf. l'annexe B). La pile locale est donc partagée en deux piles:

- une **pile locale** ne contenant que les environnements de clauses;
- une **pile de points de choix**.

Les registres d'état de la WAM sont étendus pour tenir compte de cette modification :

- A: registre sommet de la pile locale (sauvé dans tout point de choix);
- BP: registre point de reprise (cache du point de reprise du point de choix courant).

Les instructions *try_me_else*, *retry_me_else* et *trust_me_else_fail* comportent un paramètre additif donnant le nombre de registres à sauvegarder. Ces modifications sont induites par le fait que l'échec (*fail*) est un simple saut à l'adresse contenue dans BP; les opérations de restauration d'état (registres et déliaison des variables) sont exécutées par les instructions *retry_me_else* et *trust_me_else_fail*.

Par ailleurs, un point de choix restreint (point de choix local) s'ajoute au point de choix standard pour l'implantation des disjonctions [VAN 84]. Il ne comporte pas de sauvegarde des registres arguments. Les instructions correspondantes sont notées *ltry_me_else*, *lretry_me_else* et *ltrust_me_else_fail*.

Les instructions permettant l'implantation de la coupure [CAR 87] sont:

- store_B_in_X_variable <registre>
- store_B_in_Y_variable <variable locale>
- store_B_in_X_value <registre>
- store_B_in_Y_value <variable locale>

et

- load_B_from_X <registre>
- load_B_from_Y <variable locale>

Ces opérations sauvent (restaurent) B dans (depuis) un registre argument ou une variable locale.

C.2.2. La Compilation de la Coupure

Après un bref rappel sur la compilation d'un prédicat Prolog, nous présenterons la compilation d'un prédicat avec coupure. Nous décrirons séparément la compilation des deux formes de coupure, puis nous traiterons le problème de leur coexistence.

C.2.2.1. Principe de Compilation

Le principe de compilation d'un prédicat Prolog avec coupure est basée sur une transformation élémentaire destinée à le mettre sous une forme standard. Celle-ci a pour but de faire apparaître explicitement:

- la coupure comme une variable logique;
- la portée de cette coupure.

Nous ne présenterons de cette forme standard que ce qui est nécessaire à la suite de l'exposé.

C.2.2.2. Coupure Locale

La compilation est triviale. Une clause contenant une coupure locale:

... début, ($test_1$, !!, cas_1 ; $test_2$, !!, cas_2 ; testsuite_disj ...), reste ...

devient

```
... début,
    {{define_cut(VLC),
    (test1, do_cut(VLC), cas1 ; test2, do_cut(VLC), cas2 ;
suite_disj ...)
    }},
reste ...
```

où VLC est une nouvelle variable logique, dite de coupure, synthétisée pour la circonstance.

Cette clause se compile alors en:

```
    compile(... debut)
    store_B_in_Y_variable Yvlc
    ltry_me_else label1
    compile(test1)
    load_B_from_Y Yvlc
    compile(cas1)
    execute label2
label1:
    lretry_me_else label3
    compile(test2)
    load_B_from_Y Yvlc
    compile(cas2)
    execute label2
label3:
    lretry_me_else label4   ou ltrust_me_else_fail
    compile(suitedisj ...)
label2:
    compile(reste ...)
```

Remarque:

On synthétisera autant de variables de coupures qu'il y a de disjonctions coupées localement dans la clause.

C.2.2.3. Coupure Globale

La transformation du prédicat consiste à rajouter un paramètre supplémentaire (en dernier) pour synthétiser la variable logique de coupure globale et à remplacer toutes les occurrences de coupure globale (!) au sein de la clause par des *do_cut(VGC).*

Un prédicat *pred*:

pred(t_{11}, t_{12}, ...) :- but_{11}(...), !, but_{12}(...), ...
pred(t_{21}, t_{22}, ...) :- but_{21}(...), !, but_{22}(...), ...

est transformé en une forme disjonctive équivalente :

```
{{define_cut(VGC),
  (
  pred(t11, t12, ...,VGC) :- but11(...) , do_cut(VGC),
        but12(...), ...
  ;
  pred(t21, t22, ..., VGC) :- but21(...), do_cut(VGC),
        but22(...), ...
  )
}}
```

Le code produit est alors:

```
pred/arité:
   store_B_in_X_variable  Xarité+1      % correspond au
                                        % paramètre fictif
   switch_on_term X1,... <indexation en fonction des arguments>
   ...
ECl1:
   try_me_else ECl2, arité+1            % un registre de plus est sauvé
CL1:
   compile(tête)          % Xarité+1 (VGC) est affectée à un registre
                          % ou une variable locale
   compile(but11(...))
   load_B_from_X  Xvgc   ou    load_B_from_Y  Yvgc
   compile(but12(...))
ECl2:
   trust_me_else_fail arité+1
CL2:
   compile(tête)          % Xarité+1 (VGC) est affectée à un registre
                          % ou une variable locale
   compile(but21(...))
   load_B_from_X  Xvgc   ou    load_B_from_Y  Yvgc
   compile(but22(...))
   compile(suite ...)
```

Remarque:

La variable de coupure globale est unique pour une clause.

C.2.2.4. Coexistence de la Double Sémantique de la Coupure

Le problème de la coexistence de la double sémantique de la coupure se pose des points de vue de:

- l'équivalence par pliage/dépliage;
- le contrôle de la consistance d'une variable de coupure.

Le premier problème est résolu dès que les opérations de pliage/dépliage travaillent sur des prédicats mis sous forme disjonctive, c.a.d. explicitant les variables de coupure et les imbrications de portées.

Le second problème nécessite de garantir que, lors d'une coupure, la variable de coupure considérée désigne bien un point de choix existant. Le problème de coexistence apparait donc dès que des coupures locales et globales apparaissent dans une disjonction.

Par exemple, soit la clause suivante :

pred(...) :- but_1, ($test_2$, !, but_3 , !!, but_4 ; suitedisj), suitebut.

Dans cet exemple, le *!* restaure le point de choix courant lors de l'appel à *pred*, et le *!!* restaure le point de choix courant après but_1 (avant appel de *test*). Ce point de choix a disparu à la suite du *!* antérieur. Les points de choix n'étant que des pointeurs dans la pile choix, nous avons une référence illégale (*dangling reference*) designant des données non significatives et au mieux un point de choix crée par l'exécution de but_3.

Une solution possible serait d'interdire une telle situation, c.a.d. vérifier à la compilation qu'aucune coupure locale ne suit une coupure globale. Une meilleure solution est d'être capable de produire le code restaurant un état consistant de la WAM.

Pour ce faire, il suffit simplement de substituer à la valeur contenue dans la variable de coupure locale une valeur consistante de coupure. Cette valeur ne peut être connue au moment de la coupure locale mais elle est heureusement définie par la valeur de B restaurée au moment de la coupure globale.

Comme il est possible de connaître, de façon statique à la compilation, les coupures locales contrôlées par une coupure globale (du fait de la matérialisation des portées), il est alors possible de générer après la coupure globale le code de rafraîchissement de la variable de coupure locale.

Dans l'exemple précédent:

pred(...) :- but_1, ($test_2$, !, but_3, !!, but_4 ; suitedisj), suitebut
pred(...) :- autres ...

on compile la clause transformée :

```
{{define_cut(VGC)
  (
  pred(..., VGC) :-
        but1,
        {{define_cut (VLC),
        (test2, do_cut(VGC), refresh_cut(VLC), but3,
              do_cut(VLC), but4 ; suitedisj)
        }} ,
        suitebut
  ;
  pred(...,VGC) :- autres ...
```

```
)
}}
```

L'instruction *refresh_cut* se compile en un *store_B_in_X_value* ou *store_B_in_Y_value*.

Remarque:

Comme une coupure globale peut contrôler plusieurs coupures locales dans le cas de disjonctions imbriquées, il peut être nécessaire, après une coupure globale, de rafraîchir plusieurs variables de coupures locales.

C.3. Extensions

On trouve dans la littérature des structures de contrôle basées sur des formes particulières d'élagage de l'arbre OU. Parmi celles-ci, nous montrerons comment le mécanisme précédent peut être utilisé ou étendu pour prendre en compte le **snip** et la **coupure faible**.

C.3.1. Le Snip (Arity_Prolog)

En Arity-Prolog, la portée du retour arrière peut être contrôlée par le Snip ([ARI 88]).

$$p :- but_1, [! but_2, but_3, but_4 !] , suitbut \ldots$$

Si on échoue dans le corps du snip ([! but_2, but_3, but_4 !]), le retour arrière procède normalement, c.a.d. localement au snip si l'un des buts antérieurs du snip a créé un point de choix ou sur un point de choix antérieur au snip dans le cas contraire. Si le snip est franchi (c.à.d. a réussi une fois), un retour-arrière du à un échec postérieur (dans *suitbut* ... par exemple) ignore les points de choix créés par le snip.

La compilation d'une telle structure de contrôle est triviale:

```
compile(avant_snip, [! corps_snip !], apres_snip) ==>
   compile (avant_snip),
   {{define_cut(Varcut), compile(corps_snip), do_cut(Varcut),
}}
   compile (apres_snip)
```

Remarque:

Le traitement des conflits entre la coupure du snip et un *!* apparaissant dans le snip est identique au traitement du conflit entre *!* et *!!*.

C.3.2. La Coupure Faible

C.3.2.1. Objectif

L'objectif de la coupure faible (†) est de pouvoir faire le "ou exclusif" d'un ensemble de solutions produites par un prédicat.

La sémantique opératoire de Prolog sans coupure permet d'exprimer le calcul d'un ensemble de termes (solutions d'un prédicat):

$$pred(\ldots) :- but_1(\ldots)$$
$$pred(\ldots) :- but_2(\ldots)$$
$$pred(\ldots) :- \ldots$$

avec

sol(pred(...)) == sol(but$_1$(...)) union sol(but$_2$(...)) union sol(...)

L'effet de la coupure dans un prédicat est le suivant:

pred(...) :- test$_1$(...), !, but$_1$(...)
pred(...) :- test$_2$(...), !, but$_2$(...)
pred(...) :- ...

L'ensemble des solutions est donné par:

sol(pred(...)) == si sol(test$_1$(...)) ≠ {}
alors first(sol(test$_1$(...))) * sol(but$_1$(...))
sinon si sol(test$_2$(...)) ≠ {}
alors first(sol(test$_2$(...))) * sol(but$_2$(...))
sinon si sol(...)

Le but de la coupure faible est de pouvoir construire un prédicat fournissant un résultat de la forme:

sol(pred(...)) == si sol(test$_1$(...)) ≠ {}
alors sol(test$_1$(...)) * sol(but$_1$(...))
sinon si sol(test$_2$(...)) ≠ {}
alors sol(test$_2$(...)) * sol(but$_2$(...))
sinon sol(...)

Ceci s'exprime simplement à l'aide d'une coupure faible:

pred$_1$(...) :- test$_1$(...) , †, but$_1$(...)
pred$_2$(...) :- test$_2$(...) , †, but$_2$(...)
pred$_3$(...) :-...

C.3.2.2. Réalisation

La réalisation de cette coupure nécessite l'invalidation d'un point de choix précis dans la pile des points de choix: le point de choix du prédicat concerné par le †. La difficulté provient du fait que ce point de choix n'existe pas nécessairement (effet du choix de clause par indexation, cas de la dernière clause).

Pour ce faire, nous introduisons la notion de **tranche de coupure**, définie comme la suite des points de choix compris entre un point de choix origine et un point de choix final telle que:

- le point de choix origine n'appartient pas à la tranche;
- la tranche est vide si l'origine et la fin sont le même point de choix.

La transformation élémentaire de programme, base de la compilation du ou exclusif est similaire à celle du *!*. Pour le prédicat faiblement coupé, on produit

```
{{define_cut(VGC)
  (
  pred1(..., VGC) :- ) :- {{define_slice(VGS, VGC), test1(...),
                           prune_slice(VGS), but1(...)}}
  ;
  pred2(..., VGC) :- ) :- {{define_slice(VGS, VGC), test2(...),
                           prune_slice(VGS), but2(...)}}
  ;
  pred3(..., VGC):- ):- ...
```

```
    )
}}
```

ce qui se compile en:

```
pred/arité:
   store_B_in_X_variable X_arité+1
   switch_on_term X1, ...
   < indexateur >
   ...
ECl1:
   try_me_else ECl2, arité+1
CL1:
   compile(tête1)                  % Xarité+1 (VGC) est affectée à un
                                   % registre ou une variable locale
   put_structure slice/2, Xvgs
   unify_X_value     Xvgc      ou     unify_Y_value
Yvgc
   unify_B_value
   get_Y_variable  Yvgs, Xvgs      % dans le cas où la tranche est
                                   % permanente
   compile(test1(...))
   prune_slice_choice_in_X  Xvgs
                ou
   prune_slice_choice_in_Y  Yvgs
   compile(but1(...))
ECl2:
   retry_me_else ECl3, arité+1
CL2:
   compile(tête2)          % Xarité+1 (VGC) est affectée à un registre
                           % ou une variable locale
   put_structure slice/2, Xvgs
   unify_X_value     Xvgc      ou     unify_Y_value
Yvgc
   unify_B_value
   get_Y_variable  Yvgs, Xvgs      % dans le cas où la tranche est
                                   % permanente
   compile(test2(...))
   prune_slice_choice_in_X  Xvgs
          ou
   prune_slice_choice_in_Y  Yvgs
   compile(but2(...))
ECl3:
   trust_me_else_fail arité+1
Cl3:
   compile(...)
```

L'instruction *prune_slice_in_X* (ou *Y*) invalide tous les points de choix de la tranche à l'exclusion du premier. Dans le cas particulier de la coupure faible, la tranche contient zéro ou un point de choix à invalider, selon que le prédicat a créé ou non un point de choix ou que ce point de choix a disparu (exécution de la dernière clause).

Techniquement, l'invalidation peut se réaliser de différentes façons, les plus simples étant:

- le marquage de tous les points de choix de la tranche;
- le déchaînage: il suffit de déchaîner le point de choix final de la tranche.

Cette dernière solution est très facile à mettre en œuvre dès que les points de choix sont séparés des environnements de clauses.

Par contre, la récupération de la place occupée par les points de choix est plus délicate, et doit être envisagée comme un cas spécifique du problème général d'un ramasse-miette pour Prolog ([BEK 86]).

C.3.2.3. Coupure Faible Locale/Globale

Dans le cas où le *†* apparait dans une disjonction, un problème de portée identique à celui du *!* se pose. Il peut être résolu de façon similaire, en distinguant une coupure faible globale (†) et une coupure faible locale (††).

Les raisons militant pour disposer de ces deux formes de coupure faible sont identiques aux raisons pour avoir deux coupures fortes :

- coupure faible conditionnelle;
- équivalence par pliage/dépliage.

Remarque:

La coupure faible locale permet de compiler trivialement le prédicat *defaut/2* de Prolog II par une disjonction faiblement et localement coupée:

..., defaut(but1,but2), ... ==> ..., (but1, †† ; but2), ...

C.3.2.4. Coexistence des Coupures Fortes/Faibles

Bien qu'il n'y ait pas a priori de raison significative de mélanger dans une clause ou prédicat des coupures faibles et fortes, il est nécessaire à la compilation de prévoir le maintien de la cohérence des variables de coupure faibles et fortes.

Le principe de ce maintien est identique à celui du maintien de la cohérence pour les variables de coupures fortes. On calcule à la compilation les dépendances entre variables de coupure et, lors d'une coupure invalidant des coupures portant sur des points de choix plus récents, on rafraîchit celles-ci.

C.4. Conclusion

On a décrit une méthode de compilation et d'implantation de la coupure en Prolog. L'implantation proposée est une implantation efficace, et la méthode de compilation proposée garantit de façon statique la correction à l'exécution des opérations de coupure.

La technique utilisée permet de prendre en compte d'autres structures de contrôle, à condition que la portée des opérations de coupure au sein d'un prédicat ou clause puissent être calculées de façon statique.

Outre le snip (Arity-Prolog), nous avons décrit une structure de contrôle permettant l'expression de l'union exclusive de solution (coupure faible) qu'il n'est pas possible de faire naturellement en Prolog.

Cette méthode n'est pas valable dans le cas de modification dynamique d'un prédicat ou d'une structure de contrôle à portée dynamique (non-lexicale).

D. Annexe D
Les Instructions de la Coupure en Parallèle

Cette annexe présente une description algorithmique des instructions de la TWAM utilisées pour l'implantation de la coupure en parallèle.

D.1. Nouvelles Instructions

inc_ncc(SaveB):

- Compilation: *inc_ncc* est engendrée au début de la clause qui contient une coupure globale. SaveB est la variable où l'adresse du point de choix précédent est sauvegardée (voir implantation de la coupure en séquentiel dans l'annexe C);
- Sémantique:

```
si BFREE = NO_BFREE
    alors BFREE ← SaveB;
si BFREE = SaveB
    alors NCC ← NCC + 1;
```

last_cut(SaveB):

- Compilation: *last_cut* est utilisée pour la dernière coupure globale, si elle se situe après la disjonction ou s'il n'existe pas de disjonction. Elle est de même utilisée pour la dernière coupure locale de chaque alternative.
- Sémantique:

```
"coupure séquentielle"
si BFREE = SaveB
    alors NCC ← NCC - 1;
            si NCC = 0
              alors BFREE ← NO_BFREE;
```

free_cut(SaveB):

- Compilation: *free_cut* est utilisée dans le cas où la disjonction contient la dernière coupure globale de la clause. Elle est engendrée soit au début de la prochaine alternative, si celle-ci existe, soit après la disjonction. Elle a l'effet de *last_cut* sans coupure.
- Sémantique:

```
si BFREE = SaveB
    alors NCC ← NCC - 1;
            si NCC = 0
              alors BFREE ← NO_BFREE;
```

try_me_else_cutted et try_cutted:

- Compilation: les instructions de retour-arrière *try_me_else* et *try*, utilisées pour la première clause doivent être remplacées par ces nouvelles variantes, si la clause contient des coupures globales.

- Sémantique:

si BFREE = NO_BFREE
alors BFREE ← B;
SNCC ← NCC;
"try_me_else ou try"

ltry_me_else_cutted:

- Compilation: cette variante de la *ltry_me_else* doit être utilisée, si la première alternative de la disjonction contient une coupure locale;
- Sémantique:

si BFREE = NO_BFREE
alors BFREE ← B;
SNCC ← 0;
NCC ← NCC + 1;
sinon si BFREE = B
alors NCC ← NCC + 1;
"ltry_me_else"

retry_me_else(Cutted), retry(Cutted), trust_me(Cutted) et trust(Cutted):

- Compilation: ces variantes des instructions de même nom doivent les remplacer, mais à partir de la première alternative qui contient des coupures globales. *Cutted* est un nouveau argument qui indique l'existence de coupures dans l'alternative;
- Sémantique:

si BFREE = NO_BFREE
alors si Cutted = TRUE
alors BFREE ← B.B;
SNCC ← 0;
sinon si BFREE = B
alors NCC ← 0;
si Cutted = TRUE
alors BFREE ← B.B;
SNCC ← 0;
sinon BFREE ← NO_BFREE;
sinon si BFREE = B.B
alors NCC ← SNCC;
si NCC = 0 et Cutted = FALSE
alors BFREE ← NO_BFREE;

lretry_me_else_cutted(Cutted) et ltrust_me_cutted(Cutted):

- Compilation: ces variantes des instructions de même nom doivent les remplacer, mais à partir de la première alternative qui contient des coupures locales. La différence par rapport aux instructions précédentes est l'augmentation de NCC, ceci étant possible parce que, la compilation des disjonctions n'utilisant pas l'indexation, la création du point de choix est inconditionnelle;

- Sémantique:

```
si BFREE = NO_BFREE
   alors si Cutted = TRUE
            alors BFREE ← B.B;
                  SNCC ← 0;
                  NCC ← NCC + 1;
   sinon si BFREE = B
            alors NCC ← 0;
                  si Cutted = TRUE
                    alors BFREE ← B.B;
                          SNCC ← 0;
                          NCC ← NCC + 1;
                    sinon BFREE ← NO_BFREE;
                  sinon si BFREE = B.B
                          alors NCC ← SNCC;
                                si NCC = 0 et Cutted = FALSE
                                  alors BFREE ← NO_BFREE;
```

try_switch_integer_cutted, try_switch_atom_cutted, try_switch_structure_cutted:

- Compilation: ces variantes des instructions *try_switch* sont utilisées, si au moins une clause du sous-ensemble respectif de clauses contient une coupure globale. La valeur de *Cutted*, référencée ci-dessous, est un argument de la clause choisie par la fonction de *switch*.
- Sémantique:

```
"switch"
si (Cutted = TRUE)
   alors si BFREE = NO_BFREE
            BFREE ← B;
            SNCC ← NCC;
"try"
```

D.2. Instructions Modifiées

retry_me_else, retry, trust_me et trust:

- Sémantique:

```
si BFREE = B
   alors NCC ← 0;
         BFREE ← NO_BFREE;
"retry ou trust"
```

E. Annexe E
L'émulation de la Machine Abstraite TWAM

Cette annexe présente les traits principaux de l'émulation de la machine abstraite TWAM, utilisée dans le prototype Opera (cf. chapitre 4, sections 4.2.2.2 et 4.2.2.3, et l'annexe B).

E.1. Les Langages et les Logiciels de Développement

La version séquentielle de la TWAM a été programmée en assembleur Transputer. Plus tard, quelques modules ont été réécrits en langage C. Les logiciels utilisés sont les suivants:

- ADS: un assembleur Transputer ([EUD 87a]);
- M4: le macro-processeur de Kernighan et Ritchie;
- LADS: un éditeur de liens pour le Transputer ([EUD 87b]);
- PCC: un compilateur C pour le Transputer ([SAN 89]).

E.2. Les Modules

Le code source se constitue des modules suivants (principaux):

- **twamIns**: définition des macros pour la traduction du code TWAM d'un programme, produit par le compilateur Prolog, en code assembleur Transputer (une macro pour chaque instruction TWAM);
- **rnt**: module *run-time*, c'est à dire, des routines appelées, dans *twamIns*, par les instructions plus complexes. Ceci inclut l'unification et le traitement des points de choix;
- **init**: module d'initialisation de la TWAM (en général, chargement des registres);
- **tabrnt**: tables des pointeurs des routines des autres modules (*rnt*, *builtin*, ...);
- **builtin**: prédicats pré-définis (*built-in*), par exemple *var/1* et *write/1*;
- **io**: routines pour l'entrée/sortie des données TWAM (termes Prolog);
- **const**: routines pour le traitement de la table de symboles;
- **debug**: routines pour les exceptions et pour la mise au point.

E.3. La Structure de la Mémoire

La mémoire est organisée de la façon suivante (cf. figure E.1):

- **zone de code**: cette zone contient le code de l'application. Elle a une taille variable et sa position est définie à l'édition de liens;
- **table de symboles**: cette zone est réservée à la table de symboles, la position étant définie à l'édition de liens. Elle comprend trois tables de taille variable:
  - table de prédicats;
  - table d'atomes;
  - table de chaînes de caractères (*strings*).
- **registres**: des mots contigus de la mémoire du Transputer sont alloués aux registres de la TWAM. Cette allocation est effectuée par le module *init*, la position initiale étant définie par un paramètre du système;
- **piles**: toutes les piles sont contiguës et allouées par *init*. Les piles sont groupées deux à deux, la taille de chaque groupe étant un paramètre du

système. En plus des piles définies dans l'annexe B, l'émulation utilise une pile *push-down-list* (PDL) pour les routines d'unification;
- **modules**: la position des modules est déterminée à l'édition de liens.

L'émulation n'utilise que les adresses négatives de la mémoire du Transputer.

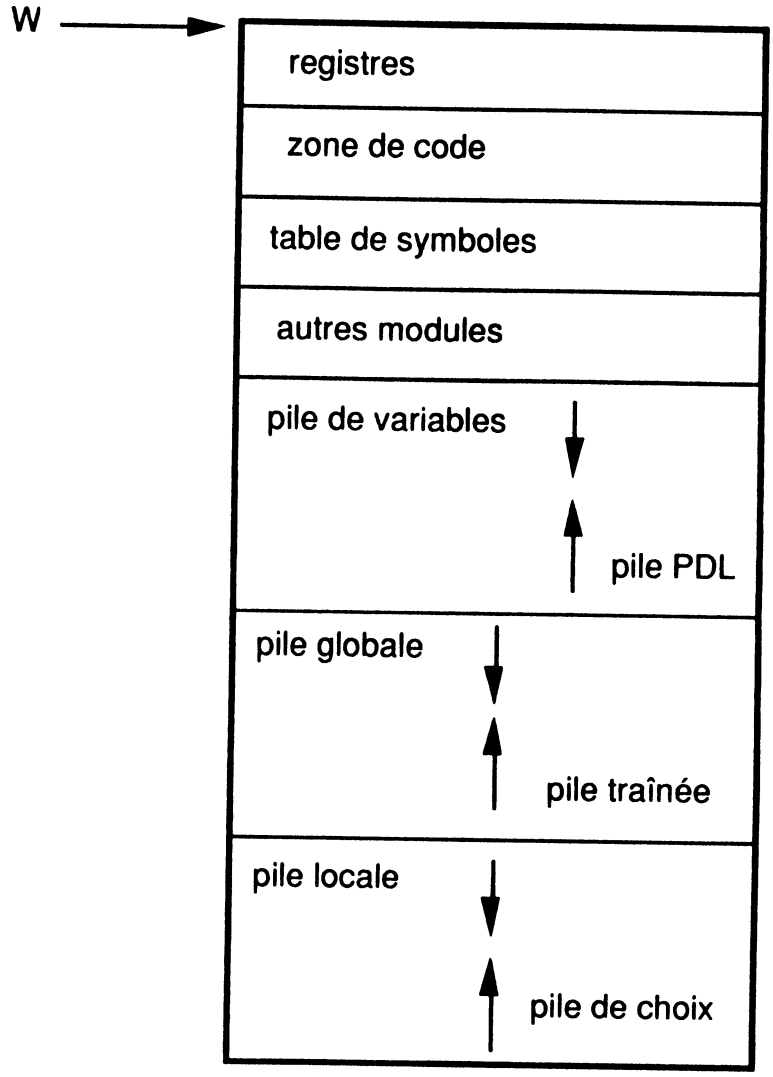

Fig. E.1 - Organisation de la mémoire de l'émulation de la TWAM

E.4. Les Registres

Des mots (32 bits) de mémoire sont alloués aux registres. Ces registres sont adressés par des déplacements relatifs au registre "pointeur vers la zone de travail" (W - *workspace*) du Transputer. En plus des registres définis dans la WAM ([WAR 83]), l'émulation de la TWAM comprend d'autres registres pour:

- la sauvegarde des pointeurs vers les tables des pointeurs des routines (module *tabrnt*);
- le transfert des arguments des routines internes, quand ceci est nécessaire. Le Transputer a seulement trois registres de travail (en pile) et nous avons évité l'utilisation de la PDL;
- la sauvegarde temporaire des certaines données, pour des routines plus complexes;

- la sauvegarde de constantes fréquemment utilisées dans le but d'augmenter la performance (un chargement plus rapide dans les registres de travail du Transputer).

E.5. Les Données

En plus des données définies dans [WAR 83] et dans l'annexe B, l'émulation emploie deux autres structures:

- **backpoint**: sauvegarde des points de choix, effectuée par les instructions d'implantation de la coupure (cf. l'annexe C);
- **atomrnt**: atome utilisé dans les modules de l'émulation, pour la mise au point (jamais unifié).

Les pointeurs n'ont pas d'étiquette et ce sont toujours des valeurs entières négatives. Ceci simplifie le test d'arrêt du déréférencement.

A cause de la difficulté de traiter sur le Transputer des données autre que des mots, la plupart des champs sont des mots. Toutes les données ont des en-têtes, dont le premier octet est l'étiquette (= type de donnée) et dont les derniers trois octets, à l'exception des variables, contiennent la taille (en octets) de la représentation. Les formats des données sont (un mot correspond à chaque étoile - *):

- **variable**:

  * en-tête: zéro (= libre, étiquette = 00) ou l'adresse (négative) d'un terme;
  * date: zéro (= libre) ou la date de liaison (CLOCK).

- **atome**:

  * en-tête: x'04tttttt' où **tttttt** est la taille en octets de la chaîne de caractères de l'atome;
  * l'index de la chaîne dans la table de chaînes de caractères;
  * le hash code pour indexer l'atome dans la table des atomes.

- **entier**:

  * en-tête: x'08000004';
  * valeur: valeur de l'entier (binaire).

- **liste**:

  * en-tête: x'0C000008';
  * car: pointeur vers le premier élément de la liste;
  * cdr: pointeur vers le reste de la liste.

- **structure**:

  * en-tête: x'10xxxxxx' ou **xxxxxx** est le nombre (*n*) d'éléments de la structure;
  * foncteur: pointeur (adresse) vers le foncteur dans la table d'atomes;
  * élément 1: pointeur vers le premier élément;
  * élément 2: pointeur vers le deuxième élément;
  * ...
  * élément n: pointeur vers l'énième élément.

- **backpoint**:

  * en-tête: x'18000008';
  * valeur: pointeur vers un point de choix (sauvegarde du sommet de la pile de points de choix - registre B);
  * clock: sauvegarde de la date (registre CLOCK).

- **atomrnt**:

  * en-tête: x'1Cxxxxx' où **xxxxxx** est la taille en octets de la chaîne de caractères de l'atome;
  * chaîne: la chaîne de caractères.

E.6. Les Instructions de la TWAM

Toutes les instructions de la TWAM ont été implantées par des macros dans le module *twamlns*. Quelques instructions, les plus simples, sont entièrement définies dans *twamlns* par du code en ligne. D'autres, comme celles qui effectuent l'unification ou qui traitent les points de choix, appellent encore des routines dans les modules *rnt* ou *builtin*. Les arguments sont transférés dans les registres de travail du Transputer et, si nécessaire, aussi dans les registres de travail de l'émulation. Ces appels utilisent les tables (module *tabrnt*) d'adresses des routines, c.a.d. l'adresse d'une routine est trouvée à partir d'un registre TWAM spécial (pointeur vers une table d'adresses) et d'un indice dans la table, qui représente le code de la routine.

Ce mécanisme d'appel est aussi utilisé pour distinguer les différents types d'unification (mode): écriture/lecture et structure/liste.

Souvent, le registre "pointeur de l'instruction courante" (P), de la TWAM, est utilisé pour la sauvegarde du point de retour. Cette sauvegarde est réalisée par la routine appelée, dans les modules *rnt*, *builtin*, ...

E.7. Le Module d'Initialisation (*init*)

Ce module est responsable par:

- l'allocation des registres de la TWAM;
- l'allocation des piles de la TWAM;
- le chargement des registres avec les valeurs initiales ou constantes.

E.8. Les Prédicats Pré-Définis (*builtin*)

Pour les prédicats pré-définis (*var/0*, *write/1*, ...), le compilateur produit un code qui se termine par:

escape(NomBuiltin)

Cette instruction engendre un appel d'une routine spécifique (une pour chaque prédicat) du module *builtin*. Tous les arguments sont transférés par les registres X_1, X_2, ...

E.9. Les Exceptions et la Mise au Point (*debug*)

Le module contient six points d'entrée pour les exceptions et la mise au point, à savoir:

- flsovfl: dépassement de capacité de la zone des piles local et de choix;
- fgsovfl: dépassement de capacité de la zone des piles globale et traînée;
- fpdlsovfl: dépassement de capacité de la zone des piles de variables et PDL;
- fnomode: mode d'unification invalide;
- fstate: impression de l'état de la TWAM et retour à la routine appelante;

- fstep: impression d'un argument et retour à la routine appelante.

Ces points ont été inclus dans une sous-table du module *tabrnt*.

E.10. L'appel des Routines

Les appels sont réalisés par une instruction de branchement (*jump*) ou par une instruction d'appel très simple (*gcall*: elle n'effectue qu'une permutation entre l'adresse de la prochaine instruction et l'adresse de la routine).

En général, la routine appelée sauvegarde l'adresse de retour dans le registre P.

Les routines récurrentes (l'unification) sauvegardent dans la PDL certains registres de la TWAM (variables locales, adresse de retour, ...).

Bibliographie:

[AIT 90] Aït-Kaci, H. The WAM: A (Real) Tutorial. PRL Research Report No.5, Digital Paris Research Laboratory, 1990.

[ALI 87] Ali, K.A.M. A Method for Implementing Cut in Parallel Execution of Prolog. In Proceedings of 4th Symposium on Logic Programming, San Francisco, pp.449-456, 1987.

[ALI 90] Ali, K.A.M. and Karlsson, R. The Muse Or-Parallel Prolog Model and its Performance. In Proceedings of North-American Conference on Logic Programming 90, Austin, pp.757-776, 1990.

[ARI 88] Arity/Corporation. The Arity/Prolog Language Reference Manual. 1988.

[BAK 78] Baker, H. Shallow Binding in Lisp 1.5. Communications of the ACM, Vol.21, No.7, 1978.

[BAR 88] Baron, U. et al. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, pp.841-850, 1988.

[BEK 86] Bekkers, Y., Canet, B., Ridoux, O. and Ungaro, L. MALI: A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages. In Proceedings of 3th International Symposium on Logic programming, Salt Lake City, pp.258-264, 1986.

[BEK 90] Bekkers, Y. Une Contrainte de Négation par l'Echec en Prolog. In Actes du Séminaire de Programmation en Logique Trégastel, Trégastel, pp.405-430, 1990.

[BOI 88] Boizumault, P. Prolog l'implantation. Masson, Paris, 1988.

[BOS 90] Bosco, P.G. et al. Logic and Functional Programming on Distributed Memory Architectures. In Proceedings of the 7th International Conference on Logic Programming, Jerusalem, pp. 325-339,1990.

[BOT 89] Botteri, T. et al. Un Système Unix-like pour une Station de Travail à Base de Transputers. In Actes des Journées Franco-Brésiliennes sur les Systèmes Répartis, Florianópolis, 1989.

[BRA 88] Brand, P. and Almgren, J. Wave-front Model for Scheduling in Or-Parallel Prolog. Internal Report, Gigalips Project, 1988.

[BRI 89] Briat, J. et al. Parx: A Parallel Operating System for Transputer Based Machine. In Proceedings of the 10th Occam User Group, 1989.

[BRI 90a] Briat, J., Favre, M. et Geyer, C. Opera: Ou Parallélisme et Régulation Adaptative en Prolog. In Actes du Séminaire de Programmation en Logique Trégastel, Trégastel, pp.329-350, 1990.

[BRI 90b] Briat, J., Favre, M., Geyer, C. et Chassin, J. Opera: a Parallel Prolog System and its Implementation on Supernode. International Workshop on Compilers for Parallel Computers, Paris, 1990.

[BRI 91] Briat, J., Favre, M., Geyer, C. et Chassin, J. Scheduling of OR-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor. In Proceedings of PARLE'91, 1991.

[BUT 86] Butler, R., Lusk, E. Olson, R. and Overbeek, R. ANLWAM - A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, Argonne, 1986.

[BUT 88] Butler, R., et al. Scheduling Or-Parallelism: An Argonne Perspective. In Proceedings of 5th International Conference/Symposium on Logic Programming, Seattle, pp.1590-1605, 1988.

[CAL 89] Calderwood, A. and Szeredi, P. Scheduling Or-parallelism in Aurora - the Mancheter Scheduler. In Proceedings of 6th International Conference on Logic Programming, Lisbon, pp.419-435, 1989.

[CAR 87] Carlsson, M. Freeze, Indexing, and Other Implementation Issues in the WAM. In Proceedings of 4th International Conference on Logic Programming, Melbourne, pp.40-58, 1987.

[CAR 88a] Carlsson, M., Danhof, K. and Overbeek, R. A Simplified Approach to the Implementation of AND-Parallelism in an OR-Parallel Environment. In Proceedings of 5th International Conference/Symposium on Logic Programming, Seattle, pp.1565-1577, 1988.

[CAR 88b] Carlsson, M., and Widén, J. Sicstus Prolog User's Manual. SICS Research Report R88007B, 1988.

[CAR 89] Carré, G. et Champion, T. Switch des Single.Node & Tandem.Node - Guide d'utilisation V1.0. Rapport Technique No.89-01, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1989.

[CHA 85] Chang, J.H. and Despain, A. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In Proceedings of The 1985 International Symposium on Logic Programming, pp.10-21, 1985.

[CHA 86] Chassin, J.K. Machines Abstraites pour l'Implantation de Prolog. Rapport de recherche No. 589, IMAG, Université Joseph Fourier, Grenoble, 1986.

[CHA 89a] Chassin, J.K. Implémentation et Evaluation d'un Système Logique Parallèle. Thèse, Université Joseph Fourier, Grenoble, Novembre de 1989.

[CHA 89b] Chassin, J.K., Codognet, P., Robert, P. et Syre, J.-C. Programmation Logique Parallèle. Technique et Science Informatique (TSI), Vol. 9, No 3 et 4, 1989.

[CLO 85] Clocksin, W.F. et Mellish, C.S. Programmer en Prolog. Editions Eyrolles, Paris, 1985.

[CLO 88] Clocksin, W.F. and Alshawi, H. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. New Generation Computing, Vol.5, pp.361-376, 1988.

[COH 88] Cohen, J. A View of the Origins and Development of Prolog. Communications of the ACM, Vol.31, pp.26-36, 1988.

[COL 72] Colmerauer, A., Kanoui, H., Pasero, R. et Roussel, P. Un Système de Communication Homme-Machine en Français. GIA, Université d'Aix-Marseille II, 1972.

[CON 85] Conery, J.S. and Kibler, D.F. AND Parallelism and Nondeterminism in Logic Programs. New Generation Computing, Vol.3, pp.43-70, Springer-Verlag, 1985.

[CON 87] Conery, J.S. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In Proceedings of the 4th Symposium on Logic Programming, San Francisco, pp.457-467, 1987.

[DEB 90] Debray, S., Lin, N. and Hermenegildo, M. Task Granularity Analysis in Logic Programs. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, pp. 174-188, 1990.

[DEG 84] DeGroot, D. Restricted And-Parallelism. In Proceedings of the International Conference on Fifth Generation Computer Systems 1984. ICOT, Tokyo, pp.471-478, 1984.

[DUN 90] Duncan, R. A Survey of Parallel Computer Architectures. Computer IEEE, february 1990.

[DWO 84] Dwork, C., Kanellakis, P. and Mitchell, J. On the Sequential Nature of Unification. Journal of Logic Programming, Vol.1, pp.35-50, 1984.

[EMD 76] van Emden, M.H. and Kowalski, R. The Semantics of Predicate Logic as a Programming Language. Journal of the ACM, Vol. 23, pp.733-742, 1976.

[EUD 87a] Eudes, J. Assembleur APT: Manuel de Référence. Document Interne, LGI, IMAG, Grenoble, 1987.

[EUD 87b] Eudes, J. Assembleur APT: User's Guide. Document Interne, LGI, IMAG, Grenoble, 1987.

[GAB 85] Gabriel, J., Lindholm, T., Lusk, E. and Overbeek, R. A Tutorial on the Warren Abstract Machine for Computational Logic. Privately circulated draft, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, 1985.

[HAR 86] Harp, J.G., Jesshope, C.R., Muntean, T. and Whitby-Strevens, C. Supernode Project P1085: Development & Application of a Low Cost High Performance Multiprocessor Machine. In Proceedings of ESPRIT'86: Results and Achievements, Bruxelles, 1986.

[HAU 88] Hausman, B. Ciepielewski, A. and Calderwood, A. Cut and Side-Effects in Or-Parallel Prolog. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, pp.831-840, 1988.

[HER 86a] Hermenegildo, M. An Abstract Machine for Restricted AND-Parallel Execution of Logic. In Proceedings of 3th International Conference on Logic Programming, London, pp.25-39, 1986.

[HER 86b] Hermenegildo, M. et Nasr, R. Efficient Management of Backtracking in AND-parallelism. In Proceedings of 3th International Conference on Logic Programming, London, pp.40-55, 1986.

[HER 86c] Hermenegildo, M. An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel. Ph.D. Thesis, University of Texas at Austin, 1986.

[HOA 78] Hoare, C.A.R. Communicating Sequential Processes. Communications of the ACM, Vol.21, pp.666-677, 1978.

[INM 89] INMOS. The Transputer Data Book. Press Ltd., Bath, 1989.

[KOW 74] Kowalski, R. Predicate Logic as a Programming Language. Information Processing 74, IFIP Congress, pp.569-574, 1974.

[KOW 79] Kowalski, R. Algorithm = Logic + Control. Communications of the ACM, Vol.22, pp.424-436, 1979.

[KOW 83] Kowalski, R. Logic Programming. In Proceedings of Information Processing, IFIP 83, pp.133-145, 1983.

[KUM 86] Kumon, K et al. Kabu-Wake: A New Parallel Inference Method and its Evaluation. In Proceedings of Computer Conference (Spring), pp.168-172, 1986.

[KUR 88] Kurozumi, T. Present Status and Plans for Research and Development. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, pp.3-15, 1988.

[LLO 88] Lloyd, J.-W. Fondements de la Programmation Logique. Editions Eyrolles, Paris, 1988.

[LUS 88] Lusk, E. et al. The Aurora Or-Parallel Prolog System. In Proceedings of the International Conference on Fifth Generation Computer Systems. ICOT, Tokyo, 1988.

[MAS 86] Masuzawa, H. et al. "Kabu-Wake" Parallel Inference Mechanism and its Evaluation. Technical Report No. TR-193, ICOT, 1985.

[MUD 89] Mudambi, S. Performance of Aurora on a Switch-Based Multiprocessor. In Proceedings of North American Conference on Logic Programming 1989, Cleveland, pp.697-712, 1989.

[MUN 89] Muntean, T. Supernode: Architecture Parallèle & Dynamiquement Réconfigurable de Transputers. In Actes d'Architectures Avancées pour l'Intelligence Artificielle, 11° Journées Francophones sur l'Informatique, Nancy, 1989.

[MYA 85] Myazaki, T., Takeuchi, A. and Chikayama, T. A Sequential Implementation of Concurrent Prolog Based on the Shalow Binding Scheme. In Proceedings of The 1985 International Symposium on Logic Programming, pp.110-118, 1985.

[NAI 86] Naish, L. Negation and Quantifiers in NU-Prolog. In Proceedings of 3th International Conference on Logic Programming, Springer-Verlag, London, pp.624-634, 1986.

[OUD 87] Oudot, O. Utilisation des Modes Directionnels dans la Résolution. Thèse, Institut National Polytechnique de Grenoble, 1987.

[PAA 88] Paaki, J. A Note on the Speed of Prolog. SIGPLAN Notices, 23(8), 1988.

[PER 88] Pereira, F. et al. C-Prolog User's Manual, Version 1.5. Dept. of Architecture, University of Edinburgh, 1988.

[REI 88] Reintjes, P.B. A VLSI Design Environment in Prolog. In Proceedings of 5th International Conference/Symposium on Logic Programming, Seattle, pp.70-81, 1988.

[ROB 65] Robinson, J. A Machine-Oriented Logic Based on Resolution Principle. Journal of the ACM, Vol.12, pp.23-41, 1965.

[ROB 88] Robert, P. 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, 1988.

[SAN 89] Santana, M. TCC: Un Compilateur C pour le Transputer. Document Interne, LGI, IMAG, Grenoble, 1989.

[SHA 89] Shapiro, E. The Family of Concurrent Logic Programming Languages. ACM Computing Surveys, Vol.21, No.3, September 1989.

[SOH 85] Sohma, Y. et al. A New Parallel Inference Mechanism Based on Sequential Processing. Tecnhical Memorandum No. TM-0131, ICOT, 1985.

[STE 86] Sterling, L. et Shapiro, E. The Art of Prolog. The MIT Press, London, 1986.

[SZE 89] Szeredi, P. Performance Analysis of the Aurora Or-Parallel Prolog System. In Proceedings of North American Conference on Logic Programming 1989, Cleveland, pp.713-732, 1989.

[TIC 84] Tick, E. and Warren, D.H.D. Towards a Pipelined Prolog Processor. In Proceedings of The1984 International Symposium on Logic Programming, Atlantic City, pp.29-42, 1984.

[TIN 87] Tinker, P. and Lindstrom, G. A Performance-oriented Design for Or-Parallel Logic Programming. In Proceedings of 4th International Conference on Logic Programming, Melbourne, pp.601-615, 1987.

[TOU 87] Touati, H. and Despain, A. An Empirical Study of the Warren Abstract Machine. In Proceedings of 4th Symposium on Logic Programming, San Francisco, pp.114-124, 1987.

[TOU 90] Touzene, A. et Plateau, B. Mesures de Performance des Communications du Meganode à 128 Transputers. Rapport Interne, Laboratoire de Génie Informatique, Institut IMAG, Grenoble, 1990.

[VAN 84] Van-Roy, P. A Prolog Compiler for the PLM. Technical Report No. UCB/CSD 84/203, University of California at Berkeley, 1984.

[VAN 86] Van Caneghem, M. L'anatomie de Prolog. InterEditions, Paris, 1986.

[WAI 90] Waille, P. Introduction à l'Architecture des Machines Supernode. Rapport Technique No.56, Laboratoire de Génie Informatique, Institut IMAG, Grenoble, 1990.

[WAR 77] Warren, D.H.D. Implementing Prolog - Compiling Predicate Logic Programs. Research Reports No. 39, 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[WAR 80] Warren, D.H.D. An Improved Prolog Implementation wich Optimises Tail Recursion. Research Paper No. 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[WAR 83] Warren, D.H.D. An Abstract Prolog Instruction Set. Tecnhical Note No. 309, Artificial Intelligence Center, SRI International, 1983.

[WAR 84] Warren, D.S. Efficient Prolog Memory Management for Flexible Control Strategies. In Proceedings of The 1984 International Symposium on Logic Programming, Atlantic City, pp.198-202, 1984.

[WAR 87a] Warren, D.H.D. Or-Parallel Execution Models of Prolog. In Proceedings of TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, pp.243-259, 1987.

[WAR 87b] Warren, D.H.D. The SRI Model for Or-Parallel Execution of Prolog -Abstract Design and Implementation Issues. In Proceedings of 4th Symposium on Logic Programming, San Francisco, pp.92-102, 1987.

[YAN 90] Yang, R. and Costa, V.S. Andorra-I: A System Integrating Dependent And-parallelism and Or-parallelism. TR-90-03, University of Bristol, 1990.

[YAS 83] Yasura, H. On the Parallel Computational Complexity of Unification. Technical Report No. TR-027, ICOT, Tokyo, 1983.

[YAS 84] Yasuhara, H. and Nitadori, K. Orbit: A Parallel Computing Model of Prolog. New Generation Computing, Vol.2, pp.277-288, 1984.

AUTORISATION DE SOUTENANCE

T6 03850

DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du 5 juillet 1984,

Vu les rapports de M ..Yves..BEKKERS.....

M ..Gilles BERGER SABBATEL.....

M RESIN-GEYER, CLAUDIO FERNANDO.....est autorisé(e)
à présenter une thèse en vue de l'obtention du ..Doctorat de.....
.Grenoble I.....

Grenoble, le .....

Le Président de l'Université
Joseph Fourier - Grenoble 1

UNIVERSITE JOSEPH FOURIER

p/o A. NEMOZ

D. CORDARY
Vice-Président