HAL
open science

Environnement de programmation parallèle: application au langage Prolog

Eric Morel

To cite this version:

Eric Morel. Environnement de programmation parallèle: application au langage Prolog. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1996. Français. NNT : . tel-00346188

HAL Id: tel-00346188

https://theses.hal.science/tel-00346188

Submitted on 11 Dec 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.

THÈSE

Présentée par

Eric MOREL

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(Arrêtés ministériels du 5 Juillet 1984 et du 30 Mars 1992)

Spécialité : INFORMATIQUE

ENVIRONNEMENT DE PROGRAMMATION PARALLELE : APPLICATION AU LANGAGE PROLOG

Date de soutenance : 14 novembre 1996

Composition du jury :

Laurent Trilling	*Président*
Patrice Boizumault	*Rapporteurs*
Philippe Codognet	
Manuel Hermenegildo	
Jacques Briat	*Examinateurs*
Jacques Voiron	

Thèse préparée au sein du Laboratoire de Modèlisation et Calculs

Résumé

Cette thèse présente l'étude de l'implantation d'un système Prolog parallèle sur une architecture sans mémoire commune dans le cadre du projet PLoSys (Parallel Logic System). L'exécution exploite le parallélisme de manière implicite. Le système repose sur un modèle OU multiséquentiel. Le partage de l'état d'exécution est assuré par copie des données. Le langage Prolog supporté est complet, et intègre les effets de bord classiques du langage. La gestion parallèle fait l'objet d'une étude complète pour préserver la compatibilité avec l'exécution séquentielle du langage Prolog. En particulier, une méthode originale est présentée pour la gestion parallèle des effets de bord.
Enfin, ce document présente la réalisation d'un prototype portable, ainsi que l'analyse des résultats obtenus.

Abstract

This thesis discribes the implementation of a parallel Prolog system on distributed memory architecture. This work is a part of the PLoSys (Parallel Logic System) project.
Parallelisation of a program is done implicitly using. The program execution uses an OR-parallel multisequential model. Data copying is used for sharing execution state among processors.
The supported language represent all Prolog predicates, including common side-effects. An original method is described to implement the parallel management of side-effects which preserves the sequential semantic of a Prolog program execution.
Finaly, this document presented a portable prototype with its evaluation, with the analysis of the first results obtained.

Remerciements

Je tiens à remercier :

- M. Laurent Trilling, Professeur à l'Université Joseph Fourier, pour m'avoir fait l'honneur de présider mon jury de soutenance.

- M. Patrice Boizumault, Professeur à l'Ecole des Mines de Nantes, M. Philippe Codognet, Chargé de recherche à l'INRIA Rocquencourt, et M. Manuel Hermenegildo, Professeur à l'Université de Madrid, pour l'intérêt qu'ils ont porté à mon travail en acceptant d'être rapporteurs de cette thèse.

- M. Jacques Briat, Maître de Conférence à l'Université Joseph Fourier, pour m'avoir fait connaître le langage Prolog ainsi que le parallélisme. Je le remercie aussi vivement pour ses conseils avisés lors de la rédaction de ce document. Enfin, je tiens aussi à le remercier pour le soutient permanent qu'il a su apporter aux membres de l'équipe du projet PLoSys.

- M. Jacques Chassin de Kergommeaux, Chargé de recherche CNRS, pour m'avoir donné l'occasion de participer à un projet Européen avec la Hongrie et pour son aide généreuse pour la publication d'articles.

- M. Salah Eddine Kannat, frais Docteur, pour son amitié et son soutien. Je le remercie encore pour toutes les heures partagées dans la bonne humeur, à celles passées devant nos écrans, aux longues discussions sur l'informatique et autre, et enfin pour les publications que nous avons partagées.

- M. Alexandre Carissimi, futur Docteur, pour son amitié et sa bonne humeur, pour l'aide qu'il a apporté à notre équipe et enfin pour sa participation active aux longues discussions sur l'informatique.

- Ma famille pour le soutien quotidien, discret mais essentiel, sans lequel je n'aurais pu mener bout ce travail de thèse.

A mes parents.

Table des matières

1. Introduction 1

1.1 La programmation parallèle 1
1.1.1 Les machines parallèles 1
1.1.2 Le parallélisme et les langages de programmation 2
1.2 Motivations et objectifs 2
1.3 Contexte de recherche 3
1.4 Structure du document 4

2. Prolog et le parallélisme 5

2.1 Le langage Prolog 5
2.2 La compilation de Prolog 7
2.2.1 La représentation des données en Prolog 7
2.2.2 Les piles de la WAM 8
2.2.3 Les instructions de la WAM 9
2.2.4 Les prédicats à effet de bord 11
2.3 Différents types de parallélisme 13
2.3.1 Le parallélisme OU 14
2.3.2 Le parallélisme ET 14
2.3.3 Le parallélisme ET/OU 14
2.3.4 Les autres formes de parallélisme 14
2.4 Les principales techniques d'implantation 16
2.4.1 Les modèles OU-parallèles 16
2.4.2 Les modèles ET-parallèles 17
2.4.3 Le problème des effets de bord 18
2.4.4 Ordonnancement du travail 19
2.5 Quelques systèmes Prolog parallèles 19
2.5.1 Architecture avec mémoire commune 19
2.5.2 Architecture sans mémoire commune 21

3. Le traitement des effets de bord 25

3.1 Sémantique séquentielle des effets de bord 25
3.1.1 La coupure 25
3.1.2 Les entrées et sorties 26
3.1.3 Les prédicats collecteurs 27
3.1.4 La base des prédicats 27
3.2 Problèmes principaux 29
3.3 Retrouver l'ordre séquentiel 30
3.3.1 Solutions classiques 30
3.3.2 Solution sans mémoire commune 31
3.4 La coupure 36
3.4.1 Détermination du travail spéculatif 36
3.4.2 Gestion dynamique de la coupure 38
3.5 Les entrées et sorties 43
3.5.1 Gestion des entrées et sorties 43

3.5.2 Ordre séquentiel et les entrées et sorties 44
3.5.3 Interaction avec la coupure 45
3.6 La base des prédicats 46
3.7 Les prédicats collecteurs 47
3.7.1 Choix de la méthode d'implantation 47
3.7.2 Principes de l'exécution parallèle 48
3.8 Influence sur la régulation de charge 49
3.8.1 La coupure 49
3.8.2 Les entrées et sorties 50
3.8.3 Les prédicats collecteurs 51
3.8.4 La base des prédicats 51
3.8.5 La suspension de travail 51
3.9 Extensions vers un système totalement distribué 52

4. Le système PLoSys 55

4.1 Objectifs et organisation 55
4.1.1 Choix du langage supporté 55
4.1.2 Machines cibles 56
4.1.3 Environnement de programmation 57
4.2 Environnement d'exécution parallèle 59
4.2.1 Choix du type de parallélisme 59
4.2.2 Principes de mise en œuvre 60
4.2.3 Architecture logicielle 63

5. Implantation du système 71

5.1 Architectures matérielles 71
5.2 Le support d'exécution parallèle 72
5.2.1 La gestion des unités 72
5.2.2 Les communications 72
5.2.3 La bibliothèque de programmation parallèle 73
5.3 Le noyau Prolog 74
5.3.1 Choix d'un noyau Prolog 74
5.3.2 Description de WAMCC 76
5.3.3 Modifications générales du noyau d'exécution WAMCC 77
5.3.4 Modifications générales lors de la compilation 77
5.4 Structure logicielle du prototype 78
5.5 Le système parallèle 79
5.5.1 La console 80
5.5.2 Ordonnanceur 81
5.5.3 Les travailleurs 83
5.6 Les unités indépendantes 85
5.6.1 L'unité de régulation de charge 85
5.6.2 La gestion de la coupure 86
5.6.3 Les entrées et sorties 88
5.6.4 Les autres effets de bord 89
5.7 Outils d'analyse des performances 89
5.7.1 Système séquentiel 89

5.7.2 Système parallèle: 90

6. Résultats 91

6.1 Caractéristiques des systèmes utilisés 91
6.1.1 IBM SP/1 91
6.1.2 Le réseau de stations ou de serveurs de terminaux 92
6.1.3 Réseau compatible PC 92
6.2 Performances séquentielles de PLoSys 92
6.2.1 Performances de WAMCC 92
6.2.2 Les programmes de test 93
6.2.3 Performances de PLoSys séquentiel 95
6.3 Exécutions parallèles 95
6.3.1 La régulation de charge 96
6.3.2 Premiers résultats 96
6.3.3 Une exécution détaillée 100
6.4 Performances et effets de bords 106

7. Conclusion et perspectives 107

7.1 Conclusion 107
7.1.1 Environnement de programmation 107
7.1.2 Exécution parallèle 107
7.1.3 Performances 107
7.1.4 Portabilité 108
7.2 Perspectives 108
7.2.1 Amélioration du système 108
7.2.2 Les extensions du système 109

Liste des figures

Figure 2-1 - Le langage Prolog 7
Figure 2-2 - Arbre ET/OU 7
Figure 2-3 - Format des données de la WAM 8
Figure 2-4 - Exemple de code WAM optimisé 10
Figure 2-5 - Les prédicats collecteurs 13
Figure 3-1 - Entrées et sorties séquentielles 27
Figure 3-2 - Evaluations comparées de *findall* et *setof* 27
Figure 3-3 - Effets de bords et architecture à mémoire commune 31
Figure 3-4 - Ordre séquentiel et parcours 32
Figure 3-5 - Ordre séquentiel lors d'une exportation 33
Figure 3-6 - Transfert du plus récent point de choix et ordre séquentiel 33
Figure 3-7 - Intervalles de temps lors des transferts 35
Figure 3-8 - Positions séquentielles des travailleurs 35
Figure 3-9 - Gestion des niveaux de coupure 37
Figure 3-10 - Instructions WAM et niveaux de coupure 38
Figure 3-11 - Optimisations de la WAM et niveaux de coupure 39
Figure 3-12 - Gestion parallèle des niveaux de coupure 40
Figure 3-13 - Validation des branches spéculatives 41
Figure 3-14 - Invalidation des branches 41
Figure 3-15 - Le renvoi de message d'invalidation ou de validation 42
Figure 3-16 - Travail spéculatif et effets de bord 43
Figure 3-17 - Découplage des entrées et sorties 44
Figure 3-18 - Liste séquentielle avec entrées et sorties 45
Figure 3-19 - Principe du maintien de l'ordre séquentiel en distribué 52
Figure 4-1 - Environnement logiciel de PLoSys 58
Figure 4-2 - Les différentes méthodes de copie de piles 61
Figure 4-3 - Politiques de transfert des points de choix 63
Figure 4-4 - Architecture de PLoSys 64
Figure 4-5 - Principe d'échange de travail 65
Figure 4-6 - Ordonnanceur 67
Figure 4-7 - Le travailleur 68
Figure 5-1 - Athapascan-0a 73
Figure 5-2 - Organisation logicielle du prototype 79
Figure 5-3 - Les tâches du système d'exécution parallèle 80
Figure 5-4 - Implantation de l'ordonnanceur 82
Figure 5-5 - Implantation du travailleur 84
Figure 5-6 - Elément de la table des points de choix exporté 87
Figure 5-7 - Elément de la table de points de choix importés ou exportés 88
Figure 6-1- Accélérations relatives pour quelques programmes 98
Figure 6-2 - Accélérations réelles pour quelques problèmes 99
Figure 6-3 - Messages de charge envoyés par chaque travailleur 101
Figure 6-4 - Transferts par travailleur pour le programme q1(12) 102
Figure 6-5 - Quantité de données transférées pour le programme q1(12) 103
Figure 6-6 - Répartition du temps d'exécution pour le programme q1(12) 104
Figure 6-7 - Efficacités moyennes mesurées pour le programme q1(12) 105

Liste des tableaux

Tableau 6-1 - Performances relatives des systèmes Prolog (temps en secondes) 93
Tableau 6-2 - Temps d'exécutions pour différents systèmes (en secondes)..... 93
Tableau 6-3 - Temps d'exécution sur Pentium 166Mhz et Linux (en secondes). 95
Tableau 6-4 - Temps d'exécution avec le SP/1 et réseau Ethernet..... 97

1. Introduction

1.1 La programmation parallèle

L'évolution des besoins informatiques a conduit les fabricants de matériel à proposer soit des processeurs spécialisés très puissants mais très coûteux, soit à associer un nombre élevé de processeurs généralistes au sein d'une même architecture. Si cette solution présente un avantage financier important pour une puissance théorique largement supérieure, elle pose de nouveaux problèmes tant matériels que logiciels.

1.1.1 Les machines parallèles

Il existe deux principales familles d'architectures, selon que leur mémoire est partagée entre tous les processeurs, ou distribuée localement à chaque processeur.

Dans la première famille, chaque processeur de la machine partage des zones de mémoire avec les autres processeurs. Si ces zones sont indifférenciées, toute la mémoire est partagée, l'accès à la mémoire est dit uniforme (UMA, Uniform Memory Access). Si l'accès à ces zones dépend de la distance entre le processeur et la mémoire, l'accès est non-uniforme (NUMA). Pour réduire la congestion des accès à la mémoire partagée, plusieurs niveaux de caches peuvent être ajoutés entre le processeur et la mémoire. Il faut alors ajouter un mécanisme matériel de contrôle de la cohérence des caches. Ces difficultés limitent le nombre de processeurs que l'on peut regrouper au sein d'une même machine. Ce nombre est actuellement de l'ordre de la centaine [Baetke95].

Dans la seconde catégorie de machines, il n'existe pas de mémoire partagée entre les différents processeurs, la mémoire est distribuée (NORMA, No Remote Memory Access). Les informations ne peuvent être échangées entre les différents processeurs, que par le biais de communications. L'architecture de ces machines est généralement très régulière, et permet leur extension assez facilement. Le nombre de processeurs ainsi regroupés peut devenir très grand, de l'ordre du millier, et est le plus souvent limité par le coût résultant.
Cette famille de machines se présente comme un réseau de stations de travail. En effet ceux-ci peuvent être vus comme un ensemble de processeurs interconnectés par un réseau de communication, donc comme des machines à mémoire distribuée.
Pour toutes ces machines, il existe plusieurs types d'exploitation des processeurs :

- Exploitation synchrone ou SIMD (Single Instruction Multiple Data). Tous les processeurs effectuent les mêmes opérations en même temps. Les données sont locales à chaque processeur, et sont souvent de taille réduite.
  L'échelle physique de ce type de machine peut être un système monocomposant, intégrant un grand nombre de processeurs élémentaires. Dans ce cas son utilisation est généralement très spécialisée, comme le traitement numérique de l'image.
- Exploitation asynchrone ou MIMD (Multiple Instruction Multiple Data). Les processeurs effectuent différentes opérations sur leurs données respectives. Dans un cas extrême, ils peuvent exécuter des programmes différents (MPMD, Multiple

Program Multiple Data). Dans cette catégorie, on peut classer les réseaux de stations de travail.

1.1.2 Le parallélisme et les langages de programmation

Les ordinateurs à architecture parallèle ont atteint une certaine maturité ces dernières années. Ils disposent actuellement des systèmes d'exploitations classiques (Unix, AIX, Solaris, ...) ce qui facilite leur utilisation courante. Mais l'utilisation du fort potentiel de calcul offert par les architectures parallèles n'est pas encore systématique, car leur programmation demeure encore un obstacle de taille pour la majorité des programmeurs.

Les langages les plus courants (C, C++) possèdent maintenant des extensions pour gérer le parallélisme, ou des librairies spécialisées pour le développement d'applications parallèles, tels que MPI[MPI93] ou PVM[Geist93], ainsi qu'un ensemble d'outils associés, pour la mise au point et l'évaluation des performances.

D'autres langages comme Fortran disposent de compilateurs adaptés à ces nouvelles machines (HP-Fortran [HPFF93]). L'efficacité de ces outils est très irrégulière selon les programmes, car il est souvent difficile d'extraire le parallélisme d'un algorithme conçu et optimisé pour le traitement séquentiel des données.

Enfin il existe des langages dont le niveau d'abstraction vis à vis du matériel est tel qu'il permet l'exploitation automatique du parallélisme d'un programme. Parmi ces langages, les langages logiques représentent de bons candidats pour la parallélisation automatique, et certains, comme Prolog, sont déjà exploités très efficacement sur certaines architectures parallèles.

Il existe aussi de nouveaux langages de programmation tels que Strand [Foster89], Charm [Charm92], PCN [Foster92], qui ne peuvent cependant prétendre détrôner les langages classiques, de par leur faible base installée et du peu de bibliothèques sources disponibles en rapport de celle des langages courants.

1.2 Motivations et objectifs

La principale motivation du projet dans lequel s'inscrit ce travail de thèse est donc de démontrer que le langage Prolog est un bon candidat à la parallélisation automatique, et un langage privilégié pour l'utilisation des machines parallèles sans mémoire commune. Ceci a déjà été montré pour les machines à mémoire commune (chapitre 2).

L'objectif principal du projet est donc l'implantation d'un système Prolog parallèle efficace et complet sur des machines sans mémoire commune. Il se décompose en plusieurs points :

- *Gestion implicite du parallélisme*

Il s'agit d'offrir le moyen au programmeur d'accroître les performances de ses logiciels sans pour autant avoir à tout réécrire. Ainsi son effort peut se concentrer sur l'utilisation efficace du parallélisme, si les performances de ces logiciels ne suffisent plus.

- *Support complet du langage et standardisation*

Pour garantir que l'exécution d'un programme soit possible sans avoir à modifier son code pour le réadapter au système. Cela permet la réutilisation des applications et librairies de programmes déjà existantes, pour peu qu'elles respectent aussi le standard.

- *Efficacité du système*

Il faut montrer que la parallélisation de Prolog apporte un réel accroissement de performances, sans compliquer la programmation, par le masquage de la structure parallèle de la machine. Il faut alors assurer que même dans le cas les plus défavorables, les performances sont au moins égales à celle d'un bon système séquentiel.

- *Portabilité*

Elle est nécessaire pour assurer une diffusion du système d'exécution avec le moins d'interventions possible sur son code pour l'adapter à un nouvel environnement.

- *Environnement de développement*

Celui-ci doit faciliter l'écriture de programme, le développement d'applications et l'optimisation des performances des logiciels créés. Il doit comporter un outil de mise au point simple, des outils de mesure de performances et permettre l'optimisation du système à l'architecture employée.

1.3 Contexte de recherche

Après une grande effervescence autour des langages de programmation logique, qui a produit de nombreux systèmes parallèles, le climat est devenu plus calme. Bien que les recherches continuent sur le mariage de Prolog et des architectures parallèles, l'accent est actuellement porté sur l'intégration de résolveur de contraintes dans le langage, et dont l'intérêt concerne de nombreux domaines, tels que l'aide à la décision, l'ordonnancement, la simulation de circuits...
Le début des années 1990 a vu la naissance de quelques systèmes Prolog parallèles complets tels que Aurora[Carlsson90] ou Muse[Karlsson92], qui tous nécessitent une architecture parallèle à mémoire commune. Or, l'évolution des architectures indique qu'une grande partie des systèmes parallèles sera à base de machines sans mémoire commune. Aussi, afin de pouvoir exploiter ce nouveau potentiel, un nouvel axe de recherche s'est ouvert.

A niveau de l'IMAG, la recherche sur Prolog parallèle a débuté avec le projet OPERA [Briat90], directement sur les machines parallèles sans mémoire commune à base de Transputer [Harp86]. Ce projet a pu montrer que, en dépit d'une programmation lourde et complexe, l'implantation de Prolog sur les premières architectures parallèles sans mémoire commune procure déjà de bons résultats [Favre92].

Les nouvelles générations de machines, se rapprochent plus encore des machines séquentielles classiques, tant au niveau du matériel, avec l'intégration des mêmes processeurs, qu'au niveau du logiciel, par l'emploi des mêmes systèmes d'exploitation. Il est aussi courant de considérer qu'un réseau de stations de travail est une architecture parallèle sans mémoire commune. Mais, si l'utilisation de ce type de machines parallèles se banalise, leur programmation reste un point délicat, et la conception d'environnements

de programmation parallèle efficaces est une nécessité. Cela relance l'intérêt d'implanter un système Prolog parallèle complet sur ce type de machines.

Le projet PLoSys

Le projet PLoSys (Parallèle Logic System) comprend deux parties indépendantes :

- La création d'un système Prolog parallèle, supportant le langage Prolog complet et standard, disposant aussi d'un environnement de développement. Ce système doit assurer une indépendance vis à vis du matériel pour une probabilité maximale.
- L'étude de la régulation dynamique de charge dans les systèmes logiques, en particulier dans le cadre d'un système Prolog sur une architecture sans mémoire commune.

Le travail au sein du projet PLoSys est logiquement divisé en deux parties distinctes, une pour l'étude de la régulation de charge, l'autre pour l'implantation d'un système Prolog.

L'étude de la régulation de charge dans un système Prolog, est bâtie sur la définition et l'emploi d'une plate-forme d'évaluation de stratégies par simulation, basée sur l'exécution parallèle de programmes Prolog synthétiques. Elle fait l'objet d'un travail de thèse dédié [Kannat96].

Ce travail de thèse s'inscrit dans le second axe et porte sur la conception d'un système Prolog parallèle devant :

⇒ être portable,

⇒ exploiter automatiquement le parallélisme implicite de Prolog,

⇒ intégrer la gestion des prédicats à effets de bord et la coupure.

1.4 Structure du document

Le chapitre 2 rappelle les bases de Prolog séquentiel et sa compilation. Suivent les principes généraux de la parallélisation automatique de l'exécution d'un programme Prolog. Quelques systèmes existants sont alors présentés.
Le chapitre 3 traite des techniques l'implantation des effets de bord du langage Prolog selon leur sémantique séquentielle. Après un rappel des problèmes posés par la parallélisation des solutions précédemment proposées, nous présentons notre solution et les mécanismes nécessaires à la gestion des différents effets de bord.
Le chapitre 4 présente les points important de la conception logicielle du prototype, comme les choix des techniques de parallélisation et l'organisation du prototype.
Le chapitre 5 décrit l'implantation de ce prototype et le chapitre 6 regroupe les premiers résultats, ainsi que leur analyse qualitative.
Le document se conclut avec le chapitre 7, qui après une rapide analyse sur le travail mené, présente les différentes perspectives liées au projet.

2. Prolog et le parallélisme

Dans ce chapitre, nous rappelons les bases et les fondements de Prolog, puis la solution la plus classique pour sa compilation séquentielle; enfin nous présentons différentes alternatives existantes pour la parallélisation de l'exécution de Prolog, ainsi que les systèmes Prolog parallèles les plus voisins de notre approche.

2.1 Le langage Prolog

Le langage Prolog est une application concrète d'un modèle de programmation logique, défini par A. Colmerauer durant les années 70 [Colmerauer72]. L'exécution d'un programme Prolog repose sur la résolution SLD [Kowalski74], avec au départ un ensemble de règles, et une requête initiale (but initial) que l'on doit résoudre à l'aide des règles.

Prolog est un langage de programmation déclaratif qui repose sur la logique des prédicats du premier ordre, la résolution SLD associée à une opération complexe, l'unification, dont le rôle peux être comparée au passage de paramètres des langages impératifs. Ce langage permet de faire abstraction de la machine en autorisant une expression plus naturelle du problème à résoudre; ainsi il n'y apparaît aucune variable typée, ni même de structure de contrôle classique, seulement un ensemble de règles (ou prédicats) transcrivant la méthode pour résoudre un problème.

Une règle comporte généralement deux parties : la tête et le corps.

- Si le corps est vide la règle est alors appelée un fait, celui-ci est toujours vrai.
- Si le corps n'est pas vide, il contient la liste des buts à vérifier pour valider la tête, ces buts sont aussi décrits par des règles. Les buts peuvent être composés par deux opérations différentes, la conjonction et la disjonction. Les différentes formes d'affectation ou de comparaison de variables sont remplacées par l'unique opération unification des termes lors de l'appel de règles.

L'ensemble des règles (ou prédicats) constitue le programme Prolog (Figure 2-1). La décomposition d'une règle en sous-buts, pour sa résolution, représente le mécanisme d'appel de prédicats.

Un programme Prolog manipule des données, appelées termes, qui sont de quatre types :

- Les variables.
- Les constantes, elles mêmes divisées en atome, entier, réel, chaîne, ...
- Les listes, pouvant contenir des variables, des constantes, ...
- Les structures, dont la notation utilisée sera le couple foncteur (un atome, considéré comme le nom de la structure), et arité (nombre d'éléments de la structure).

faits :

pere(paul,marie). « *Paul est le père de Marie.* »

pere(alain,paul).

pere(alain,jean).

règles :

grand_pere(X,Y) :- pere(X,Z) , pere(Z,Y). (conjonction ou ET)

X est le grand-père de Y, si X est le père de Z ***et*** *Z est le père de Y.*

parent(X,Y) :- pere(X,Y) ; mere(X,Y). (disjonction ou OU)

X est le parent de Y, si X est le père de Y, ***ou,*** *si X est la mère de Y.*

Une autre écriture de la disjonction est possible, à l'aide de deux clauses :

parent(X,Y) :- pere(Y,X).

parent(X,Y) :- mere(Z,X).

Dans cet exemple, le prédicat (règle) « parent » contient deux alternatives, et donc peut être résolu de deux façons différentes. Alors que le prédicat « grand-père » ne sera résolu que si les deux appels successifs au prédicat « pere » peuvent être résolus.

Figure 2-1 - Le langage Prolog

La stratégie de Prolog est fondée sur un parcours en « profondeur d'abord » de l'arbre ET/OU déterminé par l'évaluation du programme (voir Figure 2-2). En effet, chaque nœud de l'arbre correspond à un but et :

⇒ Si le but est décomposé en une disjonction de sous-buts, alors le nœud correspondant est un nœud OU. Le but sera prouvé si au moins une des disjonctions est complètement prouvée.

⇒ Si le but est décomposé en une conjonction de sous-buts, le nœud équivalent est un nœud ET. Le but ne sera prouvé que si tous les sous-buts de la conjonction sont prouvés.

⇒ Enfin une feuille de l'arbre correspond soit à la réussite de la résolvante (appel d'un prédicat amenant à un fait), soit à son échec.

Enfin, Prolog utilise un mécanisme de retour arrière (*backtracking*), pour évaluer tous les choix possibles crées par les différentes disjonctions présentes dans les prédicats (Figure

2-2). Quand une branche de l'arbre d'évaluation échoue, Prolog effectue retour en arrière vers le premier nœud OU disponible. Il essaye alors d'évaluer la première branche non évaluée (prochaine alternative du prédicat). Quand toutes les branches d'un nœud OU sont évaluées, Prolog remonte vers le nœud précédent dans l'arbre, parcourant ainsi l'arbre en profondeur d'abord. Une description plus complète de Prolog est donnée dans de nombreux livres, dont les plus connus sont [Bratko88], [Shapiro86].

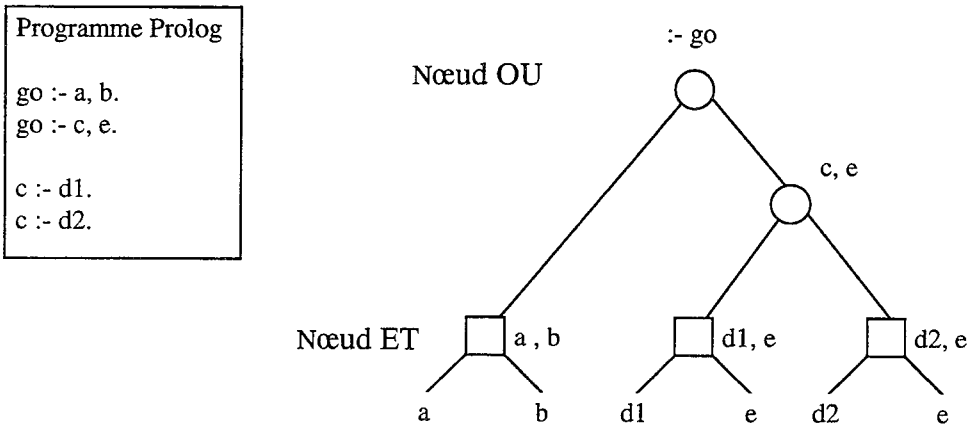


Figure 2-2 - Arbre ET/OU

Nous utiliserons comme référence pour le langage Prolog la norme définie dans [Covington93]. Dans la suite toutes les variables seront désignées par une chaîne alphanumérique commençant par une lettre majuscule (ou un soulignement), ainsi *Marie* est une variable alors que *marie* est une constante.

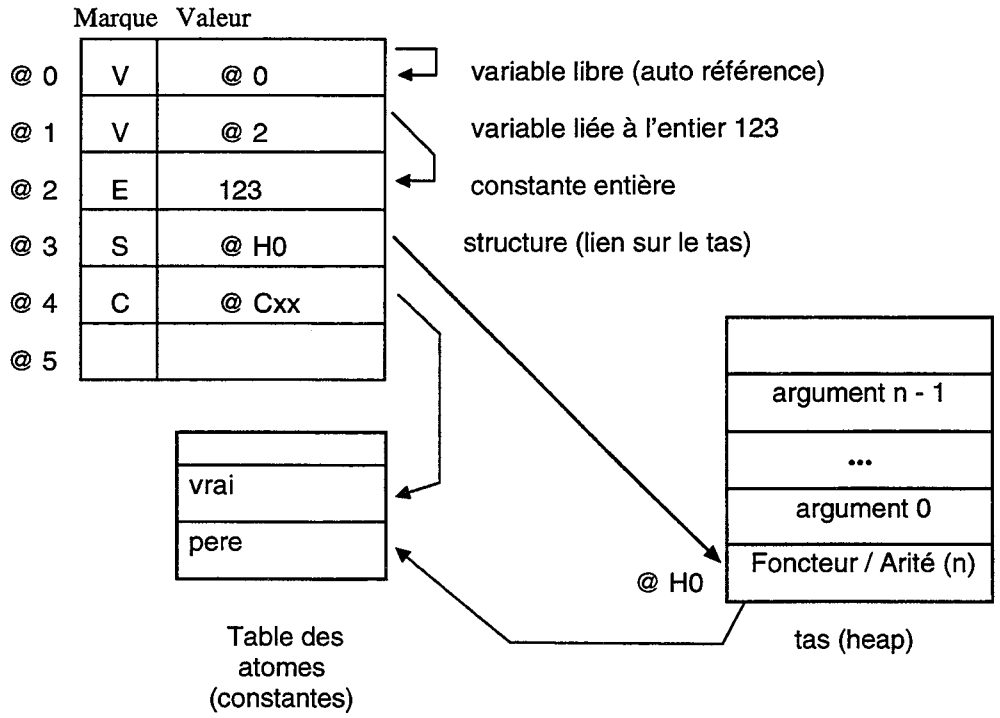
2.2 La compilation de Prolog

Une grande partie des compilateurs actuels sont basés sur la machine abstraite de Warren [Warren84]: la WAM (Warren Abstract Machine). Cette machine abstraite permet une production de code efficace, et qui peut être facilement adaptée à différents processeurs physiques. Pour faciliter la lecture de ce document, nous n'aborderons que les notions essentielles de la WAM; aussi le lecteur trouvera de plus amples détails sur l'implantation de Prolog dans [Boizumault88], et une description de la WAM dans [Warren84], ou plus « accessible » [AitKaci90]. Il existe encore de nombreuses variantes de compilation présentées pour la plupart dans [VanRoy93].

2.2.1 La représentation des données en Prolog

La représentation des objets Prolog est dite « marquée ». Chaque objet (ou terme), variable, liste, foncteur, est représenté par une cellule ou plusieurs cellules. La première contient toujours deux champs (Figure 2-3) :

- La marque, ou type, indique la nature de l'objet, telle que variable, liste, entier, constante ou foncteur.
- La valeur de l'objet, qui peut être la valeur d'une constante entière ou une référence vers un autre terme, dans ce cas c'est une liaison, ou encore une référence vers la table des atomes (constantes littérales). Une autoréférence est considérée comme la représentation d'une variable non liée. Pour un foncteur, cette valeur est un codage du couple ***nom/arité*** de la structure, ***nom*** étant une référence dans la table des atomes. Ce couple est suivi de la liste des cellules correspondant aux termes du foncteur.



Cette figure donne la réprésentation d'une variable libre, d'une autre liée à la valeur entière 123, de la constante entière 123, de la structure pere(...) à n arguments, et de la constante littérale vrai.

Figure 2-3 - Format des données de la WAM

L'affectation des variables est représentée par leur liaison à un autre terme Prolog. Les cellules ont toutes la même taille, et sont stockées dans différentes piles que nous allons présenter maintenant.

2.2.2 Les piles de la WAM

Nous n'étudierons pas en détail l'ensemble des piles gérées par la WAM. Nous nous intéresserons plus particulièrement aux piles enregistrant le parcours des nœuds ET et OU de l'arbre d'exécution du programme Prolog. Voici une brève description des piles utilisées:

Pile globale :

La pile globale contient les valeurs de termes quelconques, dont la durée de vie est celle du chemin de résolution (ou résolvante). Elle peut contenir des références sur elle-même, mais pas sur la pile locale dont la durée de vie des éléments est inférieure.

Pile locale :

Cette pile contient les valeurs des variables locales des clauses, c'est à dire les variables qui doivent être conservées d'appel de sous but en appel de sous but. Leur durée de vie est celle de l'exécution de la clause les contenant initialement. A l'appel d'un but correspond la création d'un environ (espace de la pile locale), à partir duquel sont placées les variables du but.
Cette pile contient aussi les données nécessaires au mécanisme de retour arrière (points de choix), constituées principalement par la sauvegarde de l'état de la machine ainsi que la prochaine alternative du prédicat à tester.
Cette pile contient des références à des zones d'elle-même, ou de la pile globale.

Pile de liaison ou traînée :

Dans la WAM, il existe deux types de liaisons, les liaisons conditionnelles et inconditionnelles. Une liaison conditionnelle, est la liaison d'une variable créée avant un point de choix, et liée après le point choix. Cette liaison doit être annulée si un nouvel essai de résolution est conduit depuis le point de choix, car elle n'est plus valide. S'il n'y a pas de création de point de choix entre la création de la variable et sa liaison, la liaison est dite inconditionnelle. Elle n'est jamais défaite.
La WAM utilise une pile de liaison, ou traînée, pour enregistrer les variables qui ont été liées de manière conditionnelle et qui doivent être libérées pour le prochain essai du point de choix. Ces variables seront déliées lors de la restauration du point de choix, par un simple parcours de la pile de traînée. Si la liaison a été enregistrée dans cette pile après la position du sommet de la pile lors création du point de choix, elle sera annulée, sinon elle demeure valide. La pile traînée ne contient donc que des références aux piles locale et globale.

2.2.3 Les instructions de la WAM

Les instructions qui nous intéresseront dans la suite sont essentiellement les instructions de manipulation des points de choix. L'ensemble des instructions de la WAM et leur description algorithmique sont disponibles dans [AitKaci90].
Un exemple de code WAM est donné en Figure 2-4.

Les instructions de manipulation des points de choix :

- *la création* : La création d'un point de choix est réalisée par l'instruction **try_me_else** . Dans ce point de choix, seront mémorisés les différents paramètres d'exécution, dont les arguments d'appel du prédicat, la prochaine clause à évaluer, le point de choix précédent (en cas d'échec du prédicat), les différents registres de gestion de la mémoire...
Cette instruction sera générée avant la première clause d'un prédicat.

Programme Prolog :

```
call(call(X)) :- call(X).
call(repeat).
call(repeat)  :- call(repeat).
call(true).
```

Code WAM correspondant :

```
S:  switch_on_term S1,C,fail,F   ; index suivant variable, constante, liste, foncteur
C:  switch_on_constant 2, { repeat: C1 , true: Sd } ; 2 = taille de la table de hashcode
F:  switch_on_structure 1, { call/1: S1 }

C1: try Sb                       ; essayer la clause en Sb, créer un point de choix pour la suite
    trust Sc                     ; essayer la clause en Sc et détruire le point de choix

                                 ; call(call(X)) :- call(X).
S1: try_me_else S2               ; essayer cette clause, créer un P.C. pour reprendre en S2
Sa: get_structure call/1, A1     ; prendre le premier paramètre (tête)
    unify_variable A1            ; tenter de l'unifier avec la variable d'appel (but)
    execute call/1               ; appel le prédicat call/1 (récursif)

                                 ; call(repeat).
S2: retry_me_else S3             ; prendre le P.C. crée en S1, le mettre à jour pour reprendre en S3
Sb: get_constant repeat,A1       ; lire la constante repeat
    proceed                      ; retour à l'appelant

                                 ; call(repeat) :- call(repeat).
S3: retry_me_else S4             ; prendre le P.C. crée en S1, le mettre à jour pour reprendre en S4
Sc: get_constant repeat, A1      ; lire la constante repeat
    put-constant repeat,A1       ; passer la constante repeat
    execute call/1               ; appel le prédicat call/1 (récursif)

                                 ; call(true).
S4: trust_me                     ; prendre le point de choix crée en S1, le détruire
Sd: get_constant true, A1        ; lire la constante true
    proceed                      ; retour à l'appelant
```

Figure 2-4 - Exemple de code WAM optimisé

- *la mise à jour* : Lors du retour arrière, pour évaluer la clause suivante du prédicat, il faut restaurer l'état des arguments donnés lors de l'appel du prédicat, des variables modifiées, de la mémoire. L'instruction qui met à jour les informations contenues dans le

point de choix est l'instruction **retry_me_else** . Cette instruction est générée avant toutes les clauses intermédiaires d'un prédicat, si elles existent.

- *la destruction* : Lorsque la dernière clause d'un prédicat doit être évaluée, le point de choix n'est plus utile, et donc après avoir restauré les paramètres utiles à l'évaluation de la clause, le point de choix est enlevé de la pile. La prochaine clause à évaluer en cas d'échec du prédicat sera obtenue à partir du point de choix précédent, s'il existe, sinon le programme termine. L'instruction de destruction d'un point de choix est l'instruction **trust_me** . Elle est générée pour la dernière clause d'un prédicat.

- *l'indexation* : La machine abstraite définie par D.H.D. WARREN est très efficace, en partie par l'utilisation de l'indexation des clauses. Cette indexation permet de sélectionner un groupe de clauses à partir du type et des valeurs pour le premier argument de la tête de la clause. Elle permet de limiter le nombre de clauses à essayer pour un appel de prédicat donné. Dans le meilleur cas une seule clause est sélectionnable, et il n'y a pas besoin de créer de point de choix. Ce mécanisme de sélection est réalisé par l'indexation de l'entrée vers les clauses suivant que la nature du premier argument est une variable, une constante, une liste, ou un foncteur). Un second niveau est employé ensuite, par utilisation de la valeur du premier argument. Enfin, un troisième niveau est possible si plusieurs clauses correspondent à la même valeur (pas de variable), ce niveau n'est pas une indexation mais plutôt une limitation de l'espace d'essais à un sous ensemble de clauses; les instructions associées sont similaires aux instructions de manipulation de point de choix. L'emploi des instructions de manipulation des points de choix, ainsi que l'utilisation des optimisations sont réunis dans l'exemple suivant extrait de [AitKaci90] (pages 40-45).

2.2.4 Les prédicats à effet de bord

L'utilisation du langage Prolog « pur » pour la programmation se limite à la vérification d'une requête, mais ne permet pas d'autre interaction, comme la consultation de fichier ou l'affichage de résultat. Aussi un bon nombre de prédicats « non logiques » ont fait leur rapidement apparition. Le langage Prolog permet maintenant d'échanger des informations avec l'extérieur (système de fichiers ou console), il peut aussi modifier ses connaissances et son comportement (base de faits et règles). Enfin, Prolog propose des mécanismes pour optimiser ou contrôler l'exécution des programmes, ou des prédicats très évolués pour les rendre plus simples à concevoir. Toutes ces possibilités non-logiques sont traitées par les prédicats à effet de bord que nous présentons dans la suite.

Les entrées et sorties

Comme tout programme, un programme Prolog a besoin de communiquer avec le monde extérieur. Pour cela des prédicats internes spéciaux ont été définis. Ces prédicats permettent l'entrée et la sortie de termes.

read(X)	: lit un terme sur le dispositif d'entrée, et l'unifie à X. Si l'unification est possible l'appel du prédicat réussit, sinon c'est un échec. Le prédicat read/1 consomme un terme à chaque appel.

write(X)	: écrit un terme sur le dispositif de sortie; une variable libre peut être envoyée en sortie sans provoquer d'erreur ou d'échec. Le prédicat write/1 se comporte comme le prédicat true/0 qui réussi toujours.
nl	: sort une séquence de passage à la ligne suivante, réussit toujours.
put(C)	: prédicat write/1 limité à un caractère.
get(C)	: prédicat read/1 limité à un caractère.
Ces deux derniers prédicats n'offrent pas le formatage des sorties et des entrées, et permettent de manipuler des données binaires par exemple.	

Il existe enfin des prédicats pour permettre à l'utilisateur de choisir la provenance de ses entrées et la destination de ses sorties, agissant comme l'ouverture et la fermeture des fichiers en langage C (***see***, ***seen***, ***tell***, ***told***).

La base des prédicats

La base de faits de Prolog n'est pas figée. Elle peut être modifiée pendant l'évaluation d'un programme. Des faits peuvent être ajoutés ou retranchés de la base, modifiant alors le programme. Les principaux prédicats de gestion dynamique du programme sont les suivants :

assert(Clause)	: ajoute une clause à un prédicat, la clause est ajoutée après celles qui existent déjà. Ce prédicat peut aussi s'appeler *assertz*.
asserta(Clause)	: comme *assert* mais insère en début de prédicat.
retract(Clause)	: retire la clause donnée d'un prédicat dont la tête est celle de Clause.
abolish(Foncteur,Arité)	: retire toutes les clauses du prédicat Foncteur d'arité Arité.

La coupure

La coupure ou « coupe-choix », notée '!', est essentiellement utilisée pour optimiser les programmes Prolog. Elle permet d'éliminer l'évaluation de branches OU de l'arbre Prolog lorsqu'on sait qu'elles ne sont pas ou plus utiles, en supprimant les points de choix associés de la pile. Elle agit sur tous les points de choix créés entre l'appel du prédicat et le franchissement de la coupure, y compris le point de choix éventuellement créé lors de l'appel du prédicat.
Elle modifie donc la stratégie de Prolog en rendant son exécution déterministe. Par conséquent si la coupure est mal utilisée, le sens donné à un programme peut être différent de celui qui était voulu. La lecture logique d'un programme ne correspond plus à la lecture opérationnelle.

Exemple :

```
(1) ascendant(X,Y)      :- pere(X,Y), !.
(2) ascendant(X,Y)      :- mere(X,Y), !.
```

(3) ascendant(X,Y)	:- ascendant(X,Z), ascendant(Z,Y), !.

Si l'ascendant X de Y est son père, l'évaluation du prédicat *ascendant* sera interrompue après la coupure de la clause (1), et les clauses (2) et (3) ne seront pas évaluées.

Dans la clause (3), les appels successifs au prédicat *ascendant* peuvent générer plusieurs solutions, mais si la coupure de cette clause est franchie, ces solutions seront coupées, empêchant un retour arrière dans cette clause.

Les prédicats collecteurs

Quand il existe de nombreuses solutions à un problème, et que celles-ci doivent être collectées pour continuer la résolution, il faut utiliser le mécanisme de retour arrière et de construire progressivement la liste des solutions. Mais cette méthode alourdit l'écriture des programmes. Les prédicats collecteurs ont été introduits afin de gérer automatiquement cette collection.

Le prédicat *setof*(V, B, C) donne la collection ordonnée C des valeurs de la variable V vérifiant le but B. Les prédicats *bagof*(V, B, C) et *findall*(V, B, C) donnent la collection C non-ordonnée des valeurs de la variable V qui permettent de vérifier le but B.Le prédicat findall produit toutes les valeurs prises ayant conduit à une réussite pour vérifier le but B, le prédicat bagof produit les valeurs ayant généré une arborescence qui réussit.

L'action des différents prédicats est résumée dans la Figure 2-5.

Faits et règles :

```
p(3).    q(3).
p(2).    q(1).
p(1).

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

:- bagof(X, p(X), L).	:- bagof(X, s(X,Y), L).
L = [3, 2, 1].	L = [3, 2, 1].

:- setof(X, p(X), L).	:- setof(X, s(X,Y), L).
L = [1, 2, 3].	L = [1, 2, 3].

:- findall(X, p(X), L).	:- findall(X, s(X,Y), L).
L = [3, 2, 1].	L = [3, 3, 2, 2, 1, 1].

Figure 2-5 - Les différents prédicats collecteurs

2.3 Différents types de parallélisme

Le langage Prolog génère plusieurs formes de parallélisme [GuptaACH], [Chassin94], dont nous présentons les plus utilisées actuellement.

2.3.1 Le parallélisme OU

L'idée est de résoudre de manière indépendante les différentes clauses d'un prédicat, au lieu de les parcourir à l'aide du mécanisme de retour arrière; et par extension, l'ensemble de branches de l'arbre OU de la résolution du programme. Cela revient à une stratégie de parcours en largeur d'abord de l'arbre.
C'est généralement une forme de parallélisme à gros grain car il s'applique au niveau des alternatives pour résoudre un prédicat, qui peuvent ensuite se développer en longues résolvantes. Le parallélisme OU est principalement issu de problèmes de recherche exhaustive de solutions. Cette forme de parallélisme convient à toute architecture parallèle, comme on pourra le voir dans le paragraphe suivant (2.4).

2.3.2 Le parallélisme ET

Le parallélisme ET consiste à calculer simultanément tous les sous-buts d'une clause. Dans ce cas, on doit assurer que l'ensemble des liaisons de variables produit pour chaque sous-but est compatible avec les autres. Ce problème a conduit à deux formes de parallélisme ET, la forme « Et-indépendant », où on n'évalue en parallèle que des sous buts qui n'ont pas de dépendance de variables, soit car elles sont libres soit par ce que l'intersection des ensembles de variables est nulle. La forme « Et-dépendant » se rencontre quand deux sous-buts, ou plus, d'une clause ont une variable commune et sont exécutés en parallèle. Cette forme peut être exploitée de deux façons :

- Deux buts peuvent être exécutés indépendamment jusqu'à ce que l'un d'eux accède ou lie une variable commune. Il est possible de retarder cette synchronisation, et de laisser continuer l'exécution indépendante jusqu'à complétude des deux buts, puis alors de vérifier la compatibilité des liaisons effectuées (back unification).
- Chaque but est exécuté indépendamment, puis lorsqu'un but accède une variable, celle-ci est attachée à un flux, le but générant ce flux est appelé producteur, l'autre but agit en consommateur, en lisant dans ce flux. Le parallélisme existe alors entre le producteur et le consommateur.

2.3.3 Le parallélisme ET/OU

L'utilisation d'un seul type de parallélisme n'est rentable que pour les programmes qui correspondent bien au type de parallélisme choisi. Si on veut maintenir un accroissement en performances pour tous les programmes que l'on traite, il est intéressant de combiner les deux types ET et OU de parallélisme, comme cela a été fait dans plusieurs systèmes, tels Andorra-1[Costa91], Ace [Gupta91] ou PDP[Araujo94].
Cependant, l'utilisation combinée des deux types de parallélismes augmente la difficulté d'implantation. Les techniques employées pour un type de parallélisme pouvant entrer en conflit avec celles nécessaires pour l'autre type, au niveau du contrôle de l'exécution ou de la représentation de données.

2.3.4 Les autres formes de parallélisme

Il existe encore bien d'autres formes de parallélisme, plus ou moins intéressantes, dont les plus classiques sont :

Linda

Dans ce modèle, dit de « tableau noir » (*blackboard*) [Sutcliffe90], [Gelernter89], chaque processeur gère un programme qui peut écrire ou lire des informations dans une zone commune à tous les programmes. Cette zone appelée espace de tuples sert de mécanisme de synchronisation et de communication entre les différents programmes. Cette zone partagée peut être accédée par des programmes écrits avec des langages différents, c'est une sorte de mémoire virtuelle associative.
Le parallélisme engendré ici est de type MPMD (multiples programmes multiples données), ainsi chaque sous-ensemble de processeurs peut être spécialisé pour un certain type de tâche. Ce type de parallélisme convient à tout type d'architecture, avec ou sans mémoire commune.

Parallélisme d'unification

Ce parallélisme est obtenu quand les arguments d'un but sont unifiés avec ceux de la tête d'une clause de même nom et arité. Chaque argument peut alors être unifié en parallèle, ainsi que les différentes composantes d'un terme. Ce type de parallélisme est à grain très fin, et ne peut être exploité efficacement que sur des processeurs spécialisés [Singhal89]. L'évolution actuelle des processeurs et des machines vers des systèmes plus généralistes font que cette forme de parallélisme n'est plus étudiée.

Flot de données (Dataflow)

Ici le parallélisme n'est plus un parallélisme de contrôle, mais de données. Les formes ET et OU de parallélisme sont exploitables avec les flux de données [Smith92], [Kacsuck92a]. Ce type de parallélisme est plutôt à grain très fin, et ne présente plus guère d'intérêt actuellement, car le coût des communications devient de plus en plus élevé en rapport au coût de calcul.
Le système Multilog a aussi la particularité d'avoir été implanté sur une machine SIMD (Single Instruction Multiple Data) [Smith93].

Langages gardés

Les langages logiques gardés tels Concurrent Prolog [Shapiro83] ou Parlog [Gregory87] ont été les premiers langages logiques à profiter des architectures parallèles en exploitant le parallélisme de flot et une forme OU restreinte. La restriction du parallélisme OU vise à supprimer l'explosion combinatoire due à la recherche en largeur créée par les nœuds OU. Le non-déterminisme de Prolog, associé au principe du retour arrière, est supprimé. Enfin la sélection d'une clause d'un prédicat est déterminée par la validité d'une expression de garde [Dijkstra75]. Une fois la garde franchie, la clause est résolue, aboutissant ou échouant, dans ce cas il n'y a pas de reprise, par absence du mécanisme de retour arrière. Cette sémantique correspond à une coupure implicite systématique et étendue pour couper toutes les branches et non plus uniquement en partie droite de l'arbre d'évaluation.
Du fait de ces modifications apportées à la sémantique, un programme qui avait une solution logique peut très bien ne plus en fournir avec un langage gardé. On voit donc que les langages gardés se sont assez éloigné des bases de Prolog, et de sa simplicité

de programmation, ce qui diminue un peu leur intérêt. En plus la gestion du parallélisme n'est pas du tout implicite et reste donc à la charge du programmeur.

2.4 Les principales techniques d'implantation

Nous présentons brièvement les techniques d'implantation proposées pour les deux principales sources de parallélisme, les formes ET et OU, qui ont obtenu le plus de succès. Puis nous introduisons les problèmes classiques liés à l'exécution parallèle de Prolog.

2.4.1 Les modèles OU-parallèles

Dans ce modèle, les différentes alternatives d'un prédicat parcourues habituellement avec le mécanisme de retour-arrière, sont transformées en un ensemble de résolvantes distinctes développées en parallèle. Mais durant un parcours une résolvante va lier un certain nombre de variables, et donc modifier les piles en conséquence.
Le problème de la cohérence des piles se pose lorsque plusieurs résolvantes s'exécutent en parallèle. Plusieurs méthodes ont été proposées dont le partage des piles, la copie des piles ou la reconstruction des piles par recalcul.

Partage des piles

Dans la méthode de partage des piles, les piles locales et globales de la machine abstraite comprennent deux zones, l'une privée, l'autre partagée. La partie partagée contient les portions communes de la résolution entre les processeurs. La partie privée représente la portion locale au processeur la possédant.
Pour gérer les liaisons des variables, la pile de traînée est remplacée ou complétée par de nouvelles structures, comme les tableaux de liaisons (*binding arrays*) ou les fenêtres de hachage (*hashing windows*).

Dans le premier cas, il existe un tableau de liaisons sur chaque processeur, qui contient donc les liaisons des variables des parties communes des piles, ceci afin d'éviter la liaison simultanée d'une variable par les différents processeurs. Ainsi chaque variable partagée ne contient plus une référence, mais un index qui représente sa clé d'accès dans le tableau de liaison de chaque processeur. Quand un processeur change de portion de l'arbre, il doit aussi mettre à jour son tableau de liaisons, en effaçant celles devenues invalides et en copiant les nouvelles dues à sa position courante.

Les fenêtres de hachage sont partagées entre les processeurs, et sont associées à chaque nœud de l'arbre OU. Elles contiennent les liaisons conditionnelles des variables. Une liaison est mémorisée dans la fenêtre de hachage du nœud où la variable est liée. Lors d'un accès à la variable, la fonction de hachage détermine l'entrée de la fenêtre de hachage où devrait être mémorisé la liaison. Si elle n'est pas présente, la fonction de hachage est appliquée au nœud précédent de l'arbre, jusqu'à qu'une liaison soit trouvée, ou si le nœud de création de la variable est atteint.

Il existe encore d'autres solutions que nous ne décrirons pas, mais qui sont présentées dans [GuptaACH].

Enfin, il est évident que le partage des piles n'est envisageable qu'avec l'accès à une zone de données partagée entre les différents processeurs, c'est à dire une architecture à mémoire commune

Copie des piles

Dans ce modèle, chaque machine abstraite dispose d'un environnement séparé, et donc aucun conflit de liaison n'intervient. Le problème vient au moment du partage du travail, où il faut alors recopier des portions de piles entre les processeurs. Or, une fois les piles en place, il faut rétablir l'état de celles-ci au nœud OU de reprise. Il faut encore gérer le cas des liaisons conditionnelles. Cela peut être fait par la méthode de datation (ou Kabu-Wake [Masuzawa86]), ou uniquement à l'aide de la pile de traînée [Ali90].

L'utilisation de la datation des liaisons conditionnelles implique la conservation de la date de la liaison (profondeur dans l'arbre OU) en plus de la mémorisation de la liaison dans la pile. Lors du transfert, les liaisons dont la date est postérieure à celle du point de reprise, sont annulées.

La restauration des liaisons conditionnelles à l'aide de la seule pile de traînée impose la synchronisation des deux processeurs, afin que celui qui obtient une copie de la pile, ait une copie cohérente. Puis, celui-ci simule un retour arrière sur le point de reprise, entraînant la déliaison des liaisons conditionnelles.

La copie des piles peut être utilisée sur n'importe quelle architecture de machine car elle n'impose pas de zone de données communes, mais simplement le moyen d'échanger des données entre les zones.

Reconstruction

Cette méthode est une sorte d'antithèse des deux précédentes. Elle évite le partage et aussi la copie des piles. Les parties de l'arbre OU sont calculées de manière complètement indépendante. Le partage de travail est réalisé par recalcul de l'état de la machine abstraite à un nœud OU de l'arbre [Clocksin88]. La seule information que les processeurs échangent alors est le chemin pour retrouver cette position.

Nous préciserons les deux dernières méthodes dans le cadre des choix réalisés pour de la conception de notre système, au chapitre 4.

2.4.2 Les modèles ET-parallèles

Dans ces modèles, le parallélisme est obtenu par l'évaluation parallèle des différents buts d'une clause d'un prédicat.

Parallélisme ET-indépendant

Ici le problème est la détection de l'indépendance des buts d'une clause, vis à vis des variables contenues dans cette clause. Ceci est déterminé par l'utilisation de graphes, construit soit dynamiquement, soit par compilation, ou par combinaison des deux. Ces graphes sont mis à jour durant l'exécution, au fil des liaisons et déliaison. Mais les opérations de mise à jour coûtent cher.
Aussi, une méthode plus précise à la compilation a été proposée pour réduire ces surcoût d'exécution. Les programmes sont compilés en des graphes d'expressions conditionnelles (CGE, Conditional Graph Expression), qui permettent un traitement plus rapide des tests de dépendance. Cependant, ces graphes peuvent diminuer le parallélisme potentiel, c'est pourquoi ce type de parallélisme est encore appelé ET-restreint (*Restricted And*).

Parallélisme ET-dépendant

Dans ce modèle, les buts d'une clause sont aussi évalués en parallèle, mais il existe une dépendance de données entre eux. Dans ce cas, pour deux buts, un des processeurs gère un but en producteur de données, et l'autre en consommateur. Les variables de dépendance forme alors un flux de données.
Le problème d'implantation est la détermination de l'instance de la variable agit en producteur, et quelle instance agit en consommateur. Une fois cette détermination établie, il faut alors prendre en compte la distinction au niveau de l'exécution. En effet, un processeur qui accède une variable qui n'est pas liée, à l'état consommateur, doit attendre une liaison pour continuer. Il faut donc gérer en plus un mécanisme de suspension.

2.4.3 Le problème des effets de bord

L'exécution parallèle d'un programme Prolog ne pose donc a priori pas trop de problème. Cependant, si l'on s'intéresse aux différents prédicats à effet de bord, on remarque que leur validité dépend fortement de l'ordre séquentiel de leur exécution.

Dans le parallélisme OU, le fait d'évaluer les différentes clauses d'un prédicat en parallèle, ne garantit pas l'ordre de leur exécution. Ainsi, un effet de bord de la première clause peut très bien être réalisé après celui d'une autre clause. Ce point est encore plus important avec la coupure, qui repose sur l'ordre d'évaluation des clauses.

Pour le parallélisme ET, il existe aussi le cas où différents buts d'une clause exécutés en parallèles, contiennent des effets de bord. Si l'ordre séquentiel de parcours de la clause n'est plus respecté, les effets de bord risquent d'être effectués au mauvais moment.

Le traitement des effets de bord est donc un point crucial d'un bon système Prolog parallèle, qui doit permettre leur utilisation de manière correcte.

Les méthodes de gestion de ces prédicats font l'objet d'un chapitre particulier (chapitre 3) pour le modèle de parallélisme OU. Le parallélisme ET se situe bien au-delà du cadre de ce document, aussi la gestion des effets de bord associée n'est pas incluse.

2.4.4 Ordonnancement du travail

Pour que l'exécution parallèle d'un programme Prolog soit efficace, il faut maintenir le plus grand nombre de processeurs actifs. Par exemple, dans un système OU-parallèle, chaque alternative d'un prédicat peut être évaluée par un processeur, ce qui correspond à une exploration en largeur d'abord de l'arbre d'évaluation. Si ce partage n'est pas limité, le nombre de processeurs nécessaire peut devenir très grand. Aussi, il faut contrôler le grain de parallélisme, pour fonctionner correctement avec un nombre limité de processeurs et obtenir une bonne efficacité et ne pas perdre de vitesse.
Ainsi, la création dynamique d'une nouvelle tâche de résolution sur un processeur ne doit être effectuée que si son coût de création est inférieur au grain du parallélisme. C'est à dire, si le transfert des données et la mise en place du calcul de résolution durent moins longtemps que le temps à attendre pour traiter localement l'alternative.
L'ordonnancement du travail dépend aussi de la gestion de effets de bords. Si ceux-ci doivent respecter la sémantique d'exécution séquentielle, leurs actions respectives doivent être ordonnées dans le temps, ce qui va perturber l'ordonnancement du travail, en introduisant des relations de précédence temporelle.

2.5 Quelques systèmes Prolog parallèles

Il existe de nombreux systèmes Prolog parallèles, mais ils sont construits en majorité sur des architectures avec mémoire commune. Aussi, nous classons les systèmes non pas par leur type de parallélisme, mais plutôt par le type d'architecture cible. Ainsi il sera plus simple de comparer notre approche à celle des systèmes déjà existants.

2.5.1 Architecture avec mémoire commune

Les principaux systèmes développés sur ce type d'architecture sont ACE [Gupta95], Andorra-I [Yang93], Aurora [Carlsson90], &-Prolog [Green90], Muse [Karlsson92]. Il y a encore beaucoup d'autres systèmes, plus ou moins complets, que l'on peut retrouver dans [Chassin94] et [GuptaACH]. Aussi nous ne présenterons que les plus significatifs pour l'étude notre travail, c'est à dire, qui ont des caractéristiques proches, par exemple dans leur modèle de parallélisme, ou pour la gestion des effets de bords, ou pour la méthode de régulation de charge.

Aurora

Ce système est fondé sur l'utilisation le parallélisme OU de manière implicite, c'est à dire sans modification du code du programme pour gérer le parallélisme. L'architecture cible est une machine à mémoire commune. Chaque travailleur parcourt des portions de l'arbre de recherche, partageant au besoin les piles des portions de branches parcourues simultanément. Un travailleur comprend deux unités, un moteur Prolog et un ordonnanceur.
Les liaisons des variables des branches partagées sont réalisées par l'emploi de tableaux de liaisons (*binding array*) privés pour chaque travailleur. Les tableaux de liaisons doivent être mis à jour lorsque qu'un travailleur se déplace dans l'arbre d'évaluation du programme, et le maintient de ces liaisons est plus coûteux que celles utilisées classiquement de l'ordre de 20 à 25%.

Dans Aurora, l'arbre d'évaluation à un instant donné, est partagée en une région publique et une région privée. Dans la région publique, tous les travailleurs peuvent acquérir des alternatives non encore parcourues. Par contre, un travailleur ne peut acquérir des alternatives d'une branche dans la région privée d'un autre. C'est l'ordonnanceur qui gère le caractère privé ou public des portions de branche, selon les disponibilités en travailleurs devenus inactifs.
Le système Aurora a été conçu dans l'optique de gérer les effets de bord avec leur sémantique séquentielle. A cette fin, le système synchronise les travailleurs selon leur position dans l'arbre d'exécution du programme, lorsque ceux-ci effectuent des effets de bord. Ainsi, seul le travailleur le plus à gauche de l'arbre est autorisé à effectuer ses effets de bord. La sémantique séquentielle est ainsi préservée, les effets de bord étant correctement ordonnés de gauche à droite de l'arbre d'exécution du programme Prolog. Pour accroître les performances parallèles, au détriment de la portabilité toutefois, des effets de bord asynchrones peuvent être employés.
De nombreuses études ont été consacrées à l'ordonnancement dans Aurora. Celles-ci ont été facilitées par l'existence d'une interface de programmation entre les deux unités d'un travailleur. Elles ont donné lieu à plusieurs ordonnanceurs [Carlsson90], avec lesquels le système montre de bons accroissements de performances dus au parallélisme.

Muse

Le système OU-parallèle Muse [Karlsson92] est conçu pour des architectures à mémoire commune (UMA et NUMA) tels que Sequent Symmetry, Sun Galaxy, BBN Butterfly, ...
Ce système supporte le langage Prolog complet, tous les effets de bords sont pris en compte, avec peu d'annotations de programme ajoutées par l'utilisateur.
Le modèle de Muse utilise des travailleurs qui contiennent chacun un moteur Prolog séquentiel basé sur une extension de la WAM. Muse exploite le système séquentiel Sictus Prolog comme base de moteur Prolog.
Chaque travailleur possède un espace d'adressage local, mais partage aussi un espace commun à l'ensemble des travailleurs, qui contient les nœuds OU du graphe de l'arbre. Lorsqu' un travailleur devient inactif, la propagation du travail est réalisée par copie des piles de la WAM depuis un travailleur actif. Pour optimiser le transfert des piles, la copie ne comprend que les parties de piles différentes entre les deux processeurs.
Un travailleur contient outre le moteur Prolog, un autre composant, l'ordonnanceur, qui avec ceux des autres travailleurs, doivent ordonner le travail entre les différents travailleurs, et aussi gérer les effets de bord avec leur sémantique séquentielle.

ACE

Ce projet qui tentait initialement de combiner les formes ET et OU de parallélisme, considérant que dans le cas général, les programmes contiennent un mélange de ces deux formes de parallélisme [Gupta91]. Il vise actuellement à utiliser toutes les formes de parallélisme afin de profiter au mieux de la structure d'un programme [Gupta95]

ACE utilise le principe des copies de piles introduit pour MUSE pour la gestion des liaisons conditionnelles, mais utilise le recalcul plutôt que la recopie, pour transférer l'état d'une machine abstraite. L'utilisation des tableaux de liaisons est aussi envisagée, mais sous une forme modifiée.
Le parallélisme ET est géré par la forme indépendante, avec des graphes d'expressions conditionnelles.
Le support des effets de bord est proposé par l'utilisation des algorithmes utilisé par le système Muse pour le OU parallélisme et les graphes d'expressions conditionnelles, pour déterminer si un effet de bord est réalisable ou doit être suspendu. Il nécessite une de zone de mémoire partagée, même si elle est réduite.
On ne dispose actuellement pas de résultats de l'implantation complète de ce modèle, mais les premiers résultats montrent une bonne accélération [Gupta95].

2.5.2 Architecture sans mémoire commune

Avec ce type d'architecture, la quantité de systèmes implantés est nettement plus faible, et on peut dire qu'il n'existe encore pas de système complet et exploitable. Il existe cependant des systèmes exploitant un langage plus ou moins proche de Prolog. Certains sont décrits dans [Kacsuk92].

OPERA

Le projet OPERA, [Briat90], [Favre92] est le précurseur de PLoSys au sein de notre laboratoire. L'objectif était d'implanter le modèle OU parallèle multiséquentiel sur une architecture sans mémoire commune utilisant un réseau de Transputers T800 (SuperNode). Les principales caractéristiques du Transputer sont la gestion de processus et des liens de communications par le Transputer lui-même. Ce projet a nécessité un travail important en dehors de l'implantation de Prolog, en particulier pour obtenir sur le SuperNode un système d'exploitation pour pouvoir implanter le système Prolog [DaCosta93].
Le modèle OU multiséquentiel a été choisi car il convient bien au type d'architecture du SuperNode, car il génère un grain de parallélisme suffisant. Un ensemble de travailleurs, processus placés sur les Transputers, coopèrent à la résolution du programme, et leur activité est gérée par un ordonnanceur centralisé, placé sur le Transputer de contrôle. Le moteur Prolog est un émulateur d'une machine abstraite dérivée de la WAM (TWAM [Favre92]), est réalisé en assembleur du Transputer pour un maximum d'efficacité.
Le système OPERA ne supporte que le langage Prolog pur, bien que la gestion de la coupure inhibitrice ait été prévue au niveau de la TWAM. Les performances du système sont bonnes, tant en séquentiel, qu'en parallèle, avec de bons accroissements de vitesses dans ce cas. Le point faible de ce projet est le manque de portabilité, dû à l'implantation très dépendante de l'architecture du SuperNode et de la TWAM liée à l'assembleur du T800.

Le système OU parallèle de l'université de Malaga

Comme OPERA, ce système multiséquentiel est implanté sur une architecture à base de Transputer [Benjumea93]. Cependant, il existe des différences importantes, dont la

principale réside dans la gestion du travail, qui est ici totalement distribuée sur les Transputers, n'utilisant ainsi pas le Transputer de contrôle pour l'ordonnancement. Le moteur Prolog qui est aussi un émulateur est toutefois bien plus lent que la TWAM d'OPERA, ce qui dégrade les performances séquentielles, et favorise par ailleurs les valeurs d'accroissement de vitesses en réduisant virtuellement le coût des communications, [Briat90], [Benjumea93]. Enfin ce système n'intègre pas la gestion des effets de bord.

3DPAM

3DPAM est un système à parallélisme ET et OU utilisant un schéma d'exécution guidé par les données [Kacsuk92b]. Le moteur Prolog dérive de la classique WAM, mais présente des différences importantes au niveau des instructions de contrôle, qui servent à exploiter les deux formes de parallélismes supportées. La description complète et exacte de ce modèle est détaillée dans [Kacsuk92a].
Globalement, un programme Prolog est transformé en un graphe de flot de données, ou les sommets représentent des tâches de calcul (Et, Ou, Unification, Appel,...), et les arcs représentent les chemins de communications des données qui sont empaquetées dans des jetons (*tokens*) circulant sur les arcs. Pour pouvoir déplacer les données sans problèmes, même avec les liaisons des variables, ce modèle intègre la clôture des environnements [Conery87] transportés par les jetons. Cette opération consiste à éviter que des liaisons soient réalisées hors de l'environnement, transformant toute liaison externe en une liaison vers une nouvelle liaison locale.
Il existe actuellement une implantation de ce modèle sur Transputer. Celle-ci a montré que le modèle est très gourmand en ressource mémoire, du fait de la clôture des environnements, et aussi que le parallélisme étant à grain fin, l'efficacité de la distribution ne peut être obtenue qu'avec des coûts de communications très bas en rapport de la vitesse du processeur. Ce modèle ne convient donc pas aux nouvelles architectures sans mémoire commune, qui ont un rapport *vitesse processeur / vitesse communications* très élevé, à moins de contrôler le grain de parallélisme.

PDP

Ce système distribué gère le *ET-indépendant* et *OU* parallélisme simultanément [Araujo94], de manière explicite pour la partie *OU*, et implicite pour le *ET*. Le partage de travail n'est pas réalisé par copie des piles, mais par recalcul à l'aide des chemins suivis dans l'arbre d'exécution, appelés encore oracles [Clocksin88], ce qui est censé diminuer le coût des communications. La structure du système est hiérarchique du type maître-esclaves, bien adapté à l'architecture matérielle employée, le SuperNode. Le langage Prolog supporté ne comprend aucun effet de bord.
Malheureusement, aucune comparaison de ce système n'est vraiment possible d'après l'article [Araujo94] , même si les accélérations semblent très bonnes, car aucun temps d'exécution n'est donné. Par contre l'intérêt du recalcul semble négligeable au vu des résultats présentés, surtout qu'il n'y a encore aucune gestion des effets de bords, tâche qui complique encore le système à base de recalcul.

VPL

Vienna Parallel Logic est un langage persistant et un système de programmation qui supporte l'exécution concurrente de processus communicants, et la résolution de différentes alternatives à l'aide de retours arrières programmés [Forst94]. Il associe les mondes des langages séquentiels de programmation logique (Prolog) et les langages de programmations concurrents (Parlog [Gregory87]). Les aspects de ces deux classes de langages peuvent être combinés à un degré quelconque lors d'une exécution.

Une forme un peu différente de mise en œuvre du modèle Linda est incluse dans le système VPL qui utilise le noyau de coordination CoKe [Kuhn93]. Un programme ayant accès à CoKe peut partager des données avec d'autre programme, dans une sorte de sac commun. Le noyau CoKe permet en outre l'échange d'objets persistants entre différents langages, par exemple entre C et Prolog. Dans le cas de Prolog, cela permet de partager une base de requêtes à résoudre. Ce type de parallélisme est proche de celui offert par le modèle Linda, mais CoKe dispose en plus d'un mécanisme tolérant aux pannes. Son utilisation est préférable pour un parallélisme à très gros grain.

On peut encore trouver d'autres noyaux de ce type, avec ou sans la fonction de tolérance aux pannes, qui peuvent alors remplir le même rôle, et aborder le même type de parallélisme.

3. Le traitement des effets de bord

Ce chapitre traite de la gestion des effets de bord avec le respect de leur sémantique séquentielle. Cette étude est menée dans le cadre d'utilisation d'un système sans mémoire commune, avec un modèle d'exécution OU-parallèle multiséquentiel.
Notre approche est basée d'une part sur le maintien de la connaissance de l'ordre séquentiel au niveau de l'ordonnanceur, et d'autre part, sur l'utilisation de mécanismes de contrôle dédiés à chaque effet de bord, au sein des travailleurs.

3.1 Sémantique séquentielle des effets de bord

Pour prendre en compte les opérateurs à effet de bord lors d'une exécution parallèle, deux approches sont possibles. Soit les opérateurs sont définis avec une politique globale de synchronisation pour éviter les interblocages, soit ils respectent la sémantique séquentielle, ce qui permet de garder le langage inchangé.

La première méthode a souvent été utilisée, en particulier pour la coupure, car elle est généralement plus simple à gérer, et plus adaptée à la programmation parallèle. Elle minimise la complexité de la gestion des contraintes de synchronisation, et elle préserve les performances dues au parallélisme. Cependant il devient impossible de réutiliser un programme conçu pour être exécuté en séquentiel, ou alors il faut le modifier pour prendre en compte cette nouvelle sémantique.

La seconde façon de gérer les effets de bord, le respect de la sémantique séquentielle, augmente sensiblement les contraintes de synchronisation, et diminue potentiellement le parallélisme exploitable. Cependant le résultat de cette approche est l'utilisation quasi immédiate de tout programme Prolog développé précédemment sur un système séquentiel. C'est la voie choisie pour implanter les effets de bord dans PLoSys; aussi avant de présenter le traitement de chacun de ceux-ci, il est important de rappeler leur sémantique séquentielle.

3.1.1 La coupure

La coupure est basée sur le parcours séquentiel de l'arbre de recherche. Son action est de supprimer le parcours des alternatives non-encore explorées du prédicat, ainsi que toutes les alternatives créées depuis l'entrée dans le prédicat par les différents sous-buts. Le programmeur contrôle les branches coupées du fait du caractère déterministe et prévisible de la stratégie séquentielle.
Mais, le sens réel de la coupure est souvent donné par le programmeur, qui va respecter ou non le point de vue de la logique dont est issu Prolog. Dans l'exemple classique suivant la coupure peut changer le sens du programme :

Exemple :

(1) max(X,Y,X) :- X ≥ Y, !.
max(X,Y,Y) :- X < Y.

```
(2)     max(X,Y,X) :- X ≥ Y, !.
        max(X,Y,Y) .
```

Dans le premier cas, la coupure sert uniquement à optimiser le programme. Si elle est enlevée le programme aura toujours le même comportement. Par contre dans la seconde version, le programme produit le bon résultat uniquement à cause de la coupure. Si celle-ci est ôtée, le programme aura un comportement incorrect, il ne respecte plus le sens logique de Prolog.
Ces deux utilisations de la coupure sont couramment appelées coupure verte, lorsque celle-ci peut être enlevée sans changer le sens du programme (1); et coupure rouge quand elle est nécessaire à la préservation du comportement prévu du programme (2).

Cet exemple nous montre déjà combien il peut être important de respecter l'ordre séquentiel d'un programme. Il peut avoir été conçu comme l'exemple (2) et donc ne plus avoir de sens s'il n'est pas évalué avec la sémantique séquentielle originelle.

Variantes utilisant la coupure

Certaines structures de contrôle ont été ajoutées à Prolog, dont la négation et la structure *if-then-else* qui sont construites à l'aide d'une forme particulière de la coupure, la coupure locale, notée *!!*, dont la portée est limitée à la disjonction qui la contient. La négation est un cas particulier d'utilisation de la structure *if-then-else*.

Exemple :

négation	not(P)	est transformé en	(P,!!,fail;true)
if-then-else	P -> Q; R	est transformé en	(P,!!,Q;R)

Dans ces deux constructions, la coupure locale (!!), si elle est franchie, restaure l'état de la pile des points de choix, à celui de l'entrée dans la disjonction. Son effet est donc limité aux points de choix créés dans la disjonction.

La coupure locale n'est pas distinguée de la coupure normale dans la suite du chapitre car elle représente un cas particulier de la coupure classique. Sa gestion propre peut être adoptée au moment de l'implantation du système.

3.1.2 Les entrées et sorties

Les entrées et sorties sont exécutées immédiatement et ne génèrent pas de point de choix. Elles sont donc ordonnées par leurs positions respectives dans l'arbre d'évaluation, sans autre forme d'interaction (voir Figure 3-1). Là aussi, le programmeur peut les placer correctement dans son programme en raison du caractère déterministe et prévisible de la stratégie séquentielle.

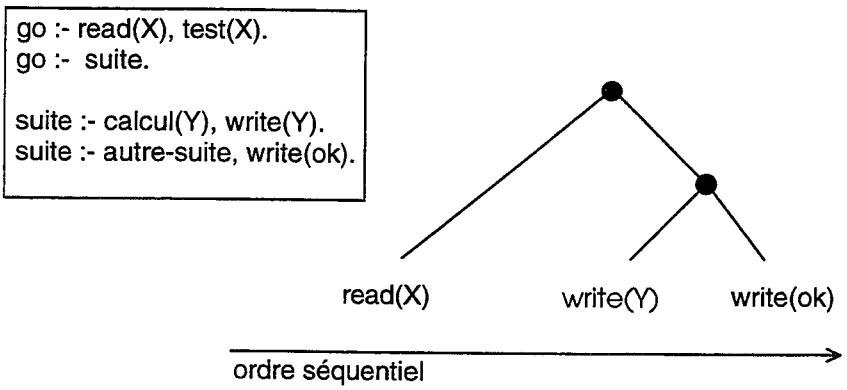


Figure 3-1 - Entrées et sorties séquentielles

3.1.3 Les prédicats collecteurs

La sémantique des prédicats collecteurs respecte l'ordre de parcours normal de l'arbre de recherche pour les prédicats *bagof* et *findall*, et l'ordre des termes pour le prédicat *setof* (Figure 3-2).

p(3). p(2). p(1).	:- findall(X, p(X), L). :- bagof(X, p(X), L). L = [3, 2, 1].	:- setof(X, p(X), L). L = [1, 2, 3].

Figure 3-2 - Evaluations comparées de *findall / bagof* et *setof*

A priori l'évaluation parallèle des prédicats collecteurs ne pose donc pas de problème, mais il faut assurer que l'ordre d'exécution des différentes composantes issues d'un prédicat *bagof* ou *findall* est identique à celui obtenu par l'évaluation séquentielle. En effet, un programme n'aura un comportement strictement identique à sa version séquentielle que si cet ordre est respecté (on pourra noter cependant que le standard en cours de définition ne prévoit pas d'imposer un ordre [Covington93]). Pour le prédicat *setof* le seul point important est d'obtenir le tri sur l'ordre des termes.

3.1.4 La base des prédicats

De tous les prédicats à effets de bord, les prédicats gérant la base des prédicats sont les plus délicats à aborder, vu le nombre important de problèmes qu'ils introduisent. Dans un premier temps nous comparerons les moyens requis pour l'implantation de la gestion dynamique de programme (modification de la base des prédicats), entre un système interprété et un système compilé. Ensuite deux sémantiques d'exécution séquentielle seront exposées pour la gestion dynamique de la base des prédicats.

3.1.4.1 De l'interprétation vers la compilation

Les premiers systèmes Prolog utilisaient l'interprétation pour l'exécution d'un programme. Or cette technique permet facilement la modification de programme pendant de l'exécution, ce qui est généralement mis à profit lors de la mise au point des programmes, ou pour la génération de nouvelles parties de programme à l'intérieur-même de celui-ci. Cela a permis de définir simplement des prédicats Prolog qui manipulent la base des prédicats, en ajoutant ou retranchant des clauses dans le code du programme.

Si la compilation de Prolog procure un gain important en performances, elle pose un problème pour les prédicats dynamiques.

Ainsi, pour implanter le prédicat *assert* de Prolog, un compilateur de clauses complet doit être introduit dans le programme Prolog (ce qui occupe un espace mémoire important). Il faut aussi pouvoir modifier le code compilé (décompilation présentée dans [Dazy89]) du programme en cours d'exécution (en évitant de le faire par recopie pour des raisons de coût mémoire et en temps). Enfin le temps de compilation dynamique du prédicat s'ajoute à son temps d'exécution, ce qui réduit d'autant les performances du système.
Heureusement, ceci n'est pas toujours nécessaire, surtout lorsque la nature dynamique du prédicat est connue dès la compilation, comme pour les prédicats simples dans [Buettner88]. Dans ce cas tous les prédicats peuvent être compilés en une fois et une marque indique leur validité durant l'exécution du programme, ne créant ainsi pas de modification de code, mais uniquement de l'état de ses prédicats. Si l'on veut compiler une clause plus complexe, une requête vers un service spécialisé pourra être envoyée. Ceci à l'avantage de libérer de l'espace mémoire, mais entraîne alors un surcoût de communication lors de l'addition d'un prédicat dans la base. L'enrichissement de la base des prédicats est donc toujours une opération très coûteuse en temps.

Le prédicat *retract* pose moins de problèmes car un marquage des prédicats suffit alors pour distinguer leur statut, présent ou absent de la base. On peut par exemple utiliser des dates, ou un intervalle de temps [Lindoholm87].

3.1.4.2 Problèmes de sémantique d'exécution

Les prédicats de gestion de la base des prédicats posent aussi des problèmes de sémantique d'exécution, principalement dus au choix d'implantation du langage. Nous étudierons les deux sémantiques communément admises [Lindoholm87] et [Buettner88]. La première repose sur une modification immédiate de la base, la deuxième reste plus « logique » en ne validant la modification qu'à la sortie du prédicat en cours d'évaluation. Nous les présenterons à l'aide de l'exemple classique (pour ce problème) suivant :

```
(1)     p :- assertz(p), fail.
        p :- fail.
```

```
(2)     q :- fail.
        q :- assertz(q), fail.
```

Prise en compte immédiate

Dans le cas de prise en compte immédiate, les appels à *p* et à *q* réussissent. En effet, les changements de la base sont effectués et pris en compte immédiatement. Ainsi après le retour arrière du deuxième échec, la base contient déjà le fait *p* pour (1) et le fait *q* pour (2). Ce principe n'est pourtant pas répandu, bien qu'il semble naturel d'un point de vue programmation. Cette sémantique pose un petit problème lors de son implantation avec un compilateur dérivé de la WAM. En effet, pour un prédicat donné si une seule clause est sélectionnable (prédicat à une clause ou résultat d'indexation), aucun point de choix n'est créé. Si dans cette clause, une clause est ajoutée en fin de prédicat (*assertz*), elle ne pourra être évaluée lors du retour arrière car il n'y aura pas de point de choix. Pour éviter cette situation, les prédicats à une clause doivent aussi créer un point de choix à leur appel. Cela diminue alors fortement l'intérêt de l'indexation pour la préservation de l'espace de la pile des points de choix, et le gain de temps en découlant.

Prise en compte différée

La prise en compte différée (ou logique [Buettner88]) permet la poursuite de l'exécution dans un état stable du code, autrement dit le code ne change pas entre deux mises à jour d'un point de choix. Ainsi, pour notre exemple aucun des appels ne doit réussir. Pour (1), à l'appel de *p*, un point de choix est créé (sauf optimisation), celui-ci mémorisera la seconde clause de (1). Lors du retour arrière, ce point de choix sera consulté, puis la seconde clause sera exécutée et échouera. La modification du code n'interviendra qu'après. Il en est de même pour (2), où après l'échec de la seconde clause, il y aura bien échec de l'appel à *q*, et non-réussite, car la modification aura lieu seulement l'échec. La mise en oeuvre de cette sémantique passe par l'utilisation de la recopie d'un prédicat lorsqu'il est évalué, pour pouvoir modifier la base sans que le prédicat soit lui-même modifié par ses actions sur la base, ou par la gestion des prédicats dynamiques différenciée des prédicats statiques.

3.1.4.3 La sémantique séquentielle générale

Nous venons de voir que la sémantique d'exécution séquentielle peut varier selon le type d'implantation choisie. Cependant, le programmeur contrôle dans tous les cas, l'état de la base et ses modifications de la manière voulue à cause du caractère déterministe et prévisible de la stratégie séquentielle.

3.2 Problèmes principaux

Avant de présenter les principes de la gestion parallèle des effets de bord, avec le respect de leur sémantique d'exécution séquentielle, il faut distinguer les problèmes sous-jacents.

Le premier est du à l'aspect déterministe de la stratégie séquentielle qui disparaît lors de l'exécution parallèle. Ainsi, les prédicats à effet de bord qui dépendent justement du déterminisme de l'exécution séquentielle, ne fonctionnent plus correctement. Pour parer à ce problème, il faut donc pouvoir replacer un effet de bord par rapport au temps séquentiel de l'exécution du programme. La connaissance de l'ordre séquentiel des prédicats à effet de bord constitue donc le point central de la gestion des effets de bord.

Le deuxième problème à resoudre, est la gestion proprement dite des prédicats à effet de bord. Globalement, ils peuvent être séparés en trois catégories. La première contient uniquement la coupure, qui est le seul prédicat à agir directement sur l'arbre ET/OU en éliminant les branches à droite de la clause « coupée » d'un nœud OU. La deuxième contient les prédicats d'ordre supérieur, ou prédicats collecteurs, qui permettent une synchronisation des branches d'un sous-arbre. Enfin la troisième englobe tous les autres effets de bord. Ceux-ci dépendent ou agissent sur l'environnement d'exécution, mais ne modifient pas directement l'arbre de recherche.

Le troisième point est l'interaction des différents effets de bord entre eux. En effet certain prédicats n'ont pas d'autre conséquence qu'une action extérieure, comme les entrées et sorties. Mais d'autres prédicats, comme la coupure, ont des répercutions directe sur l'exécution d'un programme. Il faut donc pouvoir gérer les éventuelles interactions apparaissant lors de l'exécution.

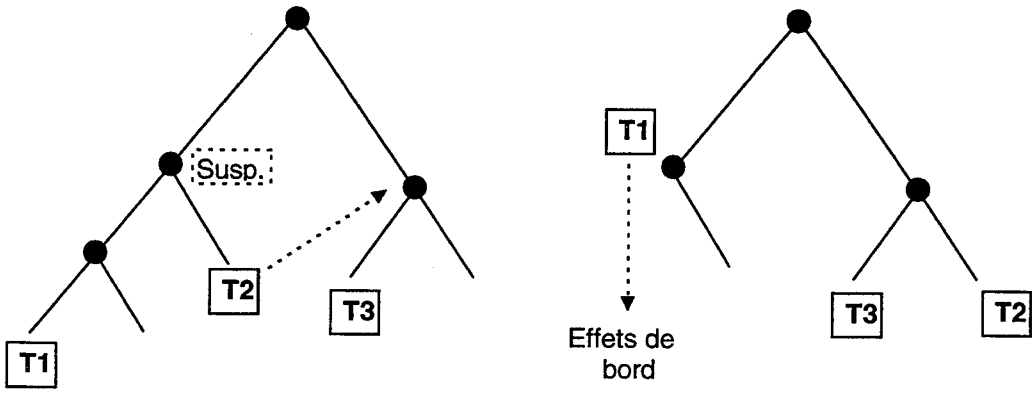
Enfin, le dernier problème concerne l'influence des effets de bord sur la stratégie de régulation de charge.

3.3 Retrouver l'ordre séquentiel

Afin de pouvoir gérer correctement les effets de bord, il est nécessaire de pouvoir simuler une exécution séquentielle, ou du moins, pouvoir reconstruire la chronologie séquentielle de l'exécution des prédicats concernés.

3.3.1 Solutions classiques

Ces solutions sont proposées et implantées principalement dans les systèmes à mémoire commune et utilisent le fait que les travailleurs partagent leurs informations de contrôle.
Ainsi, lorsqu'un travailleur veut effectuer un effet de bord, il peut déterminer s'il est le plus à gauche de l'arbre d'évaluation en utilisant les informations sur les nœuds de l'arbre OU, à l'aide d'un algorithme de recherche de sa position dans l'arbre [Hausman90], [Karlsson92].
Si le travailleur est le plus à gauche de l'arbre, il peut exécuter son effet de bord normalement, et il continue son travail. Par contre, si le travailleur n'est pas le plus à gauche de l'arbre, il ne doit pas effectuer son effet de bord. Dans ce cas il suspend son travail en cours et va quérir une nouvelle tâche dans l'arbre.



Le travailleur T1 peut effectuer ses effets de bord car il est le plus à gauche de l'arbre. Le travailleur T2 qui veut effectuer des effets de bords voit son travail suspendu, et reprend une autre partie de l'arbre.

Le travailleur T1 reprend le nœud suspendu et peut effectuer les effets de bord car il est le plus à gauche de l'arbre. Le travailleur T2 continue son travail dans la nouvelle branche.

Figure 3-3 - Effets de bords et architecture à mémoire commune

Cette approche n'est pas appropriée pour notre modèle sans mémoire commune pour plusieurs raisons, dont les principales sont l'absence de zone mémoire commune et le coût d'une communication qui est très important par rapport au temps de calcul. La mise en œuvre nécessiterait l'utilisation d'un grand nombre de messages, afin de répercuter toutes les modifications sur des parties partagées, ainsi que les différentes synchronisations à gérer lors des mises à jour. De plus la détermination du travailleur le plus à gauche de l'arbre engendrerait aussi un grand nombre de communications.
Cette solution n'est donc pas envisageable en l'état pour gérer efficacement les effets de bord dans un modèle sans mémoire commune. Il faut donc adopter une autre approche pour connaître la position des travailleurs dans l'arbre.

3.3.2 Solution sans mémoire commune

Il faut donc arriver à maintenir la connaissance de la position des différents travailleurs parcourant l'arbre OU d'exécution du programme. Mais comme il est très coûteux de partager des informations, il vaut mieux essayer de maintenir la connaissance de l'état global du système à l'aide des mécanismes de communications déjà présents pour répartir le travail. Il faut éviter au maximum d'ajouter des communications à celles déjà nécessaires.

Pour simplifier un peu le problème, nous considérons que toutes les alternatives non-explorées d'un point de choix sont transférées en même temps. L'unité de transfert est donc le point de choix.complet.

Le maintien de la connaissance de l'ordre séquentiel

Dans le modèle d'exécution multiséquentielle, chaque travailleur parcourt en fait une partie séquentielle de l'arbre de recherche, ce qui représente un intervalle temporel de l'exécution séquentielle pure (Figure 3-4). La somme ordonnée de ces intervalles est équivalente à l'exécution séquentielle car aucune partie de l'arbre n'est parcourue plus d'une fois (pas de recalcul) ou ignorée. Ainsi lors de l'exécution parallèle, le problème est de reclasser les travailleurs selon la partie de l'arbre qu'ils parcourent , par rapport au temps séquentiel (Figure 3-4).

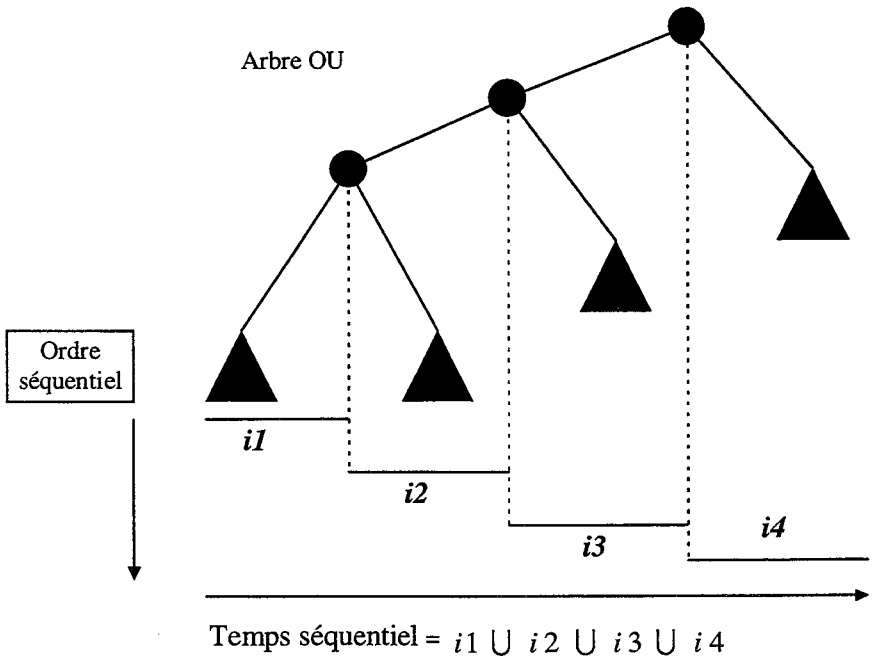


Figure 3-4 - Ordre séquentiel et parcours

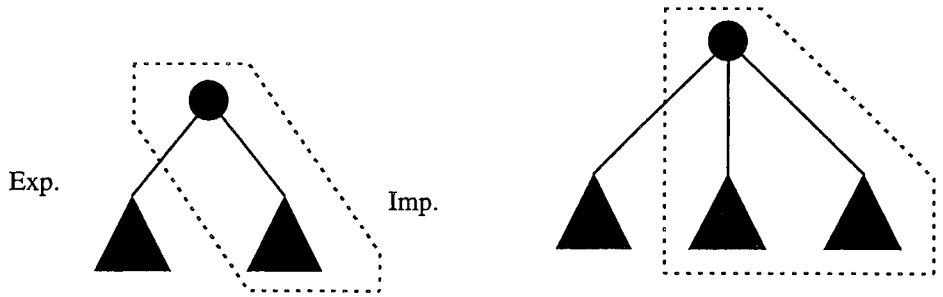
Si l'on veut profiter des communications déja existantes, il faut gérer le maintien de la connaissance des positions séquentielles lors du transfert de travail. Or, avec le modèle OU-parallèle multiséquentiel adapté aux architectures sans mémoire communes, il existe trois possibilité de partager du travail en fonction de relation chronologique :

- Transfert du plus récent point de choix créé sur un travailleur surchargé.
- Transfert d'un point de choix quelconque.
- Transfert du plus ancien point de choix.

Si les deux premières montrent vite leurs limites quant à la possibilité de déduire l'ordre séquentiel des seuls transferts de point de choix, la dernière semble très adaptée.

Point de choix le plus récent ou quelconque

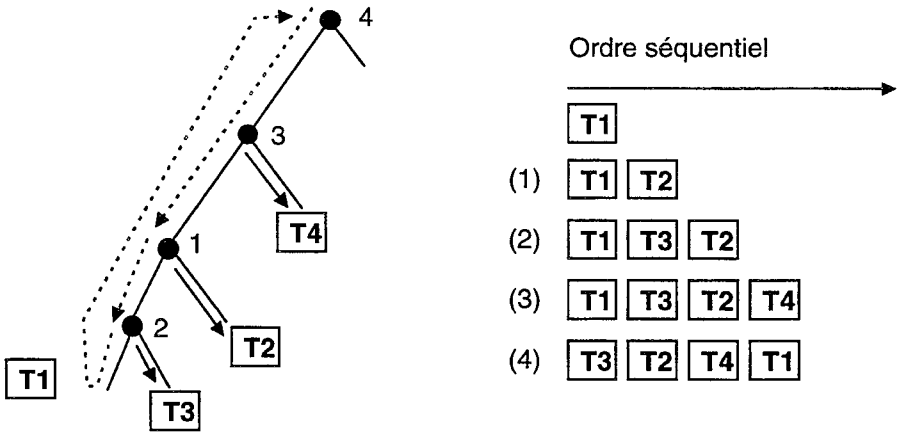
Lors du transfert du plus récent point de choix, on peut noter la relation d'ordre sur les intervalles de temps entre le travailleur exportateur et l'importateur. Le parcours de l'exportateur se situe avant celui de l'importateur dans la chronologie séquentielle.



Le travail de l'importateur suit le travail de l'exportateur.

Si toutes les alternatives d'un point de choix sont transférées en même temps, le travail de l'importateur suivra encore celui de l'exportateur.

Figure 3-5 - Ordre séquentiel lors d'une exportation



L'exécution du programme débute sur T1.Il échange un premier point de choix avec T2, en (1). T1 continue son travail jusqu'en (2) où il exporte encore un point de choix vers T3. Dans la foulée, il exporte un autre point de choix vers T4, en (3). Finallement T1 effectue un retour arrière en (4).

Figure 3-6 - Transfert du plus récent point de choix et ordre séquentiel

Pourtant la détermination de la position de travailleurs, si elle est possible dans ce cas, serait de toute façon d'un usage très lourd. En effet comme nous le montre l'exemple

(Figure 3-6), l'évolution des positions ne semble pas suivre une règle simple. La seule information due aux transferts ne suffit pas à retrouver l'ordre des travailleurs. Il faut en plus connaîttre les positions relatives des points de choix, c'est à dire la structure de l'arbre OU.

Actuellement, l'usage d'un graphe de précédence semble être la seule solution, ce qui se rapproche en fait des systèmes à mémoire commune [Hausman90], [Karlsson92], ou ce graphe est en fait l'arbre OU lui même. Mais ces systèmes utilisent en plus un algorithme pour vérifier la position d'un travailleur lors de la tentative d'exécution d'un effet de bord.
Comme il n'est pas raisonnable de garder un tel graphe en mémoire, uniquement pour retrouver l'ordre séquentiel, il faut utiliser l'arbre d'exécution. Mais sa répartition sur les différents travailleurs ne permet pas son utilisation efficace comme graphe de précédence, à cause des communications nécessaires.

Le cas du transfert d'un point de choix quelconque pose sensiblement les mêmes problèmes que celui du plus récent, et ne peut lui aussi être resolu simplement.

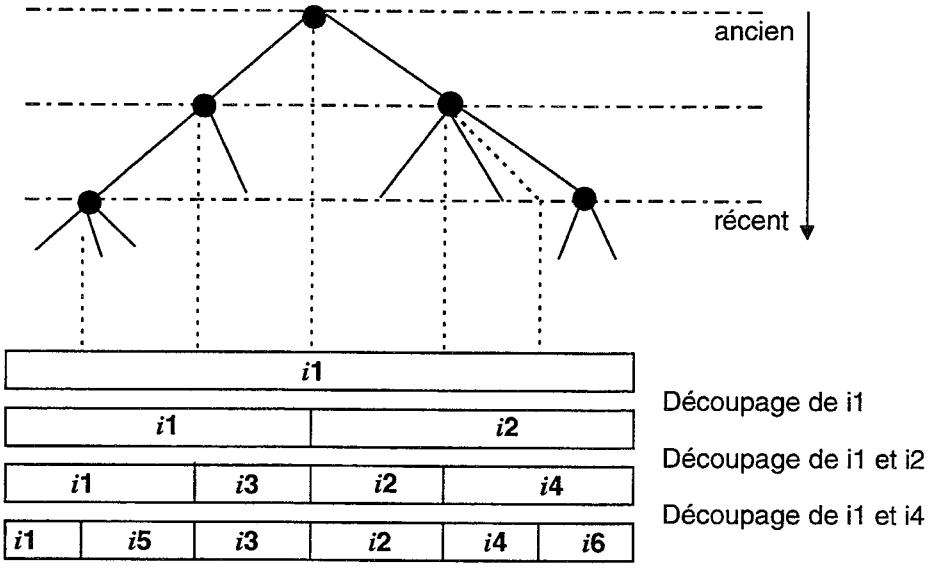
Plus ancien point de choix

Dans ce modèle multiséquentiel, lorsqu'un travailleur exporte du travail, il donne toujours ses plus anciens points de choix. Or les points de choix plus récents correspondent à un intervalle temporel plus proche, car ils seront parcourus avant les anciens d'après la politique d'exécution en profondeur d'abord et de gauche à droite de l'arbre. L'intervalle de temps séquentiel que représente le travail de l'exportateur sera donc situé avant celui que l'importateur va acquérir. Donc un travailleur important le point de choix le plus ancien sera à la suite de l'intervalle temporel de l'exportateur, comme pour le transfert du plus récent point de choix (Figure 3-5). Mais il existe une autre différence, qui revêt une grande importance. Le travail de l'importateur représente le dernier intervalle de temps séquentiel que possèdait l'exportateur. Il est donc parcouru, en séquentiel, toujours après le travail restant à effectuer sur l'exportateur. Le découpage de l'arbre d'exécution suit donc une loi simple :

⇒ Chaque transfert découpe un intervalle de temps en deux intervalles disjoints et ordonnés par rapport au temps séquentiel. Cet ordre est maintenu si les intervalles sont eux-même coupés.

L'application de cette loi sur un arbre complet d'exécution fournit un ensemble d'intervalles disjoints et ordonnés.

Lors de l'exécution parallèle d'un programme, il est possible à tout moment de connaître l'ordre des travailleurs selon leur position temporelle dans l'arbre d'exécution, en appliquant cette règle à chaque échange de travail, (Figure 3-8).



Le découpage d'un intervalle à partir de son plus ancien point de choix (nœud OU) permet de préserver l'ordre des intervalles.

Figure 3-7 - Intervalles de temps lors des transferts

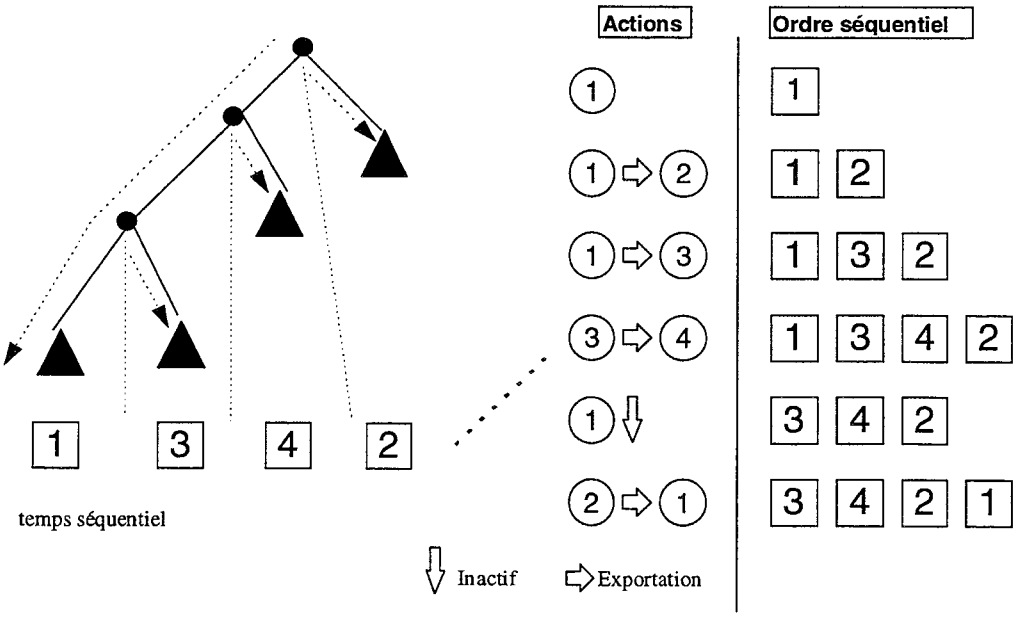


Figure 3-8 - Positions séquentielles des travailleurs

L'utilisation de ce principe est donc peu coûteuse dans ce cas, puisque la trace de l'ordre séquentiel peut être obtenue au niveau du processus d'ordonnancement à partir des seuls transferts de travail entre processeurs, sans autre forme de communication.
Une simple liste pourra donc représenter les positions relatives des travailleurs dans l'arbre d'exécution du programme. Un travailleur ayant importé des points de choix sera inséré immédiatement derrière celui qui les lui a fournis. Un travailleur n'ayant plus de travail sera retiré de la liste, jusqu'à un nouveau transfert.

La gestion de la liste sera adaptée aux différentes sortes d'effets de bords, mais elle constitue le support fondamental de la gestion séquentielle des effets de bord.

3.4 La coupure

La solution la plus simple pour exécuter une coupure est de parcourir séquentiellement les branches issues d'un nœud OU contrôlé par une coupure, la coupure est donc inhibitrice de parallélisme [Yasuhara84], [Ali90]. Cette forme n'est pas d'un vif intérêt pour accroître les performances, car elle limite le parallélisme, sous-exploitant alors les ressources disponibles.

Nous avons donc choisi une autre solution qui est déjà employée avec succès dans certains systèmes, comme Aurora [Hausman90a]. Elle préserve au maximum l'exécution parallèle, quitte à anticiper les calculs, même s'il y a un risque d'utiliser inutilement des ressources disponibles. L'exécution d'un nœud OU comportant une coupure est poursuivie de façon spéculative pour les alternatives qui sont dans le champ de la coupure. Le parcours d'une branche (alternative) dans la portée d'une coupure est appelé travail spéculatif, car il doit être annulé si la coupure est franchie. Si la coupure n'est pas franchie, cette branche devient valide et le travail correspondant devient nécessaire.

La gestion de la coupure au niveau du travailleur comporte deux phases. La première liée à la compilation est le maintien de la connaissance du caractère spéculatif de l'évaluation d'un point de choix et de l'arborescence qui en est issue. La seconde phase est le contrôle des points de choix spéculatifs exportés, afin de gérer le travail correspondant.

3.4.1 Détermination du travail spéculatif

Dans un système séquentiel, la gestion de la coupure est réalisée par l'utilisation d'une variable permettant de retrouver la position de la pile avant l'appel du prédicat contenant la coupure. En parallèle, le problème est qu'une portion de la pile peut avoir été transférée sur un autre travailleur. Il faut pouvoir alors retrouver celui-ci, pour invalider ou valider son travail selon que la coupure est franchie ou non. Cela exige une trace des points de choix exportés, pour pouvoir retrouver leurs nouveaux propriétaires lors de la gestion de la coupure. De même, un travailleur important des points de choix dans la portée d'une coupure, doit connaître le niveau de spéculativité du travail ainsi acquis.

Une première solution avait été donnée dans [Ali87] pour gérer la coupure en inhibant le parallélisme. Cette solution, simple dans son principe, prévoyait d'ajouter une marque pour indiquer qu'une coupure peut être franchie, et ce dans la pile des points de choix.

Ainsi, les points de choix empilés après la marque sont dans le champ de la coupure, et sont protégés d'une exportation par la marque. En plus de la marque, les points de choix doivent disposer d'un champ supplémentaire permettant d'indiquer si leur destruction supprime une marque ou non.
Si ce principe peut être adapté à la gestion du travail spéculatif, il introduit une nouvelle structure de donnée dans la pile, ce qui alourdit le mécanisme de contrôle de celle-ci par la machine abstraite, et l'emploi d'un champ supplémentaire dans le point de choix.

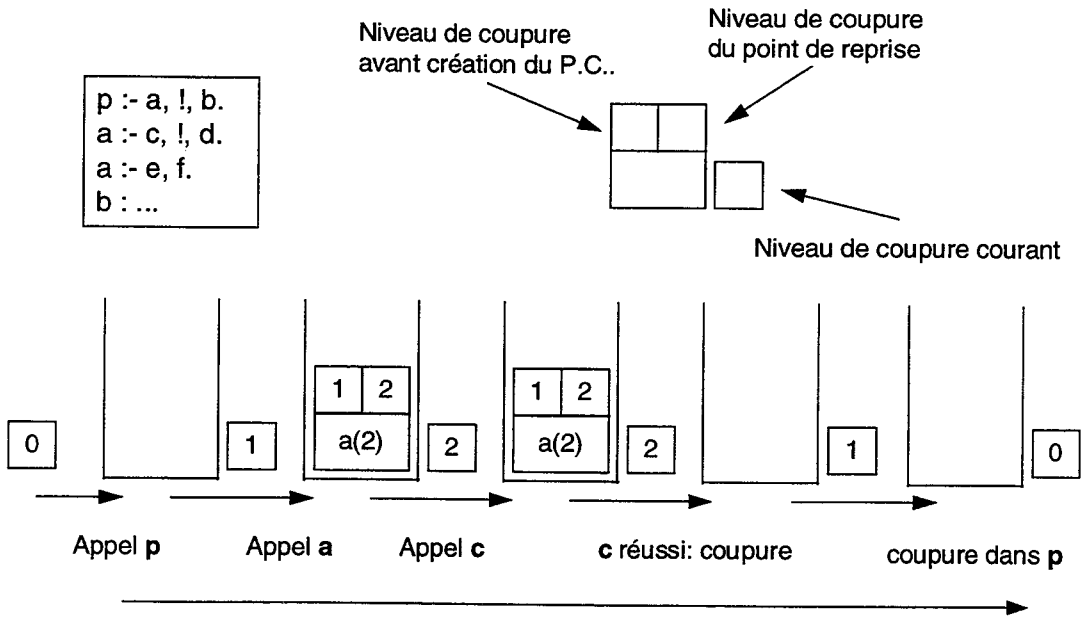


Figure 3-9 - Gestion des niveaux de coupure

Une autre méthode, issue de la gestion de la coupure dans Aurora [Hausman90c], est d'utiliser la phase de compilation pour déterminer les points de choix dans le champ d'une coupure. L'idée est de calculer lors de la compilation le nombre de coupures à venir dans les clauses des prédicats. Ainsi lors de l'exécution il est possible de maintenir une information sur la spéculativité d'une branche, à l'aide des compteurs de coupure.
Ainsi, lorsque le travailleur échange des points de choix, ceux-ci contiennent toutes les informations utiles pour connaître leur degré de spéculativité.

Ceci implique une légère adaptation de la gestion des points de choix pour les fonctions de la WAM. Une variable globale indique le niveau de coupure courant lors de l'exécution; et deux champs ajoutés au point de choix permettent la sauvegarde et la restauration du niveau de coupure lors de l'exécution et lors d'une exportation (voir Figure 3-9). Ces modifications ne changent pas le comportement séquentiel du moteur Prolog qui les ignore, et le surcoût associé reste faible.

Il faut alors modifier les instructions de la WAM liées aux points de choix pour tenir compte des champs additionnels. On doit aussi introduire deux nouvelles instructions (Figure 3-10) dédiées au comptage des coupures : *set_global_cut_level n* qui incrémente

de *n* le compteur de niveau de coupure global; et, *set_cut_level n* qui agît sur le compteur du niveau de coupure du point de reprise, champ du point de choix courant. Cette dernière instruction peut être intégrée dans les fonctions de gestion des points de choix (*try*, *retry*), à l'aide d'un paramètre supplémentaire, le nombre *n* de coupures dans la clause de reprise. Ces instructions sont générées lors de la phase de compilation, par analyse des coupures présentes dans chaque clause d'un prédicat.

Programme Prolog :

```
p :- q, !, r.
p :- s.
```

Code WAM simplifié correspondant :

```
E1 :  try_me_else E2          ; création du point de choix
      set_cut_level 1         ; niveau de coupure de la prochaîne alternative est augmenté de 1
      set_global_cut_level 1  ; niveau global de coupure est augmenté de 1
      allocate                ; reserver un environnement
      get_level Yn            ; sauvegarde du sommet de la pile des points de choix
      call q                  ; appel de q
      cut Yn                  ; la coupure est franchie, le niveau de coupure global est réduit de 1
      deallocate              ; libérer l'environnement
      execute r               ; appel de r

E2 :  trust_me                ; le point de choix est détruit
      execute s               ; appel de s
```

Figure 3-10 - Instructions WAM et niveaux de coupure

La compilation des fonctions d'optimisations (*switch*) impose une analyse des parcours possibles dans le code généré afin de générer correctement le code de gestion de la coupure. Cette étude n'est actuellement pas complètement résolue pour les cas les plus complexes. Dans le cas général, l'instruction *set_cut_level n* n'est parcourue que lorsqu'un point de choix est manipulé alors que *set_global_cut_level n* est parcourue à chaque entrée dans une clause contenant des coupures (Figure 3-11).

Ces nouvelles instructions n'ont donc pas d'effet sur l'exécution purement séquentielle, et servent uniquement à la gestion dynamique de la coupure lors de l'exécution parallèle du programme Prolog. Mais elles ne suffisent pas à procurer une exécution qui respecte la sémantique séquentielle. Il faut encore introduire d'autres moyens de contrôle.

3.4.2 Gestion dynamique de la coupure

La gestion de la coupure débute avec l'identification des points de choix potentiellement « coupés », puis par la gestion de messages appropriés pour la validation ou l'invalidation du travail issu d'un point de choix spéculatif.

```
Programme Prolog :

p(1) :- q, !, r, !, v.
p(2) :- s, !, t.
p(3) :- z.

Code WAM simplifié (pas de gestion des termes) correspondant :

S : switch_on_term (E1,C,fail,fail)          ; index suivant variable, constante, liste , foncteur
C: switch_on_constant  3, (1:E1,2:E2, 3:E3)  ; index sur les constantes (3 valeurs)

E1 :  try_me_else E2          ; création du point de choix
      set_cut_level 2         ; niveau de coupure de la prochaîne alternative est augmenté de 2
F1 :  set_global_cut_level 2  ; niveau global de coupure est augmenté de 2
      allocate                ; reserver un environnement
      get_level Yn            ; sauvegarde du sommet de la pile des points de choix
      call q                  ; appel de q
      cut Yn                  ; la coupure est franchie, le niveau de coupure global est réduit de 1
      call r                  ; appel de r
      cut Yn                  ; la coupure est franchie, le niveau de coupure global est réduit de 1
      deallocate              ; libérer l'environnement
      execute v               ; appel de v

E2 :  retry_me_else E3        ; le point de choix réutilisé
      set_cut_level 1         ; niveau de coupure de la prochaîne alternative est augmenté de 1
F2 :  set_global_cut_level 1  ; niveau global de coupure est augmenté de 1
      allocate                ; reserver un environnement
      get_level Yn            ; sauvegarde du sommet de la pile des points de choix
      call s                  ; appel de s
      cut Yn                  ; la coupure est franchie, le niveau de coupure global est réduit de 1
      deallocate              ; libérer l'environnement
      execute t               ; appel de t

E3 :  trust_me                ; le point de choix est détruit
      execute z               ; appel de z
```

Figure 3-11 - Optimisations de la WAM et niveaux de coupure

3.4.2.1 Identifier les points de choix

Dans le modèle multiséquentiel, les points de choix exportés par un travailleur ne lui sont plus utiles, aussi il est possible d'oublier leur emplacement. Mais lorsqu'il faut prendre en compte la coupure il est nécessaire d'identifier les points de choix exportés, ainsi que de mémoriser leur nouveau propriétaire, pour pouvoir propager l'effet du franchissement ou non de la coupure (invalidation/validation de travail).

L'identification du nouveau propriétaire peut être stockée dans un des champs du point de choix devenu inutile. Ainsi lors des différents transferts, les points de choix conservent les informations qui permettent de retrouver à tout moment le travailleur qui les gèrent.

Un point de choix exporté est donc laissé dans la pile des points de choix, mais il sera différencié d'un point de choix non-exporté. On peut par exemple utiliser le compteur de

niveau de coupure, avec pour convention qu'un nombre négatif pour indiquer un point de choix exporté (voir Figure 3-12). Ainsi son ancien propriétaire peut agir en conséquence lors d'un retour arrière, ou lors du franchissement d'une coupure, pour valider ou invalider le travail engendré par le point de choix exporté.

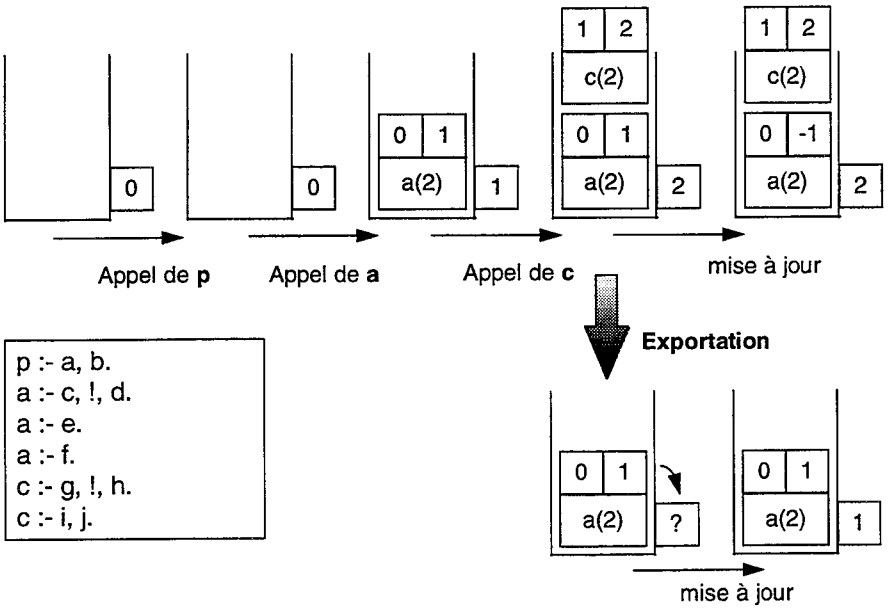


Figure 3-12 - Gestion parallèle des niveaux de coupure

Un point de choix sera considéré comme spéculatif si l'un de ses niveaux de spéculativité est non nul. Un travailleur exécutera du travail spéculatif s'il n'est pas le premier de l'ordre séquentiel, et que son plus ancien point de choix local est spéculatif. Si le travailleur est le premier de l'ordre séquentiel, son travail n'est jamais spéculatif, même s'il possède des points de choix spéculatifs.

3.4.2.2 Validation d'une branche

Le retour arrière est dit local s'il est effectué sur un point de choix non exporté, et global sinon. Si un travailleur effectue un retour arrière local, le fonctionnement de la machine Prolog est identique au mode séquentiel, il n'y a pas de traitement supplémentaire autre que la gestion des niveaux de coupure.
Lorsqu'un travailleur effectue un retour arrière global, le processus de gestion de la coupure doit effectuer un parcours descendant de la pile des points de choix (Figure 3-13) depuis l'emplacement du dernier point de choix local pour retrouver le dernier point de choix exporté (ou non local si ce travailleur n'a pas exporté). Si ce point de choix est spéculatif, il identifie alors son propriétaire, et envoie un message de validation pour ce point de choix. Dans le cas où plusieurs point de choix sont transférés à chaque échange, la validation se fait à partir du plus ancien.

La validation représente, pour le récepteur, la mise à jour des niveaux de coupures de tous les points de choix créés postérieurement à celui qui est validé, ainsi que la mise à jour du compteur global de niveau de coupure.

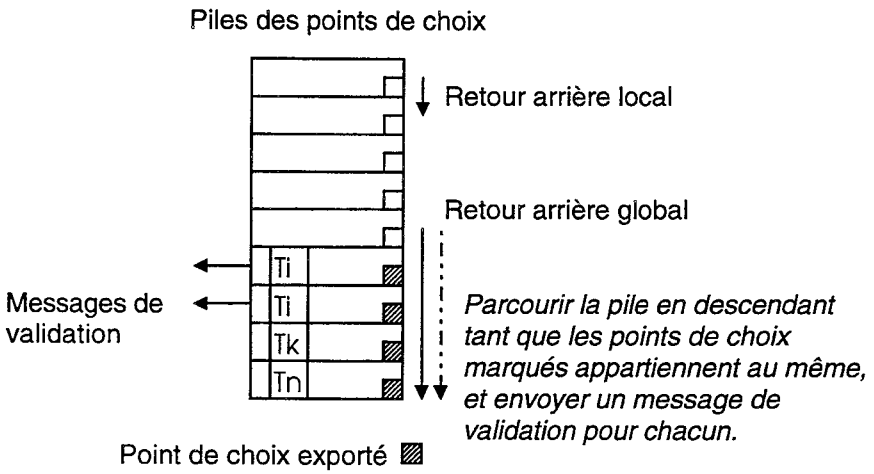


Figure 3-13 - Validation des branches spéculatives

Pour déterminer le nouveau degré de spéculativité d'un point de choix, il faut décompter le nombre de coupures qui n'ont pas été franchies. La validation ramène le niveau de spéculativité du point de choix, à celui qui était valable avant la création du point de choix. Ceci peut se déduire des deux informations mémorisées dans le point de choix, les niveaux de coupure, par simple soustraction. La valeur positive ou nulle obtenue est ensuite déduite dans tous les autres champs, et du niveau global de coupure.

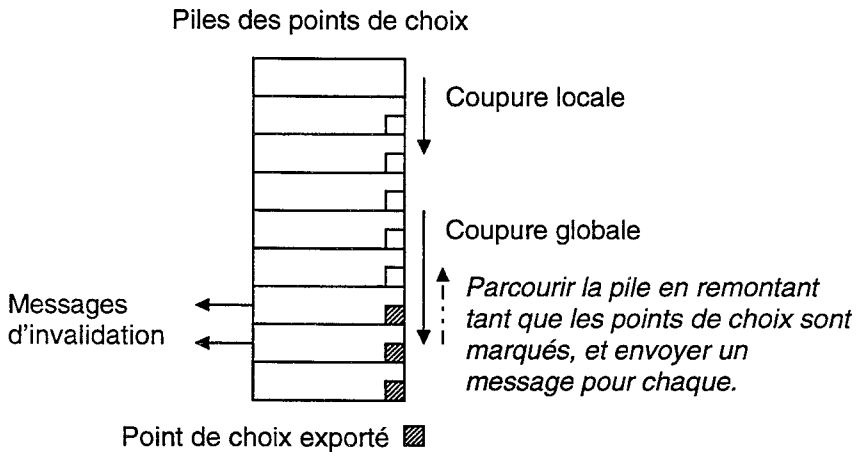


Figure 3-14 - Invalidation des branches

3.4.2.3 Invalidation d'une branche

Comme pour le retoure arrière, la coupure est locale lorsqu'elle aboutit à un point de choix non exporté, et globale sinon.

Si la coupure est locale, sa gestion se résume à une coupure séquentielle, plus les opérations de comptage des niveaux de coupure, ici réduites à la décrémentation du niveau de coupure courant.

Si la coupure est globale, la gestion comprend deux étapes, une phase locale et une phase à distance. La phase locale est identique à la coupure locale, sauf que le nouveau registre de point de choix pointe non pas sur le point de choix restauré, puisque celui-ci a été exporté, mais sur le bas de la pile des points de choix. La phase de contrôle à distance est un parcours de la portion de pile coupée contenant les points de choix exportés, avec envoi d'un message d'invalidation à leur nouveau propriétaire, pour chaque groupe de points de choix successifs coupés et exportés (Figure 3-14).

3.4.2.4 Le renvoi

Un travailleur peut importer des points de choix, puis plus tard exporter une partie de ceux-ci vers un autre travailleur. Dans ce cas, si l'exportateur reçoit un message de validation/invalidation, il doit le faire suivre vers le processus de gestion des coupures du nouvel importateur (voir Figure 3-15). Enfin, si un travailleur reçoit un message de validation, il devra aussi réactualiser les informations sur les niveaux de coupures mémorisées dans les points de choix, ainsi que pour le niveau global de coupure.

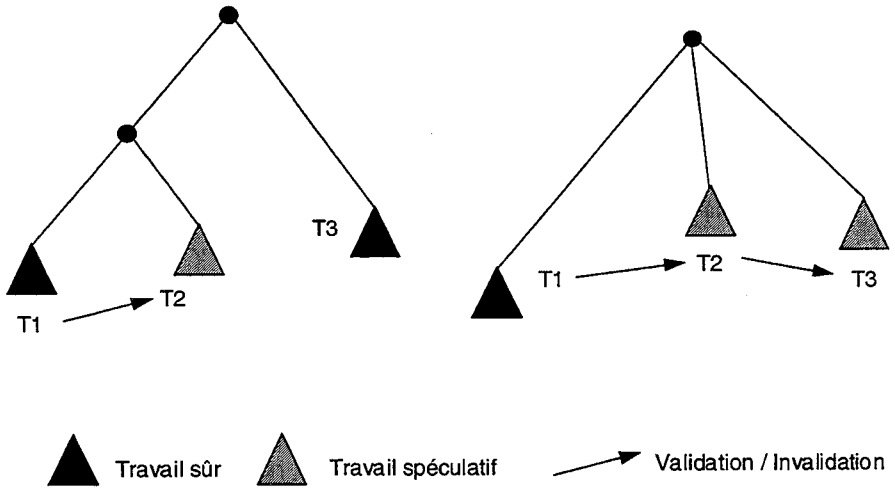
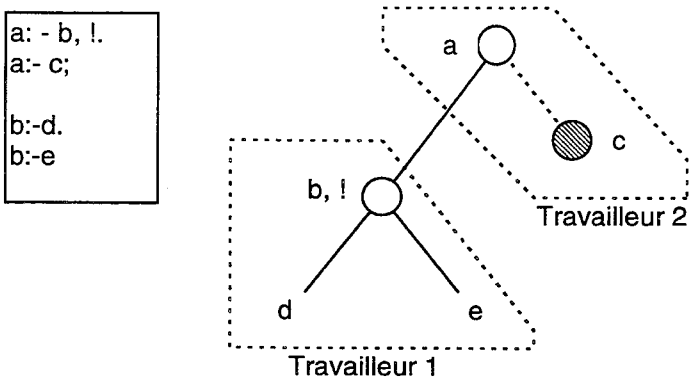


Figure 3-15 - Le renvoi de message d'invalidation ou de validation

3.4.2.5 Effets secondaire de la coupure spéculative

Il existe une interaction entre les différents effets de bord, et en particulier, avec l'utilisation de la coupure qui invalide l'existence d'une branche de l'arbre. Or cette branche qui n'aurait pas existé lors d'une exécution séquentielle, peut être parcourue lors d'une exécution parallèle. Aussi, les effets de bords, dont la coupure (Figure 3-16) doivent pouvoir être annulés s'ils ont été effectués dans une telle branche. Pour gérer l'invalidation des effets de bords il faut rendre ceux-ci réversibles, soit à l'aide d'un

mécanisme de restauration d'un état antérieur, soit en utilisant un cache que l'on pourra détruire si l'effet de bord n'est plus utile.



Lors d'une exécution séquentielle, la branche en trait pointillé n'existe que si l'appel de b échoue. En parallèle, cette branche peut être parcourue, mais sa validité dépend de l'échec ou non de b.Les evéntuels effets de bord doivent être suspendus ou être reversibles.

Figure 3-16 - Travail spéculatif et effets de bord

Pour la prise en compte de la coupure dans une branche nous proposons la mise dans un tampon des opérations globale de coupure. Celles-ci sont donc différées jusqu'à ce que la branche qui les contenait soit devenue non-spéculative, ou soit elle-même coupée. Dans ce dernier cas le tampons d'opération est simplement détruit.
Il est possible d'améliorer les décisions, pour couper de manière sûre une branche dans le cadre des coupures imbriquées [Hausman90c].

3.5 Les entrées et sorties

Nous présentons d'abord le pricipe général de gestion des entrées et sorties dans un système distribué, puis celle qui tient adaptée au respect de la chronologie séquentielle, et l'interaction avec la coupure.

3.5.1 Gestion des entrées et sorties

Il existe deux méthodes générales pour gérer les entrées et sorties du système parallèle, soit par un accès direct depuis chaque travailleur, soit par l'utilisation d'un serveur spécialisé [Carlsson90].

L'accès direct suppose à la base que le serveur d'entrées et sorties du système parallèle soit accessible depuis tous les nœuds, sinon cette méthode n'est pas réalisable. L'accès direct introduit un faible surcoût, mais nécessite tout de même la gestion de la cohérence des descripteurs de fichiers entre tous les travailleurs. Sur une machine sans mémoire commune cela se traduit par une diffusion des modifications des valeurs des différents descripteurs, opération très coûteuse en temps de communication.

Le passage par un serveur dédié introduit un surcoût de traitement pour chaque opération d'entrée ou sortie, et pire une communication entre le travailleur et le serveur. Heureusement, il est possible de diminuer le nombre de communication en groupant les requêtes au serveur, à la manière des entrées et sorties tamponnées du langage C par exemple.
Ainsi l'utilisation d'un serveur, méthode plus simple que l'accès direct, devient raisonnable en terme de performances, en permettant également le recouvrement des temps de communications par du calcul.

3.5.2 Ordre séquentiel et les entrées et sorties

Nous avons choisi de traiter les entrées et sorties à l'aide d'un serveur spécialisé, la tâche «console», qui permet l'interaction avec l'extérieur, et un processus dédié au sein de chaque travailleur (Figure 3-17). La console peut être placée sur une machine hôte, par exemple, le serveur de fichiers.

Le cas des entrées est simple, il faut être le plus à gauche dans l'arbre d'évaluation pour pouvoir lire une donnée. Le traitement des entrées se résume donc à un test pour savoir si le travailleur est le plus à gauche. Si c'est le cas il peut immédiatement effectuer sa requête d'entrée, sinon il sera suspendu jusqu'à ce que l'ordonnanceur lui signale qu'il est le premier dans l'ordre séquentiel (le plus à gauche).
Pour les sorties, la gestion la plus simple consiste encore à différer leur exécution tant que le processeur n'est pas le « plus à gauche » de l'arbre d'exécution. Mais en fait, comme une sortie n'est pas une opération bloquante, l'attente introduite est inutile. La solution consiste à enregistrer l'opération de sortie pour l'exécuter de façon différée « au bon moment » séquentiel et de laisser continuer la résolution si possible.
Le travailleur effectue alors toutes ses sorties sur un tampon local. Lorsque celui-ci est plein, et que le travailleur est le premier de la liste séquentielle, les sorties sont transférées au processus console, puis le tampon est réinitialisé. Si le travailleur n'est pas le premier, le tampon est agrandi.

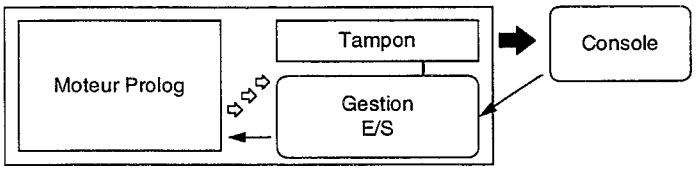
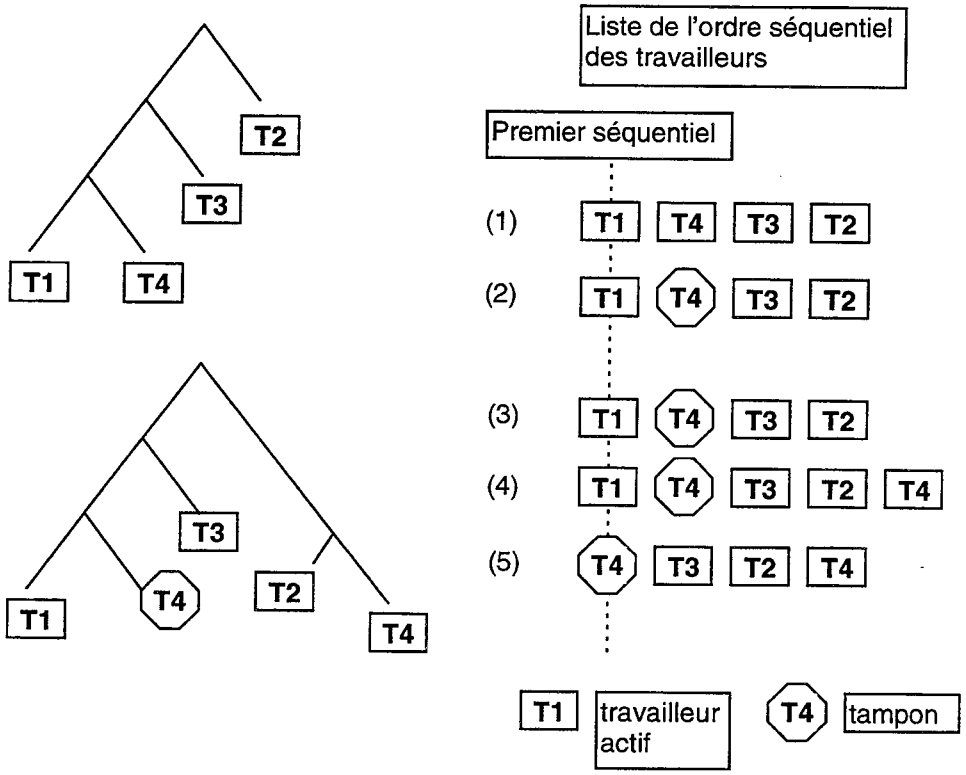


Figure 3-17 - Découplage des entrées et sorties

Dans le cas où un travailleur n'a plus de travail, et qu'il a effectué des sorties, il faut préserver celles-ci, afin de les placer correctement dans le flux de sortie pour respecter l'ordre séquentiel. Pour cela, la liste de maintien de l'ordre séquentiel gère un nouvel élément correspondant à un tampon de sortie, ou marque la structure pour la différencier du travailleur actif (Figure 3-18). Cet élément remplacera celui identifiant le travailleur, qui lui sera ôté de la liste, car le travailleur associé est devenu inactif. Puis, lorsque

l'élément de liste correspondant au tampon sera devenu le premier de la liste, l'ordonnanceur pourra envoyer un signal au processus de gestion des entrées/sorties du travailleur, qui videra alors ce tampon. Enfin, l'élément de liste associé sera ôté de la liste.



Dans cet exemple, les travailleurs sont ordonnés séquentiellement T1, T4, T3 et T2. Lorsque T4 n'a plus de travail (2), il est retiré de la liste séquentielle, mais on conserve son emplacement pour effectuer ses E/S au bon moment.
Quand T4 retrouve une part de travail (4), il est réintroduit dans la liste immédiatement à la droite de son fournisseur.
Enfin en (5) son tampon d'E/S se retrouve premier de l'ordre séquentiel, il est vidé puis retiré à son tour de la liste.

Figure 3-18 - Liste séquentielle avec entrées et sorties

3.5.3 Interaction avec la coupure

Comme nous l'avons introduit en 3.4.2.5, la coupure a une influence sur le traitement des autres effets de bord. Pour les entrées et sorties, il s'agît de pouvoir les annuler.
Dans le cas d'une entrée, comme le travailleur doit déja attendre d'être le premier de l'ordre séquentiel, il n'y a pas de problème. En effet, au moment où il pourra effectuer son entrée, il ne pourra pas être dans une branche spéculative car il sera le premier séquentiel.

Dans le cas des sorties, il faut pouvoir annuler les opérations que l'on a déjà effectuées. Ceci est relativement simple dans notre cas, car nous avons simplement différé ces opérations. Il suffit donc de ne pas les effectuer. Le tampon associé à celles-ci est simplement détruit.

3.6 La base des prédicats

Les prédicats de modification de la base des prédicats ne peuvent être implantés de manière implicite et efficace. En effet aucune branche ne peut être parcourue avec certitude tant qu'une branche plus à gauche qu'elle est susceptible de modifier la base des prédicats. Le choix de son implantation devient donc une sorte de compromis entre efficacité et conformité du langage.

Si on veut être conforme à la norme, le système deviendra très complexe et perdra en performance, car toute partie du programme sera potentiellement modifiable, et donc aucun travail effectué en partie droite d'un autre ne sera jamais fiable. Il faudrait gérer un système de reprise de travail pour chaque travailleur en partie droite de l'arbre, dès qu'une modification de la base les concerne potentiellement.

Si on limite aux seuls prédicats déclarés comme modifiables les opérations sur la base des prédicats, il devient plus simple de gérer cette base. Cette restriction est communément admise aussi par le standard en cours de définition car elle permet souvent d'optimiser le code séquentiel. La gestion des prédicats dynamiques est alors décomposée en deux phases, la modification d'un prédicat et son accès.

Modification des prédicats dynamiques

Il existe plusieurs possibilités pour gérer les répercussions de la modification de la base des prédicats dans un système parallèle :

- Mise à jour immédiate, dès qu'un travailleur modifie la base des prédicats, il doit le signaler aux autres travailleurs du système qui vont alors répercuter immédiatement ces modifications.

- Mise à jour différée, lorsqu'un travailleur modifie la base des prédicats, il le signale aux autres travailleurs qu'après un certain délai. De ce fait si une autre modification intervient, elle sera groupée avec la première, et ainsi de suite. De plus les travailleurs ne modifient leur base que lorsqu'ils sont inactifs, ou qu'ils accèdent à un prédicat dynamique.

- Mise à jour paresseuse, il s'agît d'éviter au maximum les communications, en groupant les informations de modifications en prenant un délai infini pour la méthode différée. Les modifications ne sont répercutées que sur une demande des travailleurs accédant à un prédicat dynamique, ou éventuellement inactif.

Accès à un prédicat dynamique

Il existe deux actions possibles lors de l'accès à un prédicat dynamique :

- L'attente de l'ordre séquentiel
  La solution la plus simple est la classique attente, pour que le travailleur soit le premier dans l'ordre séquentiel. Cependant elle peut engendrer une perte notable de performances.

- L'utilisation de travail spéculatif
  Comme pour la coupure, on peut supposer que l'état de la base des prédicats n'aura pas changé lorsque le travail en cours sera le premier de l'ordre séquentiel. On continue donc de manière spéculative la résolution en cours. Cependant, contrairement à la coupure où le travail invalide doit uniquement être supprimé, ici il doit être repris si la base des prédicats est modifiée pour le ou les prédicats dynamiques rencontrés. Cette gestion est bien plus complexe que la coupure, et nécessite la copie de l'état de la machine Prolog pour chaque supposition faite sur l'état d'un prédicat dynamique. Si aucun n'a été modifié dans la base, il suffit de détruire les différentes copies d'états de la machine. Par contre, si un ou plusieurs prédicats dynamiques ont été modifiés, il faut restaurer l'état de la machine abstraite au plus ancien état correspondant au passage d'un prédicat dynamique modifié, et reprendre ensuite le travail. L'inconvénient de cette technique est sa mise en œuvre très lourde et l'absence de certitude quant à l'accroissement des performances en résultant.

La solution qui nous semble la plus réaliste à court terme est l'emploi de la mise à jour différée associée au principe de suspension.
La gestion de la partie séquentielle est fonction de l'implantation du moteur Prolog. Un serveur de mise à jour et diffusion est ajouté pour la gestion parallèle. Et d'éventuelles optimisations comme le traitement de la partie à droite seulement du travailleur effectuant une modification de la base, et l'utilisation de la mise à jour paresseuse, peuvent par la suite intervenir.

Pour maîtriser l'interaction avec la coupure, la méthode choisie est la même que pour les prédicats d'entrées et de sorties. Il suffit d'utiliser un tampon pour tous les messages de mise à jour que l'on ne devra envoyer que si le travail en cours n'est pas spéculatif. Pour la suspension lors de l'appel d'un prédicat dynamique, il n'y a pas de traitement particulier, car le prédicat ne peut être franchi que si le travailleur est le premier de l'ordre séquentiel.

3.7 Les prédicats collecteurs

Il existe deux méthodes pour implanter les prédicats collecteurs en séquentiel, la première repose uniquement sur Prolog, et la seconde demande une légère extension des prédicats prédéfinis. Nous rappelons rapidement les principes respectifs, puis nous présentons la méthode retenue pour l'évaluation parallèle des prédicats collecteurs.

3.7.1 Choix de la méthode d'implantation

Le principe simplifié d'exécution d'un prédicat collecteur, est de générer la liste des solutions d'une requête, à partir du mécanisme normal d'exécution de Prolog, en profondeur d'abord et gauche à droite de l'arbre d'évaluation. Cette génération est

obtenue en forçant un retour arrière après chaque instance résolue de la requête. Il suffit alors de créer la liste des solutions lors de chaque succès de la requête. Mais apparaît à ce moment le problème suivant : lors du retour arrière, toutes les liaisons effectuées sont détruites et on perd ainsi les solutions que l'on voulait garder. Il faut donc pouvoir construire la liste des solutions dans un espace qui survivra au retour arrière. Si l'on veut garder intact le mécanisme général de Prolog, on utilisera la base des prédicats comme espace de stockage, avec cependant tous les inconvénients liés. Par contre, si on accepte de modifier le fonctionnement de Prolog, on utilisera deux nouveaux opérateurs dont le but est de créer ou de détruire une variable permanente.
Pour préserver au maximum les performances, nous avons choisi d'utiliser une variable permanente associée à un tampon spécialisé, et la définition de quelques prédicats spécifiques, dans le même esprit que celle de Muse [Karlsson92].

3.7.2 Principes de l'exécution parallèle

Après avoir rappelé les solutions déjà apportées par les systèmes à mémoire commune, nous présentons l'adaptation de l'une d'elle à notre cas.

Solutions classiques

En général, lorsqu'un système parallèle gère les prédicats collecteurs, il suspend le calcul des branches qui ne sont pas le plus à gauche du sous-arbre de la collection. Cette méthode simple ne procure pas de très bonnes performances, car elle limite le parallélisme exploitable, et impose des changements de tâches aux travailleurs suspendus lorsqu'ils acquièrent un nouveau travail.

Une méthode plus performante a été introduite dans le système Muse[Ali93]. Le calcul des solutions se déroule sur plusieurs travailleurs, qui vont construire des listes de solutions partielles.
Les prédicats collecteurs créent toujours un point de choix avec deux alternatives, la première créant le calcul de la collection, la seconde récupère la liste des solutions.

La méthode de rassemblement des solutions est la suivante :

1) Une solution trouvée dans une branche la plus à gauche est attachée au nœud du prédicat collecteur.
2) Une solution trouvée dans une branche qui n'est pas la plus à gauche, est attachée au nœud origine de la branche, avec le numéro d'ordre de la branche. Les solutions sont enregistrées en accord avec la stratégie « en profondeur d'abord » et de « gauche à droite ».
3) Quand une branche la plus à gauche est complètement parcourue, les étapes (1) et (2) sont appliquées aux solutions de la nouvelle branche la plus à gauche.

Dans notre cas, il n'est toujours pas possible de partager les informations en cours de traitement. Mais le principe des solutions partielles reste valable, il faudra seulement

transférer celles-ci vers le travailleur le plus à gauche, celui qui a généré la collection.

Solution sans mémoire commune

Un travailleur exécutant un prédicat collecteur va créer un tampon associé dans lequel la liste des solutions trouvées sera copiée. Si l'exécution du prédicat reste locale au travailleur tout se passe comme avec un système séquentiel.

Par contre si une partie du travail est exportée vers d'autres travailleurs (coopérants), le processus de gestion des collections sera utilisé. Ainsi, chaque travailleur coopérant à la résolution de la collection, mémorisera une liste partielle des solutions. Lorsqu'un travailleur coopérant finit sa part de travail, un prédicat spécial est appelé, et celui-ci provoquera l'appel d'une fonction de transfert de la solution partielle vers le processeur origine de la collection, via son unité de gestion des collections. Le travailleur coopérant pourra immédiatement reprendre du travail dans une autre partie de l'arbre, la solution partielle étant maintenant gérée indépendamment. Si le processeur origine de la collection épuise tout son travail, il doit alors attendre de recevoir toutes les listes partielles, avant de pouvoir reprendre son exécution. Il est possible d'anticiper la reprise du travail, soit pour classer les solutions déjà arrivées, pour un appel de *setof*, soit pour commencer à consommer les solutions déjà reçues, pour un appel de *bagof* ou *findall*.

L'implantation efficace des prédicats collecteurs est importante, surtout si l'on veut simplifier la génération de programmes parallèles, où l'on peut utiliser les prédicats collecteurs comme expression du parallélisme [Tarau95].

Les performances du prédicat *setof* pourraient être améliorées par l'utilisation d'algorithmes de tri parallèle de la liste des solutions. Ce qui peut être l'objet d'études futures sur le système PLoSys.

3.8 Influence sur la régulation de charge

La gestion des effets de bord avec le respect de la sémantique séquentielle de chacun introduit quelques contraintes au niveau de la régulation de charge. Nous présentons brièvement certaines solutions envisageables, mais il reste un travail expérimental important à ce sujet. Dans les systèmes existants, en particulier pour les architectures à mémoire partagée, quelques principes déjà ont été donnés [Beaumont93], [Carlsson90]. Néanmoins, peu d'études ont été menées sur des systèmes Prolog parallèles sans mémoire commune gérant les effets de bord, et aucune solution ne traite complètement du problème ainsi que de la politique de régulation de charge associée.

3.8.1 La coupure

Pour la gestion de la coupure lors de la régulation de la charge du système, deux modes peuvent être employés :

Inhibition du parallélisme

Cela simplifie la gestion de la coupure, et de la régulation de charge, qui se contente de travailler comme en Prolog pur. Mais pour éviter de bloquer trop de ressources par l'inhibition du parallélisme, il faut allouer le travail dans la partie la plus à gauche de l'arbre. Mais ce comportement semble mauvais pour des problèmes équilibrés, ou dont l'arbre se développe plus sur la droite, ou en largeur.

Gestion du travail spéculatif

Il existe plusieurs possibilités pour la prise en compte du travail spéculatif lors de la régulation de charge [Hausman90], [Beaumont93] :

- La première est de ne pas distinguer le caractère spéculatif. La gestion en est simplifiée, mais le risque d'inefficacité devient grand. Beaucoup de travailleurs peuvent alors consommer du travail spéculatif qui sera ensuite détruit, et ne participeront donc pas à l'accroissement des performances.

- La deuxième suppose que la prise en compte de la coupure au niveau du processus d'ordonnancement se résume à la connaissance du niveau de «spéculativité» du travail disponible. Pour diminuer au maximum le nombre de messages échangés, on peut envisager de ne modifier que le message d'information de la charge, en affectant par exemple un poids moindre au travail spéculatif. Cela peut être réalisé à l'aide d'un coefficient. Ce système permet d'allouer du travail spéculatif dans le cas où la quantité de travail sûr est faible, et favorise le reste du temps le travail sûr. Le principal problème est de déterminer correctement le coefficient. Il se peut que celui-ci dépende de la nature du programme qui est exécuté, ou bien de la phase d'exécution (début, milieu, fin) par exemple. Un coefficient égal à zéro inhibe le parallélisme, tandis qu'un coefficient égale à un le rend équivalent au travail fiable.

- Enfin, une autre possibilité est de favoriser systématiquement l'allocation de travail réduisant le caractère spéculatif d'un sous arbre, conduisant au choix de travail principalement en partie gauche de l'arbre d'exécution. Cette selection peut être indépendante du caractère spéculatif de l'arbre

Si l'on ne désire pas utiliser le travail spéculatif, il suffit d'interdire l'exportation de point de choix dans la portée d'une coupure, ce qui correspond à l'absence de gestion du travail spéculatif. Un travailleur ne pourra exporter des points de choix que s'ils sont hors du champ d'une coupure (niveau de coupure égal à 0). Cette solution peut être simplement réalisée à l'aide de la partie de la fonction de régulation de charge locale au travailleur qui peut simuler, dans le cas de la coupure inhibitrice, l'absence de charge utile sur ce travailleur, l'empêchant ainsi d'exporter une branche spéculative.

3.8.2 Les entrées et sorties

Il n'y a pas de problème particulier d'interaction pour les sorties, du moins si les tampons sont régulièrement répartis. Sinon, un travailleur risque d'épuiser son espace mémoire s'il

possède trop de tampons en attente d'être vidés. Par contre pour les entrées, qui sont bloquantes, il faut éviter que trop de travailleurs se bloquent sur des requêtes d'entrées. Pour cela il est encore judicieux de favoriser le travail situé à gauche des travailleurs en attente d'une entrée, afin de pouvoir les rendre actif plus rapidement.

3.8.3 Les prédicats collecteurs

Normalement le travail effectué lors d'une collection est identique au travail courant, et peut être géré de la même façon. Cependant, si le système intègre un algorithme de tri parallèle pour implanter le prédicat *setof*, il devient intéressant de distribuer le travail sur plusieurs processeurs afin de bénéficier aussi d'une accélération au niveau du tri. Chaque liste partielle peut être triée localement avant la jointure.
Il est donc possible d'adjoindre au traitement prédicat *setof* un contrôle plus fin de la charge, voire d'associer un paramètre de la fonction de régulation de charge pour sa prise en compte.
De même, si une collection se trouve en partie gauche de l'arbre, il devient plus intéressant de lui attribuer plusieurs processeurs, car il existera peu d'attente due aux effets de bord dans cette partie, et cela permettra de réduire le caractère éventuellement spéculatif de la partie droite de l'arbre.

3.8.4 La base des prédicats

La modification de la base des prédicats impose des contraintes fortes de synchronisation, aussi il faut limiter au maximum leur influence en favorisant le travail en partie gauche de l'arbre. Il existe deux cas d'interaction, la modification de la base et l'appel d'un prédicat dynamique.

Pour la modification de la base, la fonction de régulation de charge n'a pas d'information particulière. Par contre, lors de l'appel de prédicats dynamique le problème est différent, car il peut introduire une nouvelle forme de travail spéculatif lorsque la machine abstraite évalue un prédicat dynamique et que le travailleur n'est pas le plus à gauche de l'arbre.
En considérant que le programme contienne la déclaration des prédicats dynamiques, il est possible au niveau de la régulation de charge de délimiter des sous-arbres correspondant à des parties spéculatives par rapport aux prédicats dynamiques. En évitant de donner du travail provenant de ces parties, on peut espérer éviter de perdre des ressources sur des calculs spéculatifs. Là aussi, il est possible d'associer un coefficient à cette charge spéculative, avec les mêmes limitations que pour la coupure.

3.8.5 La suspension de travail

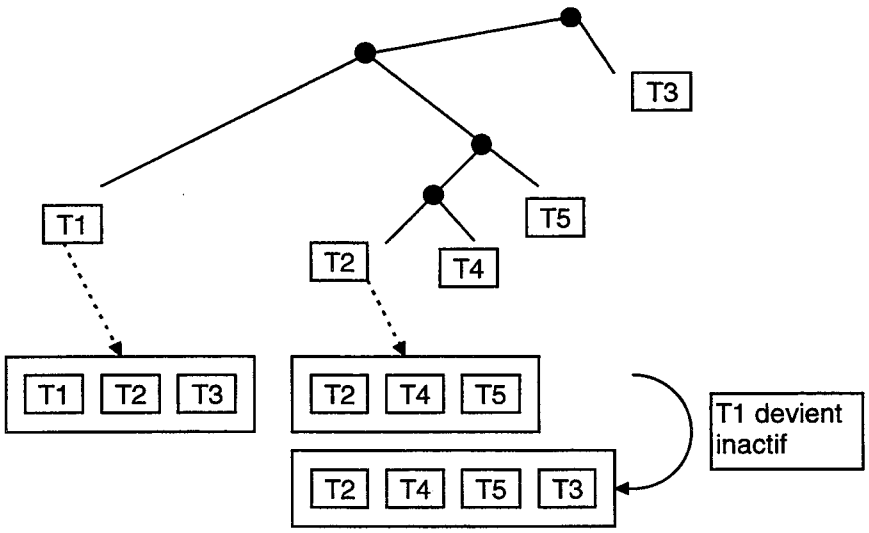
Un travailleur peut se trouver bloqué dès qu'il recontre un prédicat à effet de bord. A moins qu'ils ne puissent être différés, les prédicats à effet de bord qui sont pas dans la branche la plus à gauche de l'arbre, ou qui sont dans une branche spéculative sont suspendus tant que la branche n'est pas la plus à gauche ou reste spéculative. Il est envisageable de fournir une autre partie de l'arbre aux travailleurs ainsi bloqués. Cependant cette méthode augmente le nombre de transferts de données entre les travailleurs et suppose qu'il est possible de gérer plusieurs exécutions séquentielles avec

un travailleur. En effet si avec une mémoire commune tous les travailleurs peuvent reprendre un travail suspendu car les informations sont partagées, il n'en est pas de même sans mémoire commune.

Une solution simple à ce problème est le recouvrement des nœuds d'un système par un plus grand nombre de travailleurs, plaçant ainsi plusieurs travailleurs par nœud. Dès lors, quand un travailleur est bloqué, il suffit d'allouer du travail à un autre travailleur sur le même nœud.
Pour être sur qu'un seul travailleur s'exécute sur un nœud il faut toujours allouer du travail en partie gauche d'un travailleur bloqué. Le nouvel élu aura donc fini l'évaluation de son sous-arbre avant que le travailleur bloqué ne soit de nouveau activé. L'idéal est bien sur d'allouer la portion de l'arbre précédant immédiatement celle parcourue par le travailleur bloqué.

3.9 Extensions vers un système totalement distribué

Le système PLoSys doit à long terme fonctionner d'une manière totalement distribuée, ce qui signifie que la tâche ordonnanceur devra disparaître au profit d'une fonction de régulation répartie sur tous les travailleurs.



Les travailleurs sont ordonnés par des listes d'ordre séquentiel locales. Dans cet exemple, seuls T1 et T2 ont exporté des point de choix. Lorsque T1 devient inactif, sa liste est envoyée à T2, qui la fusionne avec sa propre liste.

Figure 3-19 - Principe du maintien de l'ordre séquentiel en distribué

Au niveau de la gestion des effets de bord, cela implique que la gestion de l'ordre séquentiel, point central du mécanisme de gestion des effets de bord, soit modifiée afin de pouvoir fonctionner correctement.

Le maintien de l'ordre séquentiel, qui était trivial avec un ordonnanceur centralisé; devient un peu plus complexe en tout distribué. Cependant le principe de base peut être préservé localement à un travailleur, au prix d'une gestion supplémentaire des références sur d'autres travailleurs. En effet lors d'un échange de travail il est toujours possible d'ordonner les deux travailleurs participant à cet échange. On peut donc maintenir un ordre local, l'ordre global sera donc déduit de celui-ci (Figure 3-19).

Chaque travailleur gère une liste dont il est l'élément de tête, et qui contient l'ordre séquentiel déterminé à partir de ses exportations. Cette liste est gérée de la même manière que la liste actuelle, sauf qu'elle est locale au travailleur. Chaque élément de la liste est un travailleur ayant importé du travail depuis le propriétaire de la liste.
Si le travailleur devient inactif, et qu'il a effectué des exportation, il doit envoyer sa liste à son suivant dans la liste. Celui-ci effectue alors une fusion sa propre liste dans la liste reçue. Pour cela il remplace l'élément l'identifiant dans la liste reçue par sa propre liste de travailleurs.

Il reste cependant à gérer les cas où les messages se croisent, il faut donc encore définir un protocole fiable, mais le pricipe semble pouvoir convenir pour maintenir l'ordre séquentiel dans un système distribué.

Répercutions sur la régulation de charge

Il est possible de distribuer le processus d'ordonnancement au niveau de la régulation de charge, il existe maintenant des algorithmes distribués efficaces [Benjumea93], [Kacsuk94]. Par contre la gestion distribuée des effets de bord n'est pas encore résolue, bien que quelques solutions non implantés aient été données [Kacsuk93]. Avec quelques modifications au niveau des paramètres ceux-ci pourraient être utilisé pour PLoSys dans le cadre d'une exploitation entièrement distribuée.
Mais il n'existe pas encore d'algorithmes distribués qui prennent en compte les effets de bord de Prolog. Les seuls algorithmes de régulation de charge gérant les effets de bord, ne sont implantés et validés que pour les architectures à mémoire partagée [Carlsson90], [Hausman90c], [Karlsson92].
De plus il faut arriver à gérer les effets de bord avec une connaissance partielle de l'état du système si l'on ne veut pas avoir recours à un nombre trop important de communications. Le fonctionnement entièrement distribué de Prolog offre donc encore de nombreux points d'étude, surtout pour une gestion efficace et implicite des effets de bord.

4. Le système PLoSys

Ce chapitre introduit les principaux objectifs du projet PLoSys, ainsi que l'organisation générale du système. Ensuite, nous précisons le choix du langage supporté, et le type de machines cibles. Puis, nous détaillons l'environnement que doit fournir le projet. Enfin, nous présentons les techniques et l'organisation retenues pour réaliser le système d'exécution parallèle.

4.1 Objectifs et organisation

Le projet OPERA [Favre92] a permis de démontrer que l'implantation OU-parallèle de Prolog sur des machines sans mémoire commune était réaliste. Mais le langage Prolog supporté était restreint au Prolog pur et ne comportait donc pas d'effet de bord. Il n'existait pas non plus d'environnement de programmation parallèle associé au système Prolog. Ce système était intimement lié à l'architecture matérielle cible et n'était donc pas facilement portable. Cependant l'expérience acquise lors de ce projet fut utile pour établir le projet PLoSys, et surtout pour éviter les écueils d'une implantation trop dépendante du matériel.

Les choix du modèle d'exécution et de gestion du parallélisme ne sont pas novateurs, et reprennent en majeure partie les choix faits dans le projet OPERA. Cela permet de disposer d'une base de travail aux solutions techniques éprouvées, du moins pour la partie Prolog pur. Par contre, le projet PLoSys intègre dès la phase de conception le support des effets de bord, en assurant la compatibilité avec l'exécution séquentielle d'un programme, ainsi qu'un environnement de programmation basé sur Prolog. De plus son architecture logicielle doit lui permettre d'être porté facilement d'une machine parallèle à une autre, ainsi que sur un réseau de stations de travail.

Enfin, le projet vise à moyen terme, une implantation totalement distribuée ainsi que différentes extensions, comme des opérateurs plus spécifiques au parallélisme ou l'intégration de la programmation à l'aide de contraintes. Pour cela la conception du système est modulaire, simplifiée au maximum, sans pour autant négliger les performances. Néanmoins, la capacité d'évolution est privilégiée par rapport aux optimisations complexes.

4.1.1 Choix du langage supporté

En raison de la nature logique de son expression, Prolog est un bon candidat à la parallélisation automatique. Cependant, il faut assurer une compatibilité entre les différentes exécutions de programmes Prolog, qu'elles soient séquentielles ou parallèles, afin de pouvoir profiter de la bibliothèque des logiciels déjà existants, ainsi que pour développer de nouvelles applications portables. De même, si l'on veut permettre à l'utilisateur d'employer un environnement de développement séquentiel performant et déjà éprouvé, la compatibilité séquentielle est donc nécessaire.

Or, les prédicats extra-logiques compliquent le passage à l'exécution parallèle implicite car leur sémantique est intimement liée à l'exécution séquentielle d'un programme. Mais si elle est bien définie par un standard, alors il devient possible de réaliser ces

prédicats en version parallèle en respectant leur sémantique séquentielle. Cette étape, présentée au chapitre précédent, n'est pas triviale, mais permet de garantir la compatibilité entre les systèmes Prologs existants, séquentiels ou parallèles.

Pourtant, de nombreux systèmes disponibles actuellement ne respectent pas cette contrainte de compatibilité, et proposent de nombreuses extensions ou modifications au langage Prolog, réduisant la portabilité des applications. L'utilisation du parallélisme la réduit encore par l'introduction de nouvelles sémantiques pour certains prédicats du langage, comme dans Aurora [Carlsson90], avec les effets de bord asynchrones, ou comme la coupure « *cavalier cut* » [Hausman90c], réduisant finalement l'intérêt de Prolog comme langage d'expression simple et portable du parallélisme. Reste que pour de nombreux domaines d'utilisation de Prolog, le parallélisme offre un moyen efficace d'accroissement des performances, même au prix d'une certaine incompatibilité avec le séquentiel.

Comme un objectif important du projet est de permettre l'exécution parallèle d'un programme Prolog séquentiel, avec le minimum d'interventions (modifications), de la part du programmeur. La plupart des extensions, Entrées et Sorties, coupure, prédicats collecteurs, fonctionnent de manière identique à une exécution séquentielle, sans aucune modification apportée au programme. Cependant, certains prédicats, comme ceux gérant la base des prédicats (*assert*, *retract*, ...), ne peuvent être implantés efficacement en parallèle, car leur gestion implicite est très coûteuse, comme on pourra le voir dans le chapitre suivant. Aussi une directive doit tout de même être ajoutée au langage pour permettre essentiellement de renseigner le système parallèle afin d'accroître ses performances. Celle-ci est déjà employée pour les systèmes séquentiels performants, et sert à déclarer un prédicat comme dynamique.

4.1.2 Machines cibles

De nombreux systèmes Prolog ont été développés sur des machines parallèles avec mémoire commune, avec un succès notable au niveau des performances, comme Aurora [Carlsson90] ou Muse [Karlsson92]. Mais, ce type de machines ne peut comporter qu'un nombre relativement limité de processeurs, et est difficilement extensible. Aussi pour disposer d'encore plus de puissance, il est nécessaire d'utiliser des machines parallèles sans mémoire commune, qui permettent de disposer d'un plus grand nombre de processeurs. Enfin l'augmentation de la puissance de ces machines est réalisée par l'ajout de nœuds (processeur et mémoire), modification généralement sans influence du point de vu de l'utilisateur ou du programmeur.

Par ailleurs, les réseaux de stations de travail peuvent aussi être considérés comme des machines distribuées, dont les performances sont semblables pour le calcul, mais dont les coûts de communications sont généralement plus importants, car ils ne disposent pas de réseaux de communication spécialisés. Les ressources disponibles sont nombreuses parmi les réseaux de stations, car leur taux d'utilisation est souvent faible, mais elles sont souvent dispersées géographiquement. Cette assimilation est rendue possible actuellement par l'utilisation des mêmes microprocesseurs généralistes dans les machines parallèles et dans les stations de travail, comme le microprocesseur de type

Power d'IBM, qui est employé aussi bien dans des stations de travail que dans les machines parallèles SP/1 et SP/2.
En plus, il existe maintenant de nombreux supports logiciels permettant de programmer indifféremment une machine parallèle ou un réseau de stations de travail, dont les plus courants actuellement sont PVM (Parallel Virtual Machine) [Geist93] et MPI (Message Passing Interface) [MPI93].

Il existe aussi des simulateurs de mémoire partagée pour les machines parallèles sans mémoire commune [Balaniuk94]. Avec une telle mémoire il est possible de reprendre toutes les études réalisées sur les machines à mémoire commune. Cependant, en raison du coût élevé d'une communication, le maintien de la cohérence de ce type de mémoire est considérablement plus pénalisant que des synchronisations et mises à jour au niveau d'un cache matériel ou d'un banc mémoire. Cette approche n'est pas possible pour un système Prolog parallèle du fait du taux important d'accès dispersés à la mémoire.

Le projet PLoSys a donc pour architecture cible les machines multiprocesseurs *sans* mémoire commune, aussi bien que les réseaux de stations, avec une seule restriction, les stations doivent être homogènes. Ce qui permet de limiter la complexité du noyau Prolog en adoptant un format de données unifiées. Ceci évite que les nombreuses références aux piles soient converties à chaque transfert hétérogène, ou soient remplacées par un système d'accès par base et déplacement.

4.1.3 Environnement de programmation

L'environnement de programmation de PLoSys (Figure 4-1) est constitué de trois parties distinctes :

- Un environnement de développement et d'exécution classique, articulé autour d'une implantation séquentielle, et qui peut être élaboré à partir d'un système Prolog déjà existant. Par la suite, l'utilisateur pourra choisir d'utiliser un Prolog séquentiel externe au lieu d'utiliser la version séquentielle du moteur Prolog de PLoSys. La seule contrainte à respecter concerne la sémantique du langage, qui doit être identique à celle choisie pour PLoSys, qui suit la norme préliminaire décrite dans [Covington93].
- Un environnement d'exécution parallèle, qui est alors utilisé pour augmenter les performances d'un programme Prolog. Il peut aussi permettre l'acquisition de mesures pour l'évaluation des performances. Cet environnement constitue l'essentiel de ce travail de thèse.
- Un environnement d'évaluation, qui permet le réglage de l'ordonnancement, de la régulation de charge, et leur adaptation à la machine parallèle. Il sert aussi à l'évaluation de l'efficacité du programme parallèle. Cette partie du projet est le sujet d'un autre travail de thèse [Kannat96].

La phase de mise au point d'un programme est traitée par l'environnement séquentiel, par l'utilisation des différents outils disponibles. Cette méthode est basée sur un des objectifs majeurs du projet : la parallélisation transparente, pour le programmeur, de

l'exécution du langage Prolog. Ceci permet alors de réutiliser les programmes Prolog développés avec la partie séquentielle de PLoSys. Le système d'exécution parallèle sert alors uniquement à l'accroissement de performances en terme de vitesse. Le modèle d'exécution parallèle de PLoSys ne permet pas d'augmenter la taille des problèmes traités, par l'utilisation de ressources matérielles supplémentaires.

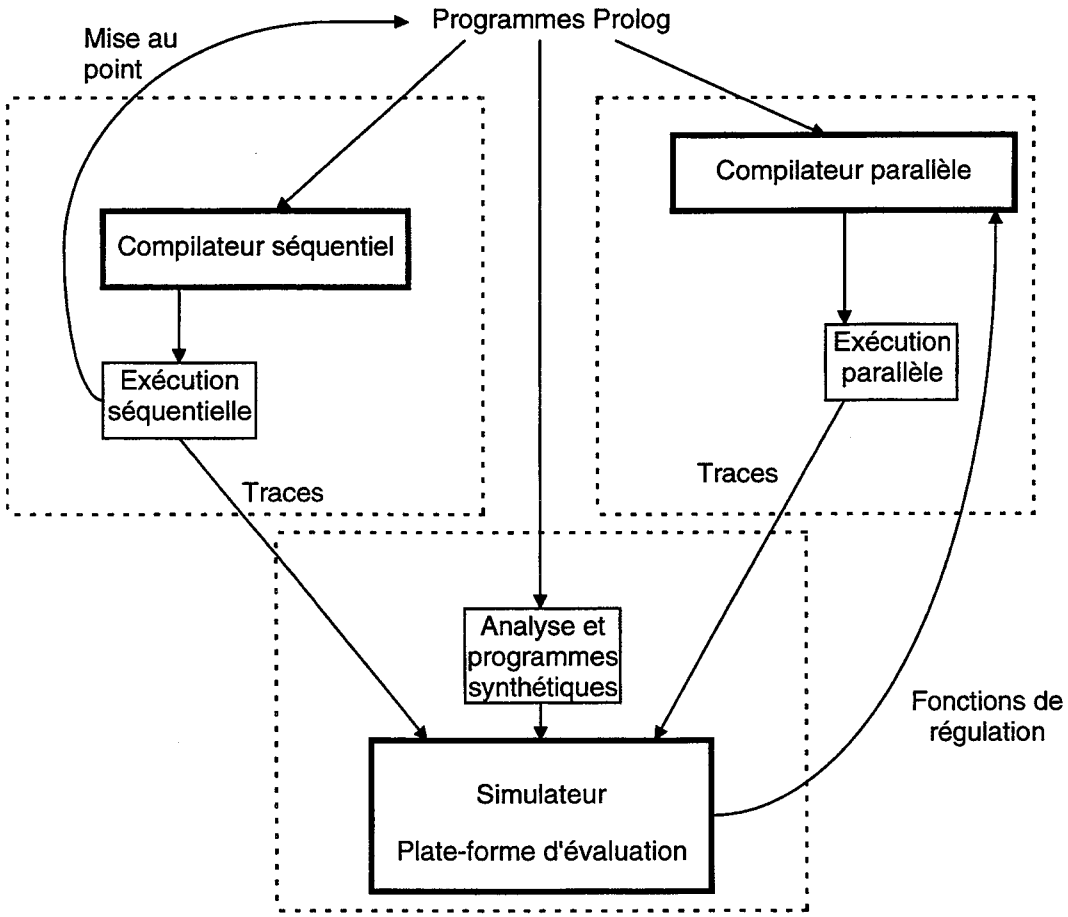


Figure 4-1 - Environnement logiciel de PLoSys

L'évaluation des performances d'un programme est obtenue par le traitement des informations issues d'une exécution parallèle d'un programme. A cette fin, le programme peut aussi être modélisé par un programme synthétique [Kannat94], et son exécution peut être simulée avec différents paramètres concernant l'architecture matérielle cible.

L'optimisation de l'exécution parallèle fait appel aux trois environnements du système PLoSys. L'environnement séquentiel permet une optimisation principalement au niveau de l'algorithme et du programme séquentiel correspondant. Il permet aussi de générer un ensemble de traces au niveau des instructions WAM utilisées. L'environnement

parallèle permet alors de mesurer les performances réelles d'une exécution. Quant à l'environnement d'évaluation, il permet, à l'aide des traces séquentielles, de déterminer la meilleure politique de régulation de charge associée à une architecture machine donnée. Il permet aussi de vérifier la stabilité de cette régulation par rapport aux variations de la taille du programme ou de des paramètres de la machine.
L'utilisateur peut alors modifier certains paramètres de la fonction de régulation, ou même créer une nouvelle fonction plus adaptée à ses besoins. A cette fin, l'utilisateur peut fixer les paramètres de la fonction de régulation lors du lancement de l'exécution. Pour des besoins plus spécifiques, il existe une interface de programmation simplifiée qui permet de créer de nouvelles fonctions de régulation dynamique de la charge.

4.2 Environnement d'exécution parallèle

Nous justifions dans cette partie le choix du modèle d'exécution parallèle retenu pour notre système. Puis, nous présentons les choix pris pour sa mise en œuvre, et sa structure logicielle.

4.2.1 Choix du type de parallélisme

Le choix du type de parallélisme est guidé en partie par l'architecture ciblée. En effet, l'utilisation de communication limite le parallélisme à un gros grain, ce qui correspond plutôt à du parallélisme OU, car le parallélisme ET ainsi que le parallélisme de données, apparaissent à un niveau généralement plus fin de calcul [Chassin94].

Un autre point en faveur du parallélisme OU est sa facilité de mise en oeuvre à partir d'un système Prolog séquentiel [Geyer91], car les modifications apportées au système séquentiel sont très faibles et le surcoût à l'exécution est très bas [Karlsson92].

Les modèles à « tableau noir » nécessitent une zone d'accès partagée qui peut être assez facilement simulée par un service réparti. Mais s'ils peuvent être exploités avec un grain de taille variable, et donc être relativement performants au niveau d'une machine sans mémoire commune, leur utilisation n'est pas transparente au niveau de la programmation, et ne permet pas l'utilisation de programmes séquentiels classiques.

Le parallélisme combiné ET/OU pourrait être envisagé dans le cas où les nœuds de la machine sont des multiprocesseurs à mémoire commune, tels des serveurs multiprocesseurs symétriques (*SMP*) reliés par un réseau. Le parallélisme ET devient alors raisonnablement exploitable au sein d'un nœud du réseau, à l'aide d'un modèle ET bien adapté sur l'architecture à mémoire commune, tandis que le parallélisme OU serait utilisé pour diffuser le travail à tous les nœuds le réseau.

Enfin nous avons présenté dans le chapitre précédent une méthode permettant la réalisation d'un Prolog complet, avec ses effets de bord conformes à la sémantique séquentielle. Cette méthode est liée à l'utilisation du parallélisme OU-multiséquentiel.

Aussi, seul le parallélisme OU multiséquentiel nous semble bien adapté aux architectures sans mémoire commune (même dans le cas des réseaux de stations), par

son grain assez gros, sa mise en oeuvre simple, et la compatibilité naturelle avec le langage séquentiel. Cela nous permet aussi de profiter de l'expérience acquise par le projet OPERA, et en réduisant le temps de conception du modèle, de concentrer nos efforts sur les objectifs de portabilité et de gestion implicite du parallélisme.

4.2.2 Principes de mise en œuvre

4.2.2.1 La gestion de la mémoire

Etant donné que nous ne disposons pas de zone de mémoire commune, les processeurs coopérant à la résolution du programme vont donc parcourir l'arbre d'exécution en partageant les parties communes de leur chemin à l'aide de copies de leur état. La gestion de la cohérence de ces copies, appelée aussi gestion de contextes multiples, peut être résolue par deux méthodes générales:

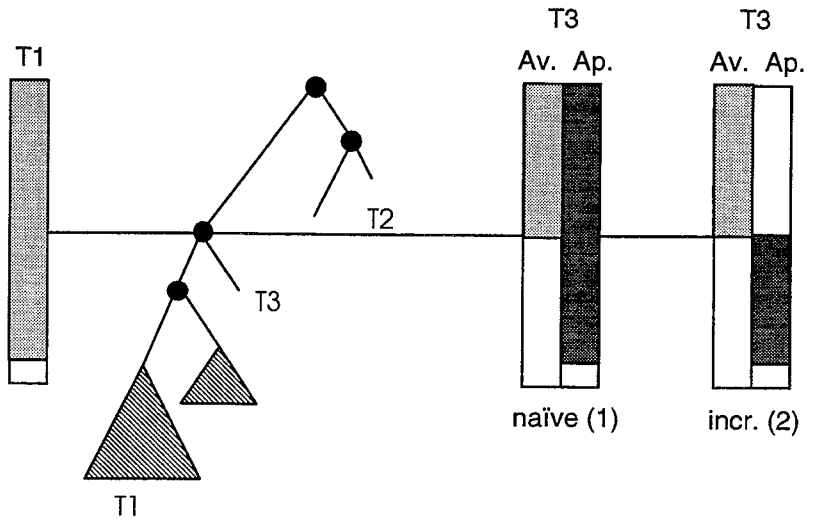
- ***La copie de piles***
  Dans ce cas, lorsqu'un processeur acquiert du travail auprès d'un autre, il recopie l'état des ses piles, et simule ensuite le retour arrière sur le point de choix qu'il a reçu, ou du plus récent point de choix s'il en a reçu plusieurs.
  Cette méthode est très courante dans les systèmes sans mémoire commune, et est même utilisé par un système à mémoire commune (Muse).

- ***Le recalcul***
  Le principe est apparu avec l'augmentation des performances des processeurs. Il fait le pari que le recalcul d'un état de l'exécution Prolog séquentiel est plus rapide que le transfert des piles. Ce recalcul est établi à partir du chemin dans l'arbre correspondant à la position devant être retrouvée [Clocksin88]. La communication ne sert plus alors qu'à transférer un chemin dans l'arbre, au lieu de l'ensemble des portions de piles, ce qui est censé diminuer fortement la taille du transfert.

Nous avons choisi d'utiliser le transfert de pile car sa gestion est maintenant bien connue, et son implantation ne pose pas de problème particulier. Les modifications apportées à un noyau séquentiel demeurent simples et très limitées. Le recalcul, outre sa gestion plus complexe, à cause du calcul des oracles, ne nous semble pas aussi bénéfique qu'au premier abord [Araujo94]. En effet, le recalcul n'est valable que si les piles sont de grande taille par rapport aux oracles, et que le taux de calcul est assez faible pour les reconstruire. Ceci nécessiterait l'analyse de nombreux programmes pour s'en assurer, et dépend fortement de la machine cible. Enfin, la taille d'une communication n'est pas le principal frein dans la course aux performances. La latence, temps d'établissement de la communication, n'est pas négligeable par rapport au temps de cycle d'un processeur, et peut représenter quelques milliers à millions d'instructions. Or celle-ci est présente dans les deux cas, recalcul ou copie des piles, Comme les débits des liens de communications augmentent sans cesse, et que la latence diminue peu, les petits messages offrent un débit relatif très faible; l'emploi de messages de grande taille est donc de moins en moins pénalisant.
Pour la gestion de contextes multiple par transfert de pile, il existe encore deux possibilités d'efficacité et de complexité croissante [Karlsson92] :

- la copie complète (naïve) est la plus simple. Toutes les piles sont recopiées sans tenir compte de l'état précédant du travailleur qui importe du travail.

- la copie incrémentale, qui tente de minimiser les transferts de données inutiles, en ne copiant pas les parties communes des piles entre l'état de l'importateur avant importation et après exportation.
  Cette forme de copie, pour être véritablement rentable doit être employée avec une politique favorisant l'échange de piles entre travailleurs ayant de grandes sections communes.
  Il existe enfin une version optimisée de la copie incrémentale dont l'intérêt n'apparaît vraiment qu'avec un modèle ET/OU [Costa94].



Le travailleur T3 prend du travail à T1. Dans la copie naïve, il doit recopier les portions de pile correspondant aux piles de T1 (1). Avec la copie incrémentale (2), seules les parties divergentes sont copiées.

Figure 4-2 - Les différentes méthodes de copie de piles

Lors du transfert des piles, il est possible soit d'interrompre l'exécution en cours, soit de la maintenir pour tenter de masquer les coûts des communications.
Dans le premier cas il n'apparaît pas de problème autre que la gestion de l'exclusion mutuelle entre le transfert et l 'exécution, ce qui est trivial.
Dans le second, il faut pouvoir assurer que l'exécution ne modifiera pas des données non encore transférées. Il faut alors prévoir un mécanisme de synchronisation sur l 'accès de ces données, ou de maintien de cohérence de l'état des données lors de leur modification. Le principe est de ne n'autoriser la modification par l'exportateur que d'une partie de pile déjà exportée, évitant ainsi une situation d'incohérence pour l'importateur.

La pile de traînée est aussi très sensible de ce point de vue car elle contient les références aux liaisons à défaire lors du retour arrière. Il existe là aussi deux approches pour gérer la cohérence des liaisons des variables.

La plus simple est l'utilisation traditionnelle de la pile de traînée, qui doit alors être transférée entière sur le nouveau travailleur. Puis lorsqu'on simule le retour-arrière sur le point de choix importé, ou le plus récent point de choix importé, la machine Prolog effectue une déliaison classique. Ce principe impose que la pile ne soit pas modifiée avant la fin du transfert par l'exportateur, et donc globalement un arrêt de la machine Prolog exportatrice; ou la pile peut être modifiée de manière synchrone au transfert, les modifications pouvant avoir lieu dans une partie déjà transférée.

La seconde approche, appelée Kabu-Wake [Masuzawa86], associe à chaque liaison une date logique, gérée par chaque travailleur. Ainsi lors du transfert des piles, la date logique correspondant au plus récent point de choix importé sert à déterminer la nouvelle validité des liaisons rencontrées. Si la date de liaison est postérieure à la date du point de choix, celle-ci doit être défaite. Cette méthode augmente la consommation mémoire à cause de la date associée aux liaisons, et ajoute un peu de traitement supplémentaire aux opérations de gestion des liaisons, ainsi qu'au niveau des points de choix (mémorisation de la date). Elle permet par contre de ne pas synchroniser l'exportateur et l'importateur lors du transfert de la pile de traînée.

Etant donnée que nous voulions apporter le moins de modifications possible au système séquentielle, nous avons choisi d'interrompre le travailleur exportateur pendant la phase de transfert des piles. Ce choix n'est pénalisant que si l'on peut effectuer les transferts en parallèle du calcul, afin de recouvrir le temps de communication; et à condition que ce recouvrement compense le surcoût du a la gestion de la datation.

4.2.2.2 Politique de transfert de la charge

Comme le montre la Figure 4-3, il existe deux politiques distinctes pour partager du travail dans le modèle multiséquentiel, soit un travailleur partage ses plus vieux points de choix, soit il partage ses plus récents. A priori cela ne semble pas très différent, mais nous allons voir en fait que le choix de l'une ou l'autre des politiques est important et pas du tout anodin.

Plus récents points de choix

Ces points de choix correspondent à la partie non-encore explorée la plus à gauche de l'arbre d'exécution, et aussi proche des feuilles de l'arbre, c'est à dire proche de la fin d'une résolvante. Ils conduisent à une quantité de travail supposée faible avant un retour- arrière antérieur à eux-même.

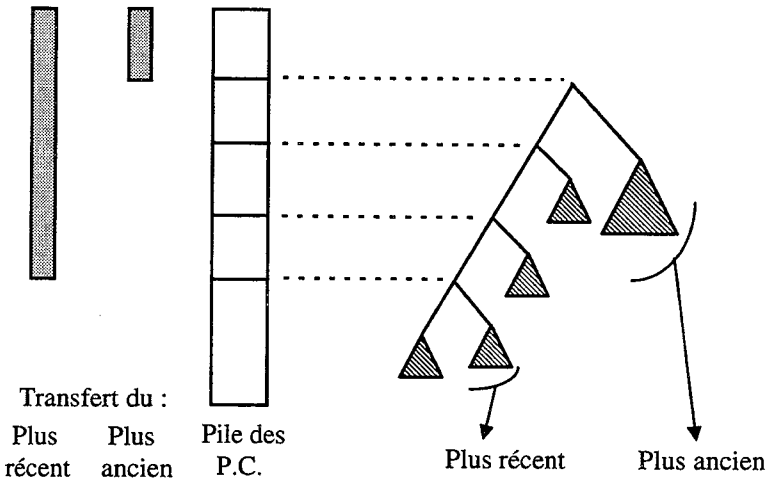
Plus anciens points de choix

Les points de choix anciens, placés au bas de la pile locale, représentent la partie non-explorée la plus à droite de l'arbre du programme, la plus proche de la racine. Le potentiel de travail peu être grand car il peut contenir des résolvantes complètes.

Le choix de l'une ou l'autre des politiques peut être établi par deux paramètres, la quantité de travail que l'on peut partager et le coût associé. Dans les deux cas, le coût

de partage, est lié à la taille des données échangées nécessaires à la reprise du travail sur le nouveau site.
On voit donc immédiatement que les deux politiques divergent alors complètement pour une utilisation sans mémoire commune, où un partage de travail d'un processeur à l'autre passe obligatoirement par le transfert de l'état de reprise. Ceci représente globalement le transfert des piles du travailleur exportant une partie de son travail. Le choix des plus anciens point de choix, réduit alors considérablement la taille de la communication. Pour l'évaluation de la quantité de travail échangée, il n'existe pas de moyen simple et efficace de la prédire. On peut supposer tout de même qu'en moyenne, un point de choix ancien a plus de chance de générer une quantité importante de calcul, qu'un point de choix récent.

La politique adoptée pour PLoSys est le partage des plus anciens points de choix, car comme nous l'avons vu dans le chapitre précédent, c'est la seule solution pour gérer simplement les effets de bord.



Le transfert du plus ancien point de choix conduit à copier une portion plus petite des piles (grisé), et le travail potentiel peut être important par rapport au point de choix le plus récent.

Figure 4-3 - Politiques de transfert des points de choix

4.2.3 Architecture logicielle

Le système PLoSys devant évoluer, sa conception repose sur des modules indépendants, communicants via une interface précise. Ainsi, chacun des modules peut être modifié facilement, et en particulier le module de régulation de charge, dont l'étude fait l'objet d'un travail de thèse spécifique [Kannat96]. Ce module doit être adapté facilement aux contraintes du système utilisé, machine spécialisée ou réseau de stations. Par exemple, le coût de communication variant fortement entre un réseau de communication dédié à

une machine et un réseau de station de travail; exige des fonctions de régulation de charge adaptées.

4.2.3.1 Organisation de l'exploitation du parallélisme

L'exploitation du travail parallèle est réalisée par un ensemble de travailleurs coopérant à la résolution d'un programme. Ceux-ci sont contrôlés par un ordonnanceur centralisé. L'architecture de ces tâches est donc du type maître-esclaves (Figure 4-4). Une tâche console permet de gérer l'interaction avec l'extérieur, comme l'affichage et les accès fichiers. Dans ce modèle, il n'est donné aucune hypothèse quant au placement de ces différentes tâches, la seule restriction est que chaque fonction soit disposée sur un processeur virtuel différent. Mais il est possible, par exemple, de placer plusieurs travailleurs sur un même processeur physique. Il est donc tout à fait envisageable d'utiliser PLoSys sur un monoprocesseur. Bien évidemment les performances seront moindre qu'avec une exécution répartie, sauf pour une exécution séquentielle.

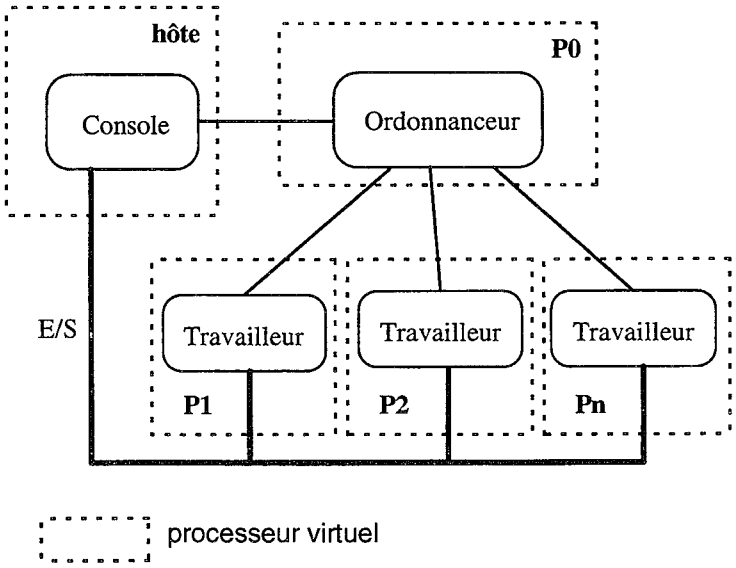


Figure 4-4 - Architecture de PLoSys

La résolution d'un programme débute sur un travailleur, puis est diffusée sur les autres au fur et à mesure de la disponibilité de travail et de processeurs libres. Le transfert du travail est à l'initiative des processeurs libres du système, par échange de messages (voir
Figure 4-5).
En raison du coût élevé des communications, le nombre de messages échangés doit être le plus bas possible afin de ne pas trop pénaliser les performances de PLoSys. Le protocole présenté (
Figure 4-5) est celui que nous avons adopté à cause de son très faible nombre de messages.

Le système parallèle sur lequel PLoSys exécute un programme, ne peut être modifié dynamiquement par ajout ou suppression de processeurs. Ceci peut être envisagé assez facilement pour l'ajout de processeurs en cours d 'exécution, où il suffirait d'introduire les nouveaux éléments comme inactifs. Pour le retrait, il faudrait introduire quelques restrictions, comme l'attente d'un état inactif de la part d'un des processeurs, afin de pouvoir y transférer le travail de celui que l'on doit retirer de l'exécution.

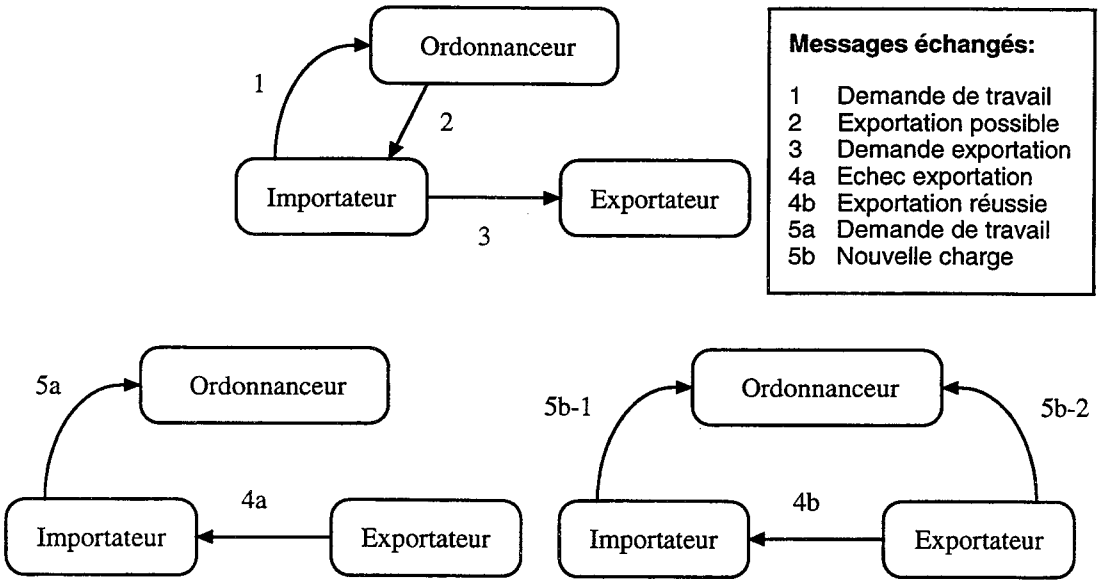


Figure 4-5 - Principe d'échange de travail

Enfin, le système PLoSys suppose un environnement fiable et par conséquent n'est pas conçu pour être tolérant aux pannes. Cela pourra faire l'objet d'un travail ultérieur. Il existe par ailleurs des environnements de programmation permettant la tolérance aux pannes, par extension des langages courants tels que C ou Prolog [Forst94].

4.2.3.2 Régulation de charge

La régulation dynamique de la charge passe par la connaissance de l'état du système, afin de pouvoir prendre les bonnes décisions, pour répartir convenablement le travail à accomplir. Cela nous conduit à la définition de deux besoins, le premier est de déterminer le type de l'information que l'on souhaite utiliser pour représenter l'état du système. Le second est la méthode avec laquelle l'information est collectée dans le système.

L'état du système

Dans notre modèle, la répartition de la charge de travail s'effectue à partir du transfert des plus anciens points de choix d'un travailleur. Ainsi chaque travailleur agit en producteur ou consommateur de points de choix.

La mesure de la charge d'un travailleur que nous avons adopté est le nombre de points de choix en attente d'être traités. Ce choix permet une évaluation très simple et rapide. Il correspond assez bien avec le taux réel d'utilisation du processeur par le travailleur [Tick91], même si les points de choix peuvent conduire à une quantité variable de travail, suivant la résolvante associée.

Mesure de l'état du système

L'estimation de la charge du système ne peut être effectuée en temps réel. En effet, l'architecture de la machine étant distribuée, il n'est pas possible de connaître l'état des différents nœuds simultanément. Par conséquent l'état du système vu par l'ordonnanceur devient une fonction discrète de l'état réel.

Un moyen simple de contrôler l'état du système, est un échantillonnage périodique. Le choix de l'unité de la période peut être le temps physique, ou le nombre d'instructions exécutées par le travailleur, ou tout autre événement régulièrement rencontré lors d'une exécution.

Dans notre système, les points les plus significatifs sont le temps et la gestion des points de choix (unité d'échange). Le comptage des instructions revenant à étudier la vitesse d'exécution, nous ramène en quelque sorte à la gestion du temps.

L'utilisation de la variation du nombre de points de choix sur un travailleur est proche de ce que l'on doit mesurer, la quantité de ces points de choix. Cependant, cette variation peut être très lente, voire nulle, et la période de l'échantillonnage peut alors ne plus être bornée.

La gestion de l'échantillonnage avec une base de temps physique, permet de garantir que la période est bornée. Par contre, elle n'est pas directement en accord avec la mesure à prendre, la quantité de point de choix en attente d'être traités, En effet, si la période est très grande, elle ne transmettra pas assez rapidement les variations du nombre de point de choix, si elle est trop petite, elle transmettra souvent la même information.

L'idéal semble donc d'utiliser une combinaison de ces deux méthodes, comme le démontre le projet OPERA. C'est donc la méthode que nous avons retenue pour notre système.

L'échantillonnage est basé sur le temps, avec une période paramétrable. Cependant, pour éviter de transmettre des informations redondantes, la variation de la quantité de points de choix en attente devra dépasser un seuil paramétrable.

Prise de décision

Ayant des informations sur l'état global du système, il faut maintenant être capable de prendre les bonnes décisions quant à l'appariement de deux travailleurs pour un échange de travail. C'est le rôle de la fonction de régulation dynamique de la charge.
D'un côté, nous savons grâce à l'expérience apportée par les différents projets, qu'une fonction universelle n'existe pas. D'un autre côté, une partie du projet PLoSys est l'étude générale de ces fonctions, mais n'entre pas directement dans le cadre de l'implantation du système. Nous avons donc décidé d'utiliser une interface simple pour pouvoir écrire des fonctions de régulation indépendamment de la réalisation du prototype. Enfin, pour pouvoir tester les premières versions du système, celui-ci intègre une fonction de régulation naïve, dérivée de celle employée dans OPERA .

4.2.3.3 L'ordonnanceur

La structure type de l'ordonnanceur de PLoSys est définie à partir des différentes fonctions que doit gérer celui-ci, et qui sont organisées par unités indépendantes. Chaque unité de l'ordonnanceur comprend les fonctions nécessaires au traitement qu'elle doit accomplir, ainsi que les fonctions interfaces permettant l'accès aux informations propre à l'unité, ce qui permet de gérer les interactions entre les différentes unités.

En premier lieu, l'ordonnanceur collecte des informations sur l'état global du système, en regroupant celles obtenues depuis les travailleurs. Ces informations seront alors utilisées par la première unité de l'ordonnanceur, la fonction de régulation de charge, qui permet de définir quel travailleur actif partagera son travail avec un travailleur inactif.

Il faut lui ajouter les unités de gestion des effets de bord. Pour pouvoir gérer chacun des effets de bord de la manière appropriée, et faciliter les éventuelles modifications, chaque type d'effet de bord disposera d'une unité spécialisée.

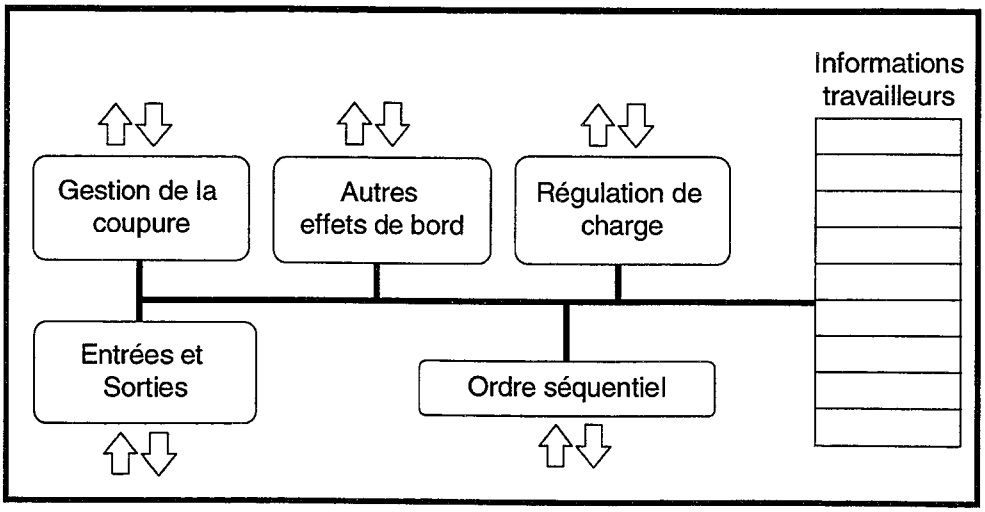


Figure 4-6 - Ordonnanceur

Les différentes unités sont regroupées au sein de la tâche ordonnanceur, par l'utilisation d'un schéma à base de processus et de modules de programme. Les processus sont employés quand l'unité à une fonction communiquante, sinon l'unité est uniquement associée à un module du programme.

A l'intérieur de la tâche, les processus communiquent par partage de mémoire. A l'extérieur, ils utilisent le passage de messages.

4.2.3.4 Les travailleurs

La structure des travailleurs est aussi organisée en unités indépendantes partageant des informations. Elles sont de quatre types :

- Le noyau Prolog, dont les piles sont rendues accessibles aux autres unités.
- La gestion de charge locale, qui assure la mise à jour des informations sur la charge locale du travailleur, ainsi que le contrôle de la validité d'un échange de travail. En effet, entre le moment où l'ordonnanceur prend une décision d'échange et le moment où cet échange est réalisé, l'état du travailleur peut avoir complètement changé, rendant caduque l'échange.
- La gestion des échanges, qui est confiée à deux unités, l'une pour l'exportation de travail et l'autre pour l'importation.
- Enfin, chaque effet de bord est géré par une unité spécialisée, en relation avec celle définie dans l'ordonnanceur.

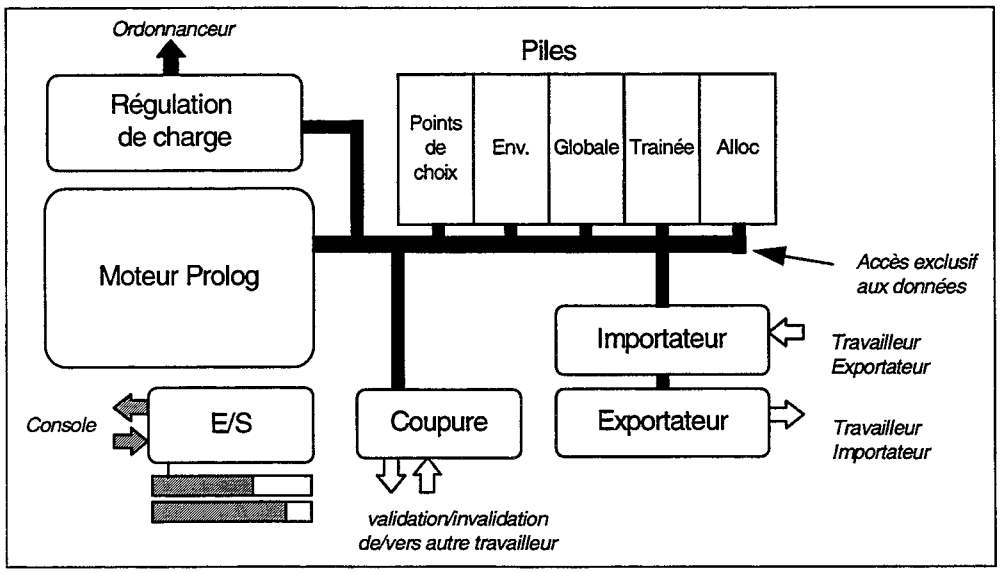


Figure 4-7 - Le travailleur

Toutes les unités du travailleur disposent encore d'une interface pour gérer leurs multiples interactions. Comme pour l'ordonnanceur, le travailleur est considéré comme une seule tâche, et les unités donnent lieu à l'utilisation de modules et de processus.

Le moteur d'exécution séquentielle

Afin d'obtenir un système Prolog réaliste et utilisable, nous avons choisi la voie de la compilation, bien que cela nous impose l'emploi d'un ensemble homogène de processeur pour la cohérence du format des données et la simplification des transferts (4.2.2.1). C'est le cas pour des machines parallèles; mais pour un réseau de stations ce n'est plus obligatoire. PLoSys ne pourra donc exploiter qu'un sous ensemble homogène d'un réseau de stations.
Le système Prolog utilisé par PLoSys est typiquement basé sur une machine abstraite séquentielle classique, dont l'implantation est modifiée pour supporter une exécution parallèle.

5. Implantation du système

L'implantation de PLoSys, outre la simplicité et l'efficacité, a été guidée par plusieurs critères, comme la configuration matérielle, la gestion des communications, la gestion des différentes unités au sein d'un travailleur ou de l'ordonnanceur, et enfin le moteur Prolog séquentiel. Nous présentons successivement les possibilités offertes et les choix que nous avons pris pour l'organisation et la réalisation d'un prototype.

5.1 Architectures matérielles

Pour réaliser notre prototype nous disposions de trois types de machines et de plusieurs systèmes d'exploitation différents.

- La première architecture cible est la machine parallèle sans mémoire commune de l'IMAG, un IBM SP/1, qui comprend 32 processeurs (ou noeuds), avec leur mémoire propre, un espace disque local, et chacun supporte le système d'exploitation AIX complet. Chaque processeur est relié aux autres par un réseau spécialisé à haute vitesse et par un réseau Ethernet classique, qui en permet aussi l'accès depuis l'extérieur de la machine. Enfin deux stations RS/6000 servent de système hôte, pour l'accès et la gestion des nœuds, ainsi que de serveur de fichiers.
  Les communications peuvent être établies entre n'importe quels noeuds de la machine, et ce quel que soit le réseau utilisé, et les noeuds sont identiques à des stations autonomes vis à vis du système. L'utilisation du réseau spécialisé est exclusive à une application pour un sous-ensemble de nœuds, mais deux applications peuvent se partager la machine en utilisant des sous-ensembles disjoints de nœuds.
  Cette machine peut donc être considérée comme une machine parallèle spécifique, si l'on utilise le système de communications spécialisé, ou comme un ensemble de stations de travail homogènes, si l'on utilise le réseau Ethernet.

- La deuxième architecture est représentée par les stations et serveurs Sun, exploitant le système Solaris. Ces systèmes ont une puissance de calcul assez importante, avec en particulier pour les serveurs, la possibilité supplémentaire d'avoir une architecture multiprocesseurs symétriques. De tels réseaux comportent généralement des dizaines de stations, voire quelques centaines de stations. Les communications sont habituellement mises en œuvre par un réseau Ethernet classique.

- La troisième architecture, très proche de la précédente, est représentée par des ordinateurs individuels du type compatible PC, exploités par le système Linux ou Solaris. Leur utilisation en réseau représente actuellement une alternative très économique par rapport à une architecture parallèle dédiée. Leur puissance de calcul a pratiquement atteint celle des serveurs classiques, et les mêmes microprocesseurs sont par ailleurs utilisés pour les nouvelles générations de machines parallèles. Les communications sont en général supportées par un réseau Ethernet classique, ou un réseau rapide. Cette dernière solution devient de plus en plus courante, car elle est financièrement abordable, et présente de très bonnes performances.

Dans les deux derniers types d'architectures, les nœuds du réseau peuvent être hétérogènes au sens du processeur ou du système d'exploitation. Pour des raisons de simplicité de programmation, ainsi que pour préserver l'efficacité du moteur séquentiel, nous ne considérerons que des réseaux homogènes, ce qui permet l'utilisation du même code exécutable pour tous les nœuds, ainsi que le même format de données.
Pour vérifier la portabilité de notre prototype, nous utilisons le même jeu de fichiers sources sur ces trois types de plates-formes.

5.2 Le support d'exécution parallèle

Il existe plusieurs solutions pour communiquer entre les différentes tâches du système, ainsi que plusieurs manières d'organiser les différentes unités du système, en tâches indépendantes. Nous présentons ici notre choix et les contraintes qu'il introduit.

5.2.1 La gestion des unités

Pour implanter notre modèle de gestion de l'exécution parallèle de Prolog, nous avons recours à des unités fonctionnelles indépendantes qui échangent des informations soit par des communications soit par partage de zones mémoire. Ces unités sont regroupées au sein d'un processeur virtuel pour chaque travailleur et pour l'ordonnanceur.

Il existe deux possibilités d'implantation, soit l'utilisation de différentes tâches (processus Unix), soit la programmation à l'aide de processus légers, au sein d'une même tâche. Dans la première solution, il est difficile de partager des zones de mémoire, et on ne peut définir le processeur virtuel comme une seule entité au sein du système d'exploitation. Par contre, l'utilisation d'une seule tâche (processus Unix) au sens du système permet de définir clairement un processeur virtuel. Il faut cependant pouvoir, dans cette tâche, gérer les différentes unités. Ceci est très simplement réalisable à l'aide de processus légers, qui partagent alors la zone d'adressage et le temps d'exécution de la tâche les contenant.

5.2.2 Les communications

Afin de pouvoir utiliser le parallélisme, une application doit pouvoir échanger des données entre les différents processeurs d'une machine, à l'aide de primitives de communication. Dans le cas général il existe deux sources pour ces primitives:

- Les services du système d'exploitation des machines offrent des primitives de base pour communiquer (TCP/IP, *sockets*). Cependant la mise en place des communications et le traitement des messages est assez fastidieux, car leur interface avec les langages de programmation classique est minimale.

- Les bibliothèques de communications, comme PVM [Geist93] ou MPI [MPI93]. Ces bibliothèques offrent généralement des primitives de communications de plus haut niveau. Le développement d'application est donc plus simple, et leur standardisation permet le portage d'application au même titre que les bibliothèques standards des systèmes.

Pour des raisons évidentes de simplicité d'implantation et de facilité d'adaptation, nous avons préféré utiliser une librairie de communications existante, plutôt que d'intégrer une interface de communication propre à notre système.

5.2.3 La bibliothèque de programmation parallèle

L'utilisation conjointe d'une librairie de communication et d'un noyau de processus léger n'est pas triviale. En effet, le mariage des deux implique une modification de la librairie de communication pour qu'elle puisse gérer l'interaction avec les processus légers, ainsi qu'une adaptation à chaque portage, aux petites différences entre les noyaux de processus disponibles.
Pour éviter de tels problèmes, il est souhaitable de choisir une bibliothèque appropriée pour la programmation d'applications parallèles à l'aide de processus légers. L'une de ces bibliothèques est développée au sein de notre laboratoire, et a donc été retenue de fait. C'est la bibliothèque Athapascan-0a [Christaller96].

Athapascan-0a

Cette bibliothèque de programmation parallèle propose une interface indépendante du noyau de communication et du noyau de processus utilisés, et offre une portabilité transparente à l'utilisateur. La gestion des communications est fournie par la librairie PVM, tandis que le noyau de processus provient de différentes sources selon l'architecture matérielle employée.

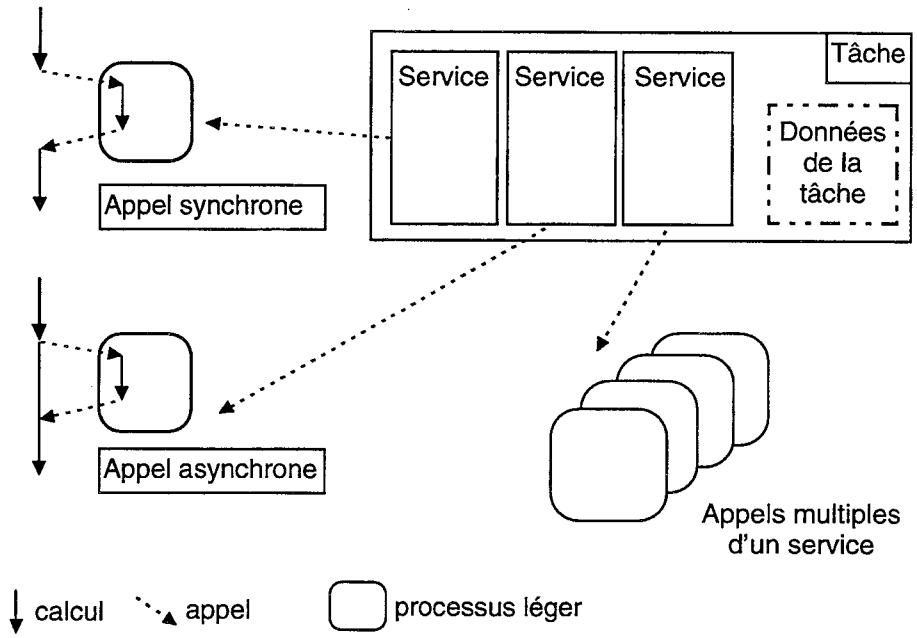


Figure 5-1 - Athapascan-0a

L'application est découpée en tâches communicantes. Chaque tâche contient des points d'entrée ou services, qui peuvent être accédés par le mécanisme d'appel à distance ou RPC [Stevens90] (Figure 5-1). Ces services, implantés comme des processus légers,

partagent les informations de la tâche les contenant, et notamment les zones mémoire. L'appel d'un service peut être synchrone ou asynchrone, auquel cas l'appelant poursuit son exécution parallèlement à l'exécution du service appelé. Chaque appel d'un service provoque la création d'un processus léger qui effectuera le traitement de la requête associée.
La transmission de données est réalisée par le passage de paramètres au niveau de l'appel à distance, l'envoi des données correspondant à l'appel, et la réception, au retour de fonction.

La librairie Athapascan-0a ne gère pas les interruptions, ni la préemption des processus légers.

5.3 Le noyau Prolog

Le choix du noyau d'exécution Prolog est primordial car il conditionne non seulement les performances du système, mais aussi sa mise en œuvre. La première qualité d'un moteur Prolog est son efficacité pour exécuter les programmes Prolog. Mais sa portabilité et sa modularité sont essentielles pour construire un système parallèle portable et extensible. Enfin, les outils développés autour du noyau d'exécution, tels les compilateurs ou outils de mise au point, de traces, apportent le confort de développement pour le programmeur et utilisateur final.

5.3.1 Choix d'un noyau Prolog

Le premier choix lorsque l'on veut implanter un système Prolog parallèle, est le type d'exécution des programmes Prolog, soit l'interprétation, l'émulation de la WAM ou la compilation pour la machine cible.

- **L'interprétation**
  C'est la solution la plus simple à réaliser, mais aussi la moins performante. Elle permet la gestion de code Prolog dynamique sans travail supplémentaire. Les modifications de l'interpréteur sont triviales, mais sa lenteur masque la plupart des problèmes liés à l'exécution parallèle de Prolog, en particulier le coût des communications.

- **L'émulation**
  Cette solution intermédiaire est utilisée par de nombreux systèmes commerciaux et universitaires, dont Sicstus Prolog par exemple. Elle offre un bon compromis entre efficacité et portabilité. En effet, le programme Prolog est traduit en pseudo-code indépendant de la machine, qui est ensuite exécuté par un émulateur. La réalisation d'un émulateur portable est très simple, et la gestion dynamique du code Prolog est plus complexe que pour l'interprétation.
  Cette approche n'est cependant pas optimale au point de vue de la vitesse d'exécution séquentielle, en raison de la phase de décodage.

- **La compilation**
  C'est la voie la plus efficace pour l'exécution, car le programme Prolog est traduit directement en code natif de la machine cible. Mais, elle est plus délicate à gérer au niveau de la portabilité, et de la souplesse de modification. La gestion des prédicats dynamiques est délicate et très coûteuse.
  Cependant ces inconvénients restent raisonnables au regard des performances obtenues lors de l'exécution.
  Le système Sicstus peut générer pour certaines machines du code natif, qui est bien plus efficace que le code émulé.

Le choix du type d'exécution pour PLoSys dépend aussi de la façon d'obtenir le système :

- créer entièrement le noyau Prolog et les outils associés,

- utiliser le noyau d'un système existant.

La première option a l'avantage de permettre de définir précisément la machine abstraite utilisée pour l'exécution de Prolog, afin de l'optimiser pour la gestion de l'évaluation parallèle. Cela autorise aussi le choix d'implantation le plus adapté pour la portabilité et les modifications futures. Par contre, pour construire le noyau Prolog et ses outils associés, l'effort de programmation est très important, et vient en plus de celui nécessaire à la gestion parallèle du système.

La seconde approche économise toute la phase de développement et mise au point du noyau Prolog et de ses outils associés. A la place de cette phase, se succèdent une étape d'analyse du système, puis de modification pour l'exécution parallèle. Les principaux avantages sont le temps réduit pour obtenir un système parallèle et l'ensemble des outils déjà présents (même s'ils sont séquentiels) peut aider à sa mise au point. Enfin, les performances du système séquentiel sont connues dès le départ. Les inconvénients liés à cette approche sont la difficulté éventuelle de modification du noyau Prolog et de son compilateur, et les possibilités d'évolution de l'ensemble que nous ne contrôlerons pas.

Comme notre effort est porté sur le parallélisme et les effets de bord de Prolog, nous avons choisi d'utiliser un noyau Prolog déjà existant, pour concentrer notre travail sur les points essentiels. Pour parer aux problèmes d'évolution et de possibilité de modifications, tout en gardant un maximum de performances, le choix du système séquentiel devient très important.

Nous avions le choix entre de nombreux systèmes, en éliminant tout de même les systèmes commerciaux qui posent des problèmes évident de diffusion des programmes sources. Sont aussi éliminés les systèmes parallèles basés sur des architectures avec mémoire commune, car ceux-ci sont trop intimement liés à leur architecture, ou utilisent aussi une version séquentielle modifiée. Il reste donc les systèmes séquentiels classiques, qui représentent déjà un vaste choix de possibilités. Parmi ceux-ci certains ont retenu notre attention :

- Sicstus Prolog : ce système qui fait souvent office de référence est donc un bon candidat. Mais, si ce système est performant et complet, avec de nombreuses extensions, il est aussi imposant et difficile à modifier. La génération de code compilé (mode natif) n'est disponible que pour certaines architectures seulement.

- BinProlog : il propose une approche différente de la compilation de Prolog en transformant toutes les clauses d'un programme en clauses binaires [Tarau91]. Cette transformation permet de simplifier la machine abstraite car elle supprime les environnements.
  Ce système procure de bonnes performances, et sa conception est assez simple. Il génère aussi du code compilé et dispose de nombreuses extensions.

- WAMCC : c'est un des systèmes les plus simples, il ne comporte qu'un compilateur et un outil de mise au point. Cependant sa particularité est de générer à partir d'un programme Prolog, un programme C. Il est très efficace, facilement portable, modulaire et déjà destiné à être modifié [Diaz95].

Nous avons choisi d'utiliser le compilateur Prolog WAMCC pour deux raisons essentielles, sa petite taille qui permet des modifications aisées, ainsi que sa portabilité qui ne dépend que de la présence du compilateur C de GNU ou d'un compilateur permettant l'inclusion en ligne d'instructions assembleur. De plus ses performances séquentielles sont très bonnes en comparaison aux systèmes commerciaux disponibles, et souvent très supérieures aux systèmes universitaires contemporains [Diaz95].

5.3.2 Description de WAMCC

Ce compilateur transforme un programme Prolog en un programme C. Il travaille en deux étapes :

- La première est la classique compilation de Prolog vers une machine abstraite issue de la WAM.

- La seconde étape est la transformation de ce code abstrait en langage C. Pour simplifier, nous pouvons considérer que chaque instruction de la WAM est traduite par une fonction C, à ceci près, que pour augmenter les performances, WAMCC utilise des sauts extra fonctions pour éviter la gestion du code d'entrée et de sortie des fonctions C. C'est ici que les spécificités du compilateur GCC (comme l'inclusion en ligne d'instructions assembleur) sont utilisées, afin de pouvoir générer ces sauts.

A partir d'un programme source en Prolog, l'utilisateur obtient un ensemble de programmes en langage C. Ces programmes doivent ensuite être liés avec la bibliothèque de WAMCC qui représente la machine abstraite de Prolog, pour obtenir finalement un programme exécutable indépendant.

5.3.3 Modifications générales du noyau d'exécution WAMCC

Pour permettre une exécution parallèle de Prolog pur avec notre système, il est nécessaire d'apporter quelques modifications au noyau WAMCC.

Gestion des piles

Dans la WAM classique, les opérations sur les points de choix sont indépendantes de celles effectuées sur les environnements. Mais pour simplifier la gestion des environnements, la WAM utilise une seule pile pour les deux types d'objets gérés [AitKaci90]. Cela pose un problème quand on veut transférer les piles d'un travailleur à un autre. En effet, des points de choix non exportés vont être intercalés avec des environnements à exporter. Il faudra donc envoyer ces environnements par blocs discontinus, augmentant ainsi le nombre de messages, ou créer par recopie un seul bloc. Ces solutions ne sont pas acceptables pour un système qui se veut performant. La solution restante est la séparation de la pile locale en deux piles, la pile des environnements et la pile des points de choix. La pile des points de choix représente donc la partie parallèle de l'exécution, étant donné que l'unité d'échange de travail est le point de choix. Le reste des piles représente l'exécution séquentielle de Prolog. Les points de choix sont aussi chaînés du plus ancien au plus récent pour faciliter les transferts et les traitements ultérieurs, dont la gestion de la coupure.

Gestion mémoire

La gestion d'une exécution sur plusieurs nœuds d'un système pose des problèmes d'adressage mémoire. En effet, vu que nous copions des portions de piles remplies de références entre elles, il faut s'assurer que l'espace d'adressage entre les différents nœuds est cohérent. Le moteur WAMCC intègre déjà une partie de ces fonctions en permettant l'usage des fonctions d'allocation d'une zone mémoire virtuelle à base fixe. Néanmoins, la gestion mémoire de la table des atomes est réalisée par l'allocation standard du langage C. Cela provoque des différences de valeurs d'adresses qui altèrent le fonctionnement en parallèle. Les fonctions d'allocation de mémoire standard sont donc elles aussi remplacées par des fonctions utilisant une partie de la mémoire virtuelle basée.

Gestion des traces

Enfin, pour intégrer la prise de traces, certaines parties de la machine abstraite ont été légèrement modifiées par l'apport d'instructions supplémentaires ou l'agrandissement des structures de données, afin de stocker les informations de trace. Cela peut engendrer un léger surcoût à l'exécution, mais le fonctionnement général de la machine abstraite demeure inchangé.

5.3.4 Modifications générales lors de la compilation

Le compilateur de WAMCC doit subir des modifications mineures afin de tenir compte de l'exécution parallèle du programme. Un travail bien plus important est à fournir pour la gestion de la coupure.

Exécution parallèle

Le programme Prolog généré par WAMCC n'est plus une entité indépendante. Il est intégré dans l'ensemble de tâches composant l'application parallèle. Aussi la phase de compilation doit être modifiée pour produire un module en langage C compatible avec notre système.

Plus important, pour éviter l'incohérence de certaines variables lors des transferts, nous avons momentanément supprimé l'optimisation d'appel terminal [AïtKaci90]. Cette incohérence est due à la séparation de la pile locale en pile de point de choix et d'environs dont la conséquence est que les points de choix ne protègent plus les environs. Les performances séquentielles ne sont pas réduites, mais la gestion mémoire est ainsi moins optimisée. En apportant quelques corrections à la machine abstraite, pour tenir compte de la nouvelle configuration des piles, il sera possible de bénéficier de nouveau de l'optimisation d'appel terminal.

Gestion parallèle de la coupure

La gestion de la coupure implique tout d'abord la phase de compilation. Nous présentons les principales interventions à apporter au compilateur pour la prise en compte automatique de la gestion de la coupure parallèle. En premier lieu il faut compter le nombre de coupure apparaissant dans chaque clause d'un prédicat. En second lieu, il faut générer les instructions de gestion des niveaux de coupure s'il y a des coupures dans la clause.

Il faut noter que la coupure superficielle, qui ne détruit que le dernier point de choix créé, a une action qui est toujours locale quand le point de choix courant n'est pas exporté, donc n'engendre pas de message de la part du système, et peut être purement et simplement ignorée pour le traitement parallèle.
Elle doit cependant tenir compte de l'ordre séquentiel, et devrait être suspendue si elle apparaît dans le champ d'une autre coupure. Or dans ce cas, les branches seront forcément coupées par l'une ou l'autre des coupures. En conséquence, la coupure superficielle peut toujours être effectuée immédiatement, sans mécanisme de gestion parallèle, et ne provoque donc pas de modifications de sa compilation.
Dans un système comme Muse, il n'est pas possible de faire cette distinction, car les plus récents points de choix peuvent être exportés.

La gestion des différentes variantes de la coupure et des disjonctions, dépendant plus de techniques de compilation que du mécanisme de gestion en parallèle, n'est actuellement pas prise en compte.
Enfin, dans la version actuelle du système, le compilateur ne génère pas encore automatiquement le code WAM incluant les fonctions de gestion des niveaux de coupure.

5.4 Structure logicielle du prototype

Après la présentation de l'organisation générale du prototype, chaque entité est détaillée ainsi que les principaux éléments de son implantation.

Le prototype de système que nous avons réalisé comprend deux parties distinctes, l'environnement séquentiel et le support d'exécution parallèle d'un programme Prolog (Figure 5-2). Actuellement ces deux parties intègrent le même noyau Prolog séquentiel comme base. Toutefois, les noyaux Prolog de chaque partie pourraient être différents, à condition qu'ils respectent la même sémantique du langage.

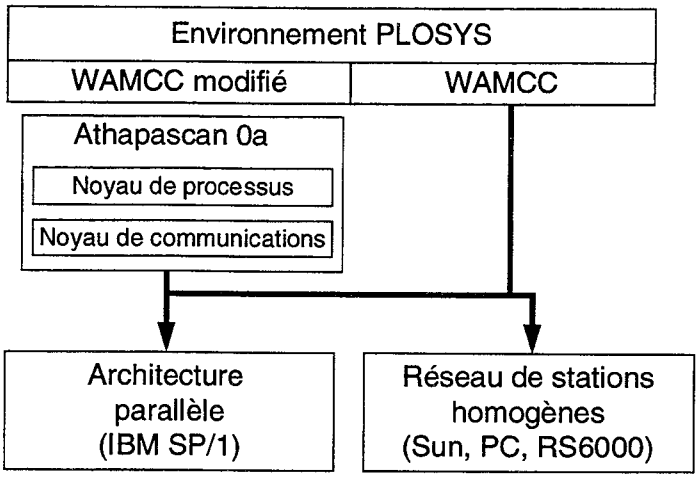


Figure 5-2 - Organisation logicielle du prototype

5.5 Le système parallèle

Le système d'exécution parallèle de PLoSys comprend trois types de tâches : la console, l'ordonnanceur et les travailleurs (Figure 5-3). Les différentes tâches sont implantées sur des processeurs virtuels, elles sont indépendantes entre elles et n'échangent des informations que par appel des différents services qu'elles mettent à disposition les unes aux autres.
L'implantation de ces tâches suit exactement le schéma défini par le projet.

Les différentes unités introduites dans la présentation du système sont implantées sous deux formes :

- La première est le processus léger, subdivision de la tâche, qui avec la bibliothèque parallèle actuelle (Athapascan-0a) est modélisé comme un service.

- La seconde, est un découpage virtuel, représenté par un module de programme indépendant. Celui-ci fournit plusieurs fonctions interfaces pour l'accès à l'unité.

Certaines unités font l'objet d'une implantation répartie entre l'ordonnanceur et les travailleurs, comme l'unité de régulation de charge, d'autres sont locales aux tâches.

Nous présentons successivement les différentes tâches de notre système, ainsi que les unités qu'elles manipulent, et les services qu'elles fournissent. L'unité de régulation de la charge et celles de gestion des effets de bord font l'objet de paragraphes distincts.

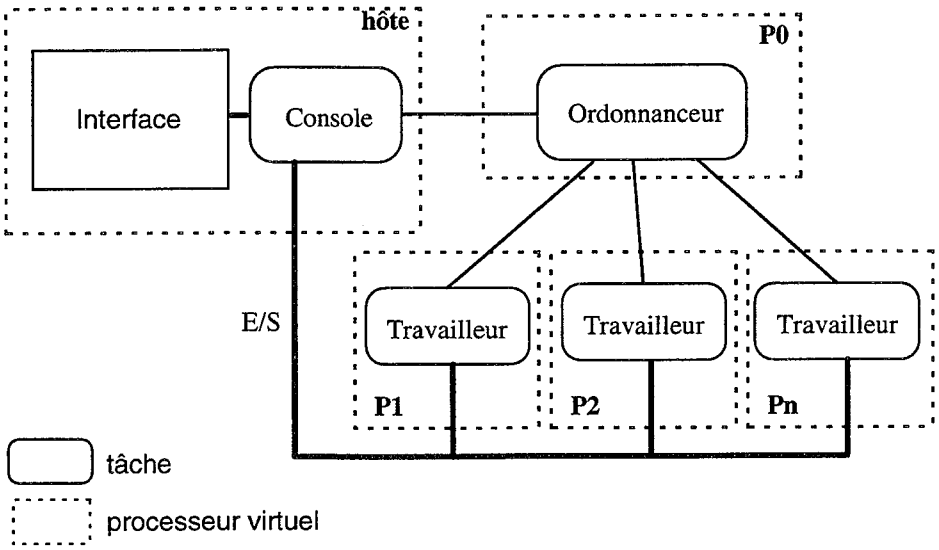


Figure 5-3 - Les tâches du système d'exécution parallèle

5.5.1 La console

Cette tâche remplit un double rôle, chargée à la fois de gérer l'exécution parallèle, et aussi de relayer les entrées et sorties effectuées par le programme Prolog.

Gestion de l'exécution parallèle

La première fonction de la console est la gestion de l'exécution parallèle d'un programme. La console permet de spécifier les paramètres de l'exécution, comme les machines à réserver, le nombre de travailleurs, le programme Prolog exécutable, le programme ordonnanceur utilisé, les paramètres éventuels pour la régulation de charge. A son lancement, la console lit un fichier de configuration pour connaître les caractéristiques du système, notamment les noms des nœuds du réseau. Puis, la console crée les tâches ordonnanceur et travailleurs sur les machines associées par la configuration. L'exécution est ensuite débutée par appel du service approprié dans la tâche ordonnanceur. A la fin de l'exécution, la console récupère les données de l'éventuelle prise de traces, puis détruit les tâches ordonnanceur et travailleurs avant de terminer sa propre tâche.
Pour faciliter la tâche de l'utilisateur, la console est aussi pourvue d'une interface de contrôle réalisée à l'aide de la bibliothèque spécialisée Tcl/Tk, qui permet l'ajustement de la plupart des paramètres du système. A moyen terme cette interface servira aussi à la répercussion des entrées et sorties écran, ainsi qu'à la visualisation et l'analyse des traces.

Gestion des entrées et sorties

La deuxième fonction de la console est la gestion des entrées et sorties effectuées par le programme Prolog. A cet effet, la console propose deux services Athapascan-0a que les

travailleurs accèdent pour leurs opérations d'entrées ou de sorties de bas niveau. Le premier service est dédié aux opérations de sortie, tandis que le second contrôle les opérations d'entrée. Ainsi chaque travailleur ayant à exécuter une opération d'entrée ou de sortie fera appel à l'un des deux services gérés par la tâche console.

- *Service des entrées*
  Il reçoit en paramètres le type d'opération à effectuer, et les différents descripteurs nécessaires ainsi que le nombre d'octets à lire. Le retour du service contient les données lues sur le dispositif d'entrée sélectionné par l'appel.
  Les fonctions supportées sont la lecture d'octets, l'ouverture ou la fermeture d'un dispositif d'entrée.

- *Service des sorties*
  Il reçoit en paramètres d'appel le type de l'opération à effectuer, les différents descripteurs nécessaires, et les données sous forme d'un flot continu d'octets. Le retour du service contient le résultat de l'écriture des données sur le dispositif de sortie sélectionné par l'appel.
  Les fonctions gérées sont l'écriture d'octets, l'ouverture ou la fermeture d'un dispositif de sortie.

Ces deux services n'ont aucun moyen de récupération d'erreur et transmettent directement les éventuelles erreurs rencontrées. La mise en forme des données est réalisée au niveau du travailleur, les services de la console ne gèrent qu'un flux d'octets ou de mots. Le traitement des entrées peut être optimisé en permettant l'interprétation des données, pour lire par exemple directement un terme Prolog en une seule requête de la part du travailleur, au lieu d'un nombre de requêtes lié à la taille en octets ou mots du terme lu.

5.5.2 Ordonnanceur

L'ordonnanceur rassemble de nombreuses fonctions dont le but est la gestion de l'exécution parallèle d'un programme Prolog, à l'aide des travailleurs. Ces fonctions sont dispersées dans les différentes unités de l'ordonnanceur (Figure 5-4) :

- *La régulation de charge*
  L'unité de régulation de charge est implantée dans un module indépendant, et invoquée depuis les services de collecte des charges et de traitement des demandes de travail. Cette unité est détaillée dans le paragraphe 5.6.1.

- *Le maintien de l'ordre séquentiel*

L'unité de gestion de l'ordre séquentiel, module indépendant, est intégrée dans le code de l'ordonnanceur et est appelée notamment lors du traitement des requêtes de demande de travail. Cette unité comprend les fonctions de maintien de la liste de l'ordre séquentiel. La principale fonction est la mise à jour de la liste, qui est invoquée quand un travailleur signale une inactivité, ou quand un travailleur acquiert du travail. De plus chaque tentative d'importation de la part d'un travailleur est numérotée. Ce numéro est associé à l'élément de la liste séquentiel correspondant au travailleur. Ainsi, les différents

éléments de la liste séquentielle représentant du travail consommé par le même travailleur peuvent être identifiés sans ambiguïté. Cela permet ensuite de pouvoir gérer correctement les effets de bord.

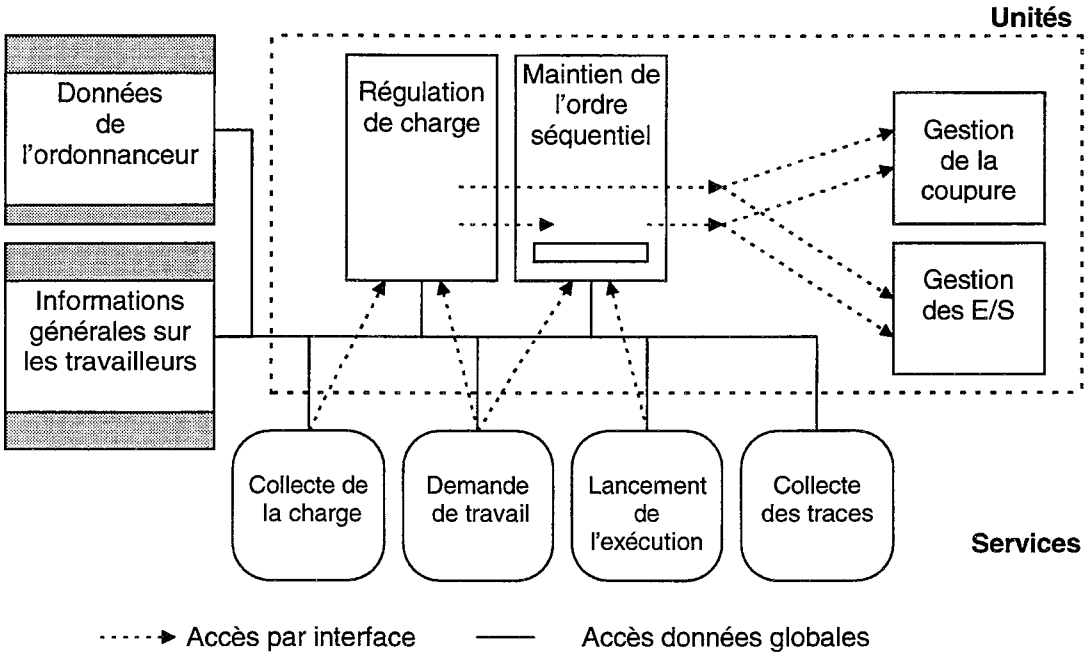


Figure 5-4 - Implantation de l'ordonnanceur

- *Les gestions des effets de bord.*
  Les unités de gestion des effets de bord sont implantées chacune dans un module indépendant, et disposent toutes de fonctions d'interface. Chaque unité ne gère qu'un seul type d'effet de bord. Ces unités sont décrites dans les paragraphes suivants (5.6.2)

Pour que la tâche console et les travailleurs puissent coopérer avec la tâche ordonnanceur, celle-ci contient les services suivants (Figure 5-4) :

- *Initialisation de la tâche et terminaison de la tâche.*
  Ces deux services sont utilisés par la bibliothèque de communication et sont déclarés automatiquement. Le service d'initialisation sert aussi à déclarer au noyau exécutif de la bibliothèque tous les autres services gérés par la tâche.
- *Lancement de l'exécution.*
  Ce service, appelé par la console, marque le départ de l'exécution parallèle d'un programme en démarrant le service d'évaluation de chaque travailleur (Figure 5-5).

- *Collecte de la charge des travailleurs.*
  Le service est régulièrement appelé par les travailleurs, afin de connaître l'état global du système. Ce service est chargé aussi de classer les travailleurs du système, à l'aide d'une fonction de l'unité de régulation de charge.
- *Traitement des demandes de travail.*
  Ce service est appelé dès qu'un travailleur n'a plus de travail à effectuer. Il utilise alors la fonction d'appariement de l'unité de régulation de la charge pour tenter de donner du travail au travailleur l'ayant invoqué.
- *Collecte des traces d'exécution de l'ordonnanceur.*
  Ce service permet de récupérer dans la tâche console, les traces obtenues de l'ordonnanceur.

5.5.3 Les travailleurs

Chacun des travailleurs, outre le moteur Prolog, contient aussi de nombreuses autres unités, et les quelques services permettant l'interaction avec les autres tâches du système Prolog (Figure 5-5).

Les services inclus dans la tâche travailleur sont :

- *L'initialisation de la tâche et terminaison de la tâche.*
  Ces deux services sont utilisés par la bibliothèque de communication et sont déclarés automatiquement. Le service d'initialisation sert aussi à déclarer au noyau exécutif de la bibliothèque tous les autres services gérés par la tâche.
- *L'évaluation.*
  Ce service prépare, puis lance l'évaluation du programme avec le moteur séquentiel. Le travailleur effectue alors une boucle « travailler / demander des points de choix ». Dans le prototype actuel, l'unité d'importation est incluse dans ce service. Chaque importation est identifiée de manière unique par un numéro d'ordre. Cette identification est nécessaire à la gestion des effets de bord.
- *L'exportation.*
  C'est le service qui est appelé pour valider ou invalider un transfert de points de choix, puis effectuer le transfert. L'accès aux piles est réalisé en exclusion mutuelle avec les autres services. Tous les points de choix compris entre le bas de la pile et le point de choix sélectionné sont exportés. Les autres piles sont actuellement entièrement transférées.
- *La collecte des traces.*
  Comme dans le cas de l'ordonnanceur, ce service est utilisé pour récupérer les traces locales de l'exécution parallèle.
- *L'ordre séquentiel.*
  Ce service est appelé chaque fois qu'un travailleur ou des tampons deviennent les premiers de la liste séquentielle. L'identificateur associé au message permet de connaître le destinataire. Si c'est un tampon en attente, celui sera vidé. Si c'est le travailleur, cela indique que le calcul en cours est le premier de l'ordre séquentiel.

Les effets de bord sont donc effectués immédiatement. Lorsqu'il est invoqué, ce service appelle les différents modules d'effets de bord (voir à partir de 5.6.2).

Les unités du travailleur qui ne sont pas englobées dans un service sont :

- *Régulation de charge.*

Cette unité est chargée de communiquer la charge du travailleur, et de déterminer localement la validité d'une exportation de travail. Les paramètres et les algorithmes employés par cette unité peuvent être modifiés selon le type d'architecture ou de programme. La description précise de cette unité est donnée dans le paragraphe 5.6.1.

- *Les effets de bord.*

Chaque effet de bord dispose d'une unité spécifique dont une majeure partie est localisée sur le travailleur. Les différentes unités implantées sont décrites dans les paragraphes suivant.

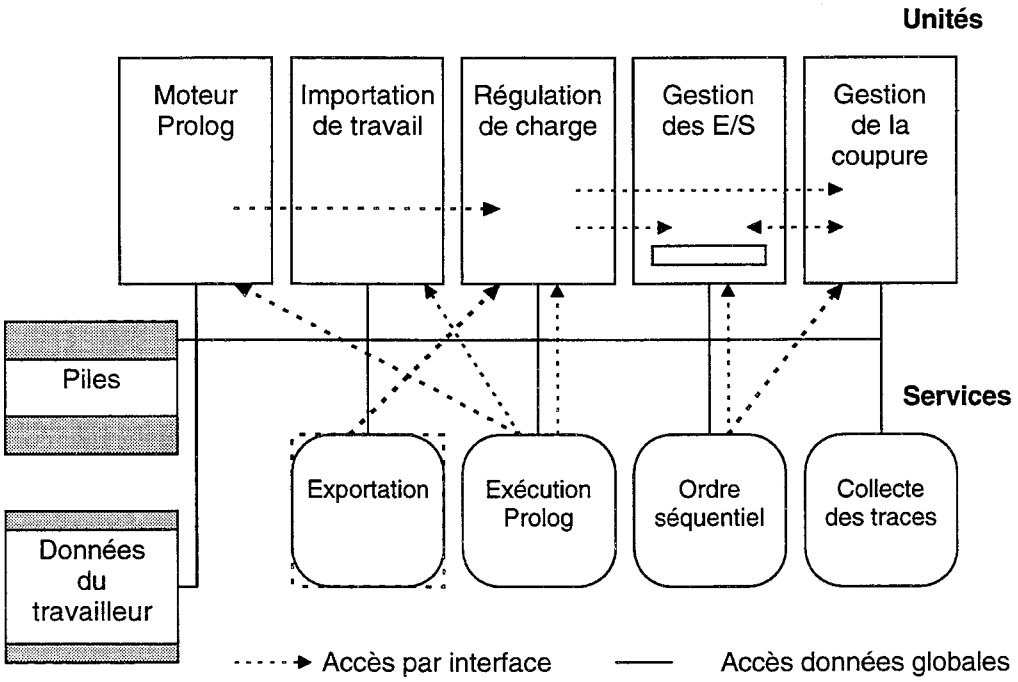


Figure 5-5 - Implantation du travailleur

Chaque travailleur constitue une tâche indépendante qui peut être placée sur un processeur virtuel. Il est donc possible de disposer plusieurs travailleurs sur un processeur physique pour recouvrir les temps d'attente des travailleurs.

5.6 Les unités indépendantes

Nous décrivons maintenant l'implantation des différentes unités de notre système permettant l'exploitation parallèle d'un programme.

5.6.1 L'unité de régulation de charge

Dans le système PLoSys, la régulation dynamique de charge est traitée à deux niveaux. Le premier prend en compte les informations globales du système au sein de l'ordonnanceur tandis que le second ne gère que les informations locales aux travailleurs. L'unité de régulation de charge est logiquement décomposée en deux entités, la première dans l'ordonnanceur et l'autre est répartie sur chacun des travailleurs du système.

Interface pour l'ordonnanceur

- La lecture des paramètres.
  Elle permet d'adapter facilement la politique de régulation de la charge à différentes architectures ou à différents types de programmes. L'utilisateur a ainsi accès aux principaux paramètres de la fonction de prise de décision.

- La collecte de la charge du système.
  Cette fonction permet de rassembler les informations provenant des travailleurs.

- La prise de décision.
  Cette fonction est appelée chaque fois qu'un travailleur inactif envoie une requête pour obtenir du travail. S'il est possible de partager du travail, cette fonction retourne l'identification d'un travailleur candidat à l'exportation.

Interface pour les travailleurs

- Lecture des paramètres locaux.
  Cette fonction permet l'initialisation des paramètres des fonctions locales de l'unité de régulation de la charge.

- L'envoi de charge.
  Cette fonction lit la valeur de la charge sur le travailleur et l'envoie à l'ordonnanceur.

- La validation locale d'une décision d'exportation
  Cette fonction permet la correction de la prise de décision d'une exportation, s'il y a eu un changement d'état notable depuis la dernière information envoyée à l'ordonnanceur.

Problème particulier de l'envoi de la charge

Nous avons choisi de collecter la charge des travailleurs par un mécanisme d'échantillonnage temporel, mais la bibliothèque parallèle Athapascan-0a ne permet ni la gestion d'interruption, ni la préemption de processus. Aussi il n'est pas possible d'utiliser directement un mécanisme d'échantillonnage, et nous devons donc le simuler.

Pour simuler l'échantillonnage, il faut assurer qu'un intervalle de temps correspondant à la période de l'échantillon sépare deux envois de charge. Or, durant l'exécution d'un programme Prolog, seules les instructions manipulant les points de choix sont de nature à faire varier la charge. Aussi, la gestion de l'envoi de la charge ne doit être prise en compte que dans ces instructions. Ainsi, dans chacune des instructions de manipulation des points de choix, il faut déterminer si un délai suffisant s'est écoulé depuis le dernier envoi de la charge. Si c'est le cas, il faudra envoyer la nouvelle valeur de la charge. Nous avons choisi deux méthodes différentes pour calculer ce délai :

- Calcul du temps écoulé
  Quand la charge du travailleur est envoyée à l'ordonnanceur, le temps physique est enregistré. Puis, lorsqu'une nouvelle instruction de manipulation des points de choix est exécutée, le temps physique est comparé à celui mémorisé. Si la différence entre ces deux temps est supérieure à la valeur de la période de l'échantillon, la charge peut de nouveau être envoyée. L'inconvénient de cette méthode est le surcoût non négligeable engendré par le calcul du temps écoulé entre chaque instruction de manipulation des points de choix. Afin de limiter ce surcoût, la gestion de la charge n'est prise en compte que pour la création ou la destruction d'un point de choix.

- Utilisation d'un processus externe.
  Ici la mesure de la période est réalisée par un processus externe qui est activé par le système d'exploitation avec une période correspondant à la période de l'échantillonnage. L'action de ce processus est d'incrémenter un mot en mémoire partagée avec le travailleur. Ainsi, il suffit de tester ce mot dans les instructions de manipulation des points de choix pour savoir si le délai désiré est écoulé. Cette méthode réduit le coût de calcul du délai à un simple test dans chaque instruction de manipulation des points de choix. Le coût du changement de contexte dû à l'activation du processus de comptage semble relativement faible, notamment pour de grandes valeurs de la période de l'échantillonnage.

Les performances obtenues avec les deux méthodes seront comparées dans le chapitre suivant.

5.6.2 La gestion de la coupure

Nous présentons les principales caractéristiques de l'implantation parallèle de la coupure.

La structure du point de choix est légèrement modifiée par rapport au système séquentiel, afin de pouvoir mémoriser le niveau de coupure avant la création du point de choix, celui de la prochaine alternative du point de choix et une marque pour indiquer si le point de choix a été exporté. Dans ce dernier cas, le point de choix contiendra aussi l'identification du travailleur destinataire, celle-ci étant placée dans un champ devenu inutile Ainsi, toute action à effectuer sur le point de choix peut être transmise à son nouveau propriétaire.

La validation et l'invalidation

Si la coupure ou le retour-arrière, sont effectués localement au travailleur, leur gestion est identique à une exécution séquentielle. Par contre si l'action est globale, il faut pouvoir valider ou invalider tous les points de choix exportés.

Mais, si le travail en cours est spéculatif il faut mémoriser les éventuelles actions à effectuer sur les points de choix exportés. Or, si le travailleur effectue une nouvelle importation de travail entre-temps, les points de choix initialement exportés sont écrasés, perdant ainsi toute information pour la validation ou l'invalidation. Ceux-ci sont donc mémorisés dans une table, ainsi que leur nouveau propriétaire (Figure 5-6) et un numéro identifiant l'importation. Les points de choix exportés ne sont mémorisés dans la table que s'ils sont spéculatifs, c'est-à-dire si un de leurs niveaux de coupure est supérieur à zéro.

Adresse PC	ID échange	Importateur	Niveaux de Coupure

Figure 5-6 - Elément de la table des points de choix exporté

La réception d'un message de validation ou d'invalidation

A la réception d'un message d'invalidation, si le point de choix concerné est encore dans la pile, la coupure est effectuée, avec destruction des tampons d'effet de bord associés. Si le point de choix n'est plus dans la pile, en raison d'une nouvelle importation, il suffit de détruire les tampons d'effets de bord associés. Enfin, dans les deux cas précédents, si le point de choix est marqué comme exporté, il faut aussi renvoyer le message d'invalidation au nouveau propriétaire.

A la réception d'un message de validation, si le point de choix est encore dans la pile, les valeurs des niveaux de coupures de tous les points de choix créés à partir du point de choix désigné sont réduites. S'il n'est plus dans la pile, il faut réduire de n le niveau de coupure associé aux tampons d'effets de bord en attente. Enfin, si le point de choix est marqué comme exporté, un message de validation est envoyé au nouveau propriétaire.

La gestion du renvoi

Un problème peut apparaître si des points de choix spéculatifs importés dans un premier temps, sont exportés vers un autre travailleur. En effet, si le travailleur a effectué une nouvelle importation, la pile de points de choix sera modifiée, et il ne restera plus d'information pour retrouver les points de choix exportés. Leur validation ou invalidation risque alors d'être perdue au lieu d'être renvoyée au nouveau propriétaire. Pour éviter cette perte, et donc gérer correctement le mécanisme de renvoi, il faut mémoriser les points de choix importés et les marquer, s'ils sont à leur tour exportés. Comme pour l'exportation, une table contient donc les références de tous les points de choix spéculatifs importés (Figure 5-7). Les deux tables sont fusionnées en une seule afin d'éviter une duplication des informations. Si le champ exportateur est nul, le point de choix a été créé localement. Si le champ exportateur importateur est nul, le point de

choix n'a pas été exporté. Enfin, si les deux champs sont non-nuls, il s'agît d'un point de choix importé puis exporté, et donc concerné par le mécanisme de renvoi, qui utilisera alors les informations de la table.

Adresse PC	ID échange	Exportateur	Importateur	Niveaux de Coupure

Figure 5-7 - Elément de la table de points de choix importés ou exportés

Travail spéculatif

La gestion des coupures et validations dans le champ d'une coupure sont identiques, sauf que l'envoi des messages associés est différé à l'aide d'un tampon. Ainsi, si la coupure les contrôlant est franchie, les tampons sont détruits, et les messages ne seront donc pas envoyés. Dans le cas contraire, si le travail n'est plus spéculatif, les tampons sont vidés, et les messages sont alors envoyés.

En raison des contraintes dues à la bibliothèque Athapascan-0a, la gestion de la coupure n'est pas activée dans le prototype, bien que ses fonctionnalités soient implantées. En effet, le mécanisme d'interruption ou de préemption, nécessaires au fonctionnement des effets de bord n'ont pas été inclus dans cette version de la bibliothèque.

5.6.3 Les entrées et sorties

La gestion des entrées et sorties comprend deux parties, un serveur localisé dans la console du système PLoSys et une partie cliente dans chaque travailleur.

Le serveur exécute toutes les opérations classiques d'entrées et de sorties sur la machine hôte qui l'abrite, tel que décrit en 5.5.1.

Les entrées et sorties sont réalisées par le déroutement des fonctions classiques du langage C des modules correspondants de WAMCC.
Si l'opération à effectuer est une entrée, une requête est envoyée au serveur d'entrées si le travailleur est le premier de l'ordre séquentiel, sinon il est suspendu. La requête ne sera transmise que quand le travailleur aura reçu le signal lui indiquant qu'il est devenu le premier de l'ordre séquentiel.
Dans le cas d'une opération de sortie, il n'y a pas d'attente due à l'ordre séquentiel. Toutes les opérations de sortie sont effectuées immédiatement dans un tampon local au travailleur. Si le travailleur est le premier de l'ordre séquentiel, le tampon est vidé lorsqu'il est plein. Sinon, le tampon est agrandi. S'il existe plusieurs flux de sortie, un tampon est associé à chacun d'eux.

Tous les tampons associés à une importation sont identifiés par le même numéro d'ordre. Lorsque le travailleur effectue une nouvelle importation, il crée de nouveaux tampons, qui seront donc différents de ceux déjà utilisés. Puis quand le travailleur reçoit un

message lui indiquant qu'il est le premier séquentiel, il peut déterminer quels tampons sont concernés par comparaison du numéro d'ordre envoyé par l'ordonnanceur. Les sorties sont effectuées à l'aide d'une communication par tampon, et sont donc correctement ordonnées vis à vis de l'ordre séquentiel.

Cependant, comme pour la coupure, la version actuelle du prototype de dispose pas des entrées et sorties, bien que les mécanismes de gestion soient déjà intégrés.

5.6.4 Les autres effets de bord

Comme la bibliothèque Athapascan-0a ne permet pas actuellement d'implanter de manière efficace les effets de bord ; les prédicats collecteurs et les prédicats dynamiques ne sont pas pris en compte dans le prototype, et leur implantation n'a pas fait l'objet d'un travail approfondi. Mais, le fonctionnement de ceux-ci sera cependant très proche de celui de la coupure et des entrées et sorties. Les mécanismes principaux, comme la gestion des tampons ou la gestion des messages de contrôle, étant similaires, leur implantation en sera facilitée.

Il restera, pour les prédicats dynamiques à implanter le protocole d'insertion et de suppression d'un prédicat. Pour les prédicats collecteurs, il faut encore réaliser le mécanisme de collecte des solutions.

5.7 Outils d'analyse des performances

Dans PLoSys, l'analyse des exécutions est réalisée avec le système séquentiel et les performances avec le système parallèle. Le système séquentiel produit des informations pour l'analyse a priori de l'exécution parallèle d'un programme tandis que le système parallèle produit des informations d'analyse a posteriori.

5.7.1 Système séquentiel

Afin de prévoir les performances du système parallèle, la version séquentielle de WAMCC est modifiée pour pouvoir produire des traces lors de l'exécution d'un programme Prolog.
Ces traces sont essentiellement utilisées par l'outil d'évaluation des fonctions de régulation de charge [Kannat94] afin d'analyser différentes fonctions de régulation et de déterminer celles qui conviennent le mieux à notre système dans le cas général. L'exécution d'un programme Prolog ainsi compilé crée un fichier contenant un graphe de tâches acyclique orienté compatible avec l'outil d'analyse. Ces tâches représentent le calcul effectué entre deux nœuds OU de l'arbre d'exécution, et les arcs orientés indiquent la relation de précédence temporelle des tâches.

Un second type de traces permet de déterminer une sorte de classe du programme, en fonction du type d'opérations de la WAM qu'il comporte et de la forme de son graphe d'exécution, pour associer les fonctions de régulation adaptées à son exécution parallèle, et qui donneront ainsi des performances optimales. Le résultat de l'exécution fournit un fichier contenant le décompte des différents groupes d'instructions de la WAM utilisées par le programme.

5.7.2 Système parallèle:

Le système parallèle peut être compilé soit pour l'exécution optimisée, soit pour l'exécution avec prise de trace. La première option est celle utilisée normalement, lorsqu'un programme Prolog est stable. L'exécution avec traces est utilisée pour déterminer la meilleure configuration logicielle et matérielle pour un programme ou une classe de programmes.

Il faut noter que les traces générées ne sont pas exactes car leur traitement introduit un surcoût, même s'il est minime, et peuvent peut-être modifier légèrement le comportement d'une exécution. Cependant, l'exécution non tracée étant déjà complètement indéterministe, on négligera l'influence des traces sur le comportement de l'exécution. Enfin, cette influence dépend aussi de l'architecture matérielle et du système d'exploitation, en particulier pour les fonctions associées à la mesure du temps. Par exemple, sur l'IBM SP/1 les processeurs des nœuds disposent d'un registre dédié au comptage du temps, ce qui permet une lecture rapide. Par contre, sur les compatibles PC ou les stations de travail, il faut faire appel à la librairie du langage C.

Les principales données fournies par les traces parallèles sont:

- Le temps d'exécution total du programme, compté depuis la console, ou depuis l'ordonnanceur. Ces deux temps peuvent être différents en raison des communications effectuées lors de la terminaison.

- Le nombre total de requêtes d'importation et d'exportation pour chaque travailleur, en différenciant celles ayant échoué.

- Les tailles totales des données (piles) transférées pour chaque travailleur, en exportation et en importation.

- Le temps total d'exécution pour chaque travailleur, ainsi que le temps de calcul, de communication, d'importation de travail, et d'inactivité. Le temps de communication comprend les envois de charge, l'exportation car avec l'implantation actuelle il est difficile de déterminer de manière précise leurs parts respectives. En effet ces temps se recouvrent, et il n'est pas possible de mesurer la durée de l'exportation ou de l'envoi de la charge directement.

Ces traces ont permis l'étude des premiers résultats de notre système présentée dans le chapitre suivant.

6. Résultats

Ce chapitre présente les résultats préliminaires obtenus avec le premier prototype du système PLoSys. Ils ont été obtenus à partir de différentes évolutions du prototype, sur plusieurs architectures de machines.
Après avoir rappelé quelques caractéristiques techniques des machines utilisées, ainsi que les performances du moteur Prolog séquentiel, nous présentons les performances obtenues par notre système en exécution parallèle. Puis nous analysons différentes exécutions, afin de cerner les points faibles de notre prototype.

6.1 Caractéristiques des systèmes utilisés

Nous rappelons brièvement les principales caractéristiques des systèmes utilisés pour tester le prototype de PLoSys. Les mesures de performances séquentielles ont été réalisées sur toutes les machines disponibles. Cependant, les mesures prises durant les exécutions parallèles proviennent uniquement de la machine IBM SP/1.
L'utilisation d'autres machines en réseau est plus délicate si l'on veut des mesures précises. Il est très difficile d'en prévenir l'utilisation par des personnes extérieures durant des tests de plusieurs minutes. Il est aussi impossible de s'assurer que le réseau ne soit pas perturbé durant les mesures.

6.1.1 IBM SP/1

Le cœur d'un nœud de la machine IBM SP/1 est le processeur IBM Power1 cadencé à 60MHz [Seznec95]. Ce processeur gère une mémoire de 64Mo associée à plusieurs giga-octets d'espace disque. Les communications sont effectuées soit par un classique réseau Ethernet à la vitesse de 1Mo/s, soit par un réseau spécialisé à la vitesse de 14Mo/s.
Chaque nœud de la machine est exploité à l'aide du système AIX d'IBM. Les programmes de test de notre système peuvent être exécutés sans débordement de l'espace mémoire.

L'utilisation conjointe de WAMCC et de la librairie Athapascan-0a sur cette machine n'est possible qu'avec l'utilisation du réseau Ethernet. En effet, la librairie PVMe dont dépend Athapascan-0a pour l'utilisation du réseau rapide ne peut être liée avec le programme Prolog compilé, en raison d'une incompatibilité des compilateurs utilisés.

Les performances obtenue avec la librairie sur le réseau Ethernet sont les suivantes :

- Temps de latence Athapascan-0a : 1100µs, ce qui représente un nombre très important de cycles du processeur (66000 cycles, soit environ 16500 multiplications entières).
- Le débit maximum est de 1Mo/s, mais d'après l'étude donnée dans [Christaller96], il dépend fortement de la taille des messages envoyés. Il voisine les 10Ko/s pour des messages de taille inférieure à 10 octets, passe à 100Ko/S environ pour des messages d'une centaine d'octets et culmine à 1Mo/s dès que les messages dépassent les 1000 octets.

Les performances d'Athapascan sont évidenmment bien meilleures sur le réseau spécialisé, réduisant la latence à 400µs et le débit passant à environ 14Mo/s.

6.1.2 Le réseau de stations ou de serveurs de terminaux

La plupart des serveurs disponible à l'IMAG sont du type multiprocesseurs symétriques d'origine Sun. Il sont généralement exploités avec le système Solaris 2.5.1, tout comme les stations de travail monoprocesseur. La quantité de mémoire disponible est telle que les programmes de test de notre système sont loin de saturer le système.

Les communications sont gérées par un réseau Ethernet standard à la vitesse de 1Mo/s. Le temps de latence de la librairie Athapascan-0a n'est pas donné pour une telle configuration. Mais il doit être logiquement voisin de celui obtenu sur la machine IBM avec le réseau Ethernet.

Le nombre important de processus gérés, même dans les heures de faible charge ne permet pas de mesure précise du temps d'exécution. De plus, le réseau est généralement très vite perturbé en raison de son étendue physique. Aussi nous n'avons pas effectué de mesures d'exécutions parallèles avec ce type de machines.

6.1.3 Réseau compatible PC

Ce type d'architecture, représente une solution financièrement économique pour la mise au point et le test d'application parallèle. Un ensemble d'ordinateurs personnels, à base de processeur Pentium d'Intel, cadencé à 133MHz ou plus, sont réliés par un réseau Ethernet. Le point faible est encore la durée élevée des communications, mais celui-ci peut être diminué à par l'installation d'un réseau rapide dédié.

Chaque nœud du réseau comprend une machine compatible PC complète, exploitée par un système Linux 2.0 ou Solaris 2.5.1. L'espace mémoire et l'espace disque sont aussi largement suffisants pour les tests de notre application.

Le prototype du projet PLoSys a été développé en majeure partie sur ce type de machine.

6.2 Performances séquentielles de PLoSys

Les résultats obtenus par notre système dépendent en grande partie des performances de WAMCC, mais aussi du surcoût apporté par la gestion du parallélisme.

6.2.1 Performances de WAMCC

Nous rappelons dans ce tableau les performances relatives de WAMCC et de différents systèmes Prolog afin de pouvoir confronter par la suite les performances de notre système à celles-ci. Ce tableau est un condensé des mesures que l'on peut consulter dans [Diaz95].

	WAMCC 2.2	BinProlog 3.0	XSB-Prolog 1.4.0	SWI-Prolog 1.8.11	Sicstus 2.1 émulé	Sicstus 2.1 natif	Quintus 2.5.1
Queens(8)	0.70	0.92	1.56	3.45	0.98	0.37	0.58
Ham	4.33	5.28	8.84	12.65	4.33	5.05	2.09
Boyer	3.45	6.70	11.5	21.2	4.94	2.35	2.85
Vitesse relative	1	0.5	0.37	0.18	0.63	1.62	1.03

Tableau 6-1 - Performances relatives des systèmes Prolog (temps en secondes)

Les performances de WAMCC sont très bonnes face aux systèmes Prolog réputés comme Sicstus ou Quintus.

6.2.2 Les programmes de test

Afin de pouvoir comparer notre système aux systèmes existants, il semblait logique d'utiliser les mêmes programmes de test. Cependant, avec l'évolution rapide du matériel, et surtout des processeurs, ces programmes sont devenus trop petits en terme de temps d'exécution. Par exemple, le classique problème de placement de 8 reines sur un échiquier (méthode générer et tester incrémentale), qui s'exécutait en quelques secondes il y a quelques années (1990 à 1993), ne prend plus que quelques dixièmes de secondes sur des machines récentes (>1994), soit un rapport de plus d'un ordre de grandeur (Tableau 6-2).

Queens (8)	Machine/Système	Processeur/Vitesse	Temps (s)
WAMCC	P.C. (Linux)	Pentium, 133MHz	0.11
WAMCC	Sun (Solaris)	SuperSparc à 60MHz	0.17
WAMCC	SP/1 (AIX)	Power1 à 60MHz	0.22
WAMCC	Sun OS	Sparc 2	0.70
Muse	TC2000	M88100	1.79
Muse	S.Symmetry	i386	7.83
Aurora	Sparc Center 2000	Sparc ?	0.81
Aurora	TC2000	M88100	2.85
Aurora	S.Symmetry	i386	6.91
Malaga	SuperNode	T800 à 20MHz	16.7
OPERA	SuperNode	T800 à 20MHz	3.89
Sicstus 3	SP/1 (AIX)	Power1 à 60MHz	0.25

Tableau 6-2 - Temps d'exécutions pour différents systèmes (en secondes)

Bien sûr, ces différences comprennent aussi celles dues au moteur Prolog, mais elles montrent tout de même qu'il est possible d'offrir une confortable puissance de calcul à l'aide d'ordinateurs personnels!

Il faut aussi remarquer, que les systèmes comme Muse et Aurora, afficheraient de bien meilleures performances sur des machines à mémoire commune récentes telles que la CONVEX Exemplar SPP1000 [Baetke95], qui permet d'après le constructeur d'utiliser jusqu'à 128 nœuds de calcul (HP 7100, 200MFlops).

Les programmes utilisés pour tester notre système sont donnés en annexe.

- **Cavalier** : Il s'agît de parcourir un échiquier, à l'aide du mouvement du cavalier du jeu des échecs, en passant par toutes les cases une fois et une seule. Ce programme permet de trouver toutes les solutions, avec d'éventuelles répétitions. Le nombre de points de choix générés est très important. Trois versions sont utilisées :

  K1_5 : générer et tester simple, à partir d'une case, on génère tous les mouvements possibles, puis on teste leur validité, et on déplace le cavalier pour continuer. La taille de l'échiquier est de 5×5 cases, le programme manipule environ 69 millions de points de choix.

  K2_5 : générer et tester optimisé, le prochain mouvement est pris à partir des cases restantes, puis on teste s'il est possible, la recherche génère environ 30 millions de points de choix.

  K3_5 : générer et tester optimisé avec prise en compte de l'alternance entre les cases blanches et les cases noire pour le choix du mouvement, ce qui divise la liste des cases restantes en deux. Le programme génère environ 17 millions de points de choix.

- **Reines :** C'est le classique problème de test de système OU-parallèle. Le programme cherche toutes les solutions pour placer n reines sur un échiquier $n \times n$ cases.

  Q1 : Générer tester incrémental. Le programme tente de placer les reines en tenant compte des reines déjà placées, en restreignant au fur et a mesure la zone de placement, pour limiter le nombre de tests infructueux.

  Q2 : Générer et tester. Cette version plus intuitive est moins optimisée, et favorise en fait l'accroissement de performances dû au parallélisme. Le programme essaye toutes les positions possibles pour chaque reine. Pour 12 reines, le programme génère 3.8 millions de points de choix.

- **Cycle hamiltonien :** Ce programme (Ham3) cherche naïvement les cycles hamiltoniens dans un graphe de 13 sommets et 42 arêtes, et génère environ 25 millions de points de choix.

6.2.3 Performances de PLoSys séquentiel

Pour pouvoir comparer les résultats des exécutions parallèles de PLoSys, et déterminer les facteurs d'accélération obtenus, il faut en quelques sorte étalonner notre système. Ceci est obtenu à l'aide de la mesure de la vitesse séquentielle du système, donc sans les traitements associés au parallélisme, comme les messages d'information de la charge.

Comme il existe deux version légèrement différentes du prototype, nous donnons les temps d'exécution obtenus par les deux versions (Tableau 6-3). La première version, appelée version 1, intègre la simulation de l'échantillonnage par test du temps écoulé depuis le dernier envoi de la charge. La seconde version, appelée version 1.5, utilise un processus externe, et bénéficie de quelques optimisations dans certaines fonctions de traitement des points de choix. Ces différences expliquent les écarts entre les vitesses relatives des deux versions par rapport à la vitesse de WAMCC. Mais dans tous les cas, les variations sont assez faibles et ne dépassent pas 20% du temps d'exécution. Elle sont maximum pour les programmes dont la manipulation des points de choix est majoritaire par rapport au reste des instructions de la machine abstraite.

	K1_5	K2_5	K3_5	K1_5 c	Ham3	Q1(8)	Q1(12)	Q2(9)
WAMCC 2.21	313	205	103	293	114	0.078	39.2	28.8
PLoSys 1	310	201	109	310	112	0.131	39.4	29.2
PLoSys 1.5	263	178	123	260	109	0.082	39.0	29.5

Tableau 6-3 - Temps d'exécution sur Pentium 166Mhz et Linux (en secondes).

Certains programmes sont exécutés plus rapidement par PLoSys que par WAMCC. Ce phénomène semble plus lié à la compilation du langage C qu'à un éventuel gain apporté par PLoSys. En effet, les temps donnés dans le tableau relatif au SP/1 (Tableau 6-4) sont toujours à l'avantage de WAMCC.

L'utilisation de la coupure superficielle dans le programme *K1_5 c* ne favorise pas le système PLoSys. En effet, la coupure ne fait pas gagner assez de temps pour compenser le traitement plus important dans notre système, dû à la mise à jour locale de la charge notamment.

6.3 Exécutions parallèles

Avant de donner des résultats d'exécutions parallèles, nous devons commencer par définir une ou plusieures fonctions de régulations de charge dont la présence est nécessaire à l'exécution du programme parallèle. Puis nous présentons les premiers résultats obtenus, et quelques détails des traces d'exécution de programmes. Enfin nous

présentons notre analyse de ces valeurs, avec les problèmes identifiés et les solutions envisagées.

6.3.1 La régulation de charge

L'architecture adoptée pour PLoSys prévoit le passage d'une régulation de charge centralisée à une gestion totalement distribuée, à l'aide d'une interface de programmation adaptée. Cette caractéristique permet aussi, dans l'état actuel du prototype, de tester différentes fonctions de régulation de charge.

Nous avons utilisé une fonction de régulation simplifiée pour les premières évaluations du système PLoSys, car les effets de bord ne sont pas disponibles actuellement lors de l'exécution en raison de l'absence de préemption du noyau de processus.

Cette fonction de régulation est dérivée d'une étude préliminaire réalisée avec la plate-forme d'évaluation du projet PLoSys [Kannat94], elle possède quatre paramètres:

* le nombre $M \geq 0$ de points de choix à partir duquel un travailleur peut exporter du travail,
* le nombre $D \geq 0$, qui représente le seuil de la norme de la variation à partir duquel le travailleur informe l'ordonnanceur de sa nouvelle charge,
* le temps $F \geq 0$ minimum entre deux envois de la charge depuis un travailleur vers l'ordonnanceur,
* le nombre $N \geq 0$ de points de choix exportés simultanément.

Lorsque qu'un travailleur inactif effectue une requête pour obtenir du travail, la sélection de l'exportateur est déterminé par les charges accumulées par chacun des travailleurs. La fonction de régulation actuelle choisit le travailleur le plus chargé, et dont la charge dépasse le seuil d'exportation M. Si l'exportation réussit, ce travailleur donnera N points de choix.

Il faut aussi noter qu'en raison de la non-préemption des processus dans Athapascan-0a, l'exportation depuis un travailleur ne peut être traitée que lorsque celui-ci a arrêté sa phase d'évaluation Prolog, soit uniquement pendant la phase d'envoi de la charge, qui est volontairement bloquante pour le processus d'évaluation. Sans ce mécanisme, il ne serait pas possible d'interrompre le moteur Prolog. Il faut donc assurer que le travailleur envoie sa charge de manière assez régulière si on veut que le travail puisse se répandre à travers les nœuds. Dans la version actuelle du prototype il faut donc trouver un compromis entre le ralentissement occasionné par l'envoi de la charge, et le temps de prise en compte d'une exportation.

6.3.2 Premiers résultats

Le tableau ci-dessous (Tableau 6-4), regroupe les temps d'exécutions obtenus en séquentiel avec WAMCC et en parallèle avec les deux versions du prototype de PLoSys. Celles-ci, disposent du maintien de l'ordre séquentiel, de la gestion des

tampons des entrées et sorties, de la gestion des niveaux de coupure, mais aucun effet de bord ne peut être utilisé à cause de l'absence de préemption ou d'interruption dans la version d'Athapascan utilisée au moment de ces évaluations.

Cependant, il est possible d'utiliser des programmes utilisant la coupure superficielle (*neck_cut* [AitKaci90]), et des sorties écrans, celles-ci sont simplement ignorées en mode terminal, et visibles pour chaque travailleur en mode de mise au point, dans des fenêtres de l'interface graphique du système.

La fonction de régulation utilisée est celle décrite en 6.3.1, avec les valeurs des paramètres identiques pour tous les programmes. Ce sont les valeurs qui donnent actuellement les meilleures performances globales.

Dans le tableau suivant (Tableau 6-4), le temps de base est celui de WAMCC et de Sicstus 3 en mode émulé. Le temps appelé *séquentiel* représente le temps d'exécution par PLoSys avec un seul travailleur et sans gestion parallèle. Les temps suivants, de 1 à 32, se rapportent aux exécutions parallèles avec le nombre correspondant de travailleurs.

	K1_5	**Q1(8)**	**Q1(9)**	**Q1(12)**	**K3_5**	**Q1(9)**	**Q1(12)**
WAMCC	689	0.38	0.92	94.5	281	0.92	94.5
Sicstus 3	533	0.25	1.08	143	224	1.08	143
	PLoSys 1.5				**PLoSys 1**		
Séquentiel	737	0.39	0.96	98.5	301	0.96	98.2
1	865	0.40	0.99	102	464	1.13	116.5
2	434	0.40	1.00	55.9	235	1.06	61.1
4	218	0.41	1.04	29.3	121	1.02	31.8
8	112	0.42	1.10	16.1	63	1.15	18.2
16	59.7	0.46	1.28	11.0	34	1.15	12.2
32	63.0	0.52	1.83	11.6	22	1.18	15.1

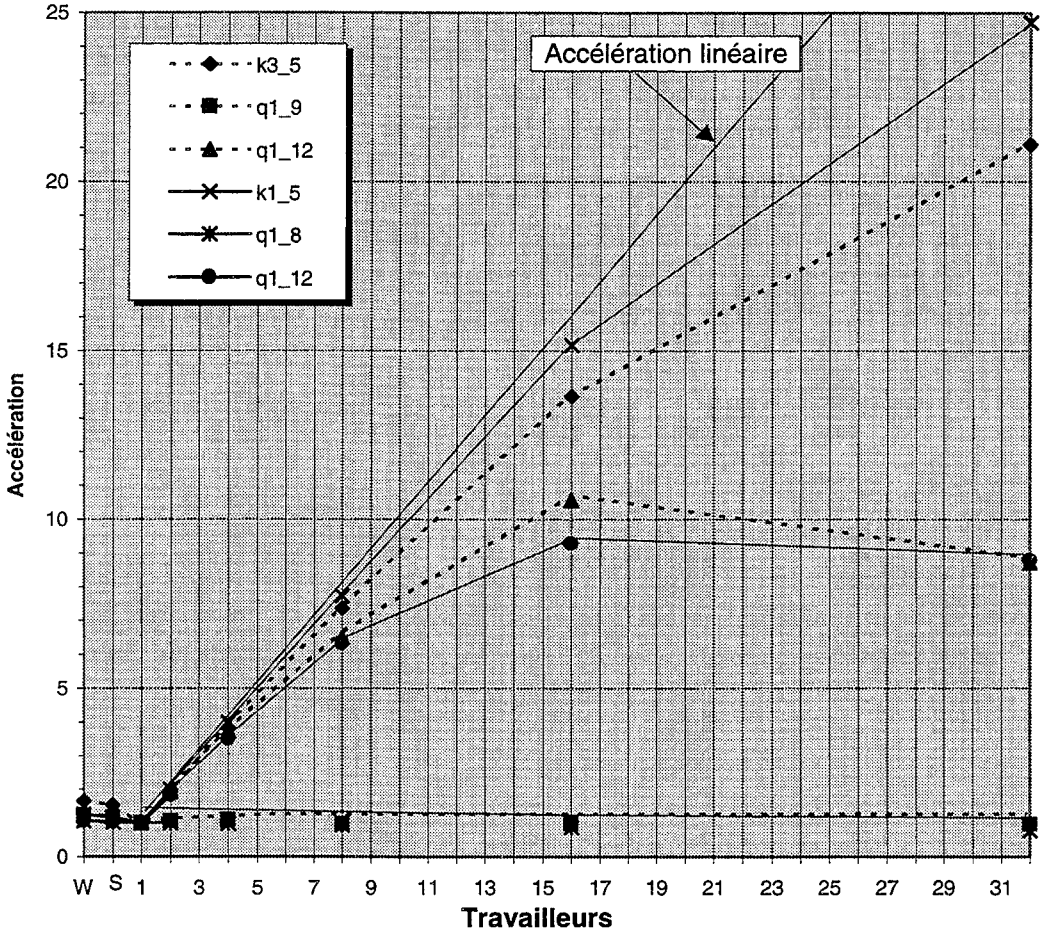
Paramètres de la fonction de régulation de charge : M=2 D=2 F=0.1s N=1

Tableau 6-4 - Temps d'exécution avec le SP/1 et réseau Ethernet

La première remarque vis à vis de ces performances, est que celles-ci représentent le plus mauvais cas d'exécution sur le SP/1 car on n'utilise pas du tout le réseau de communication dédié, mais le simple réseau Ethernet. On peut donc considérer ces performances comme le minimum de ce qu'il est possible d'obtenir.

Il existe deux catégories de programmes, ceux qui bénéficient d'une accélération de performances, car leur exécution dure suffisament longtemps, et ceux qui ne sont pas du tout accélérés. Dès que le temps d'exécution séquentiel, dépasse quelques dizaines de secondes, il devient rentable d'utiliser une exécution parallèle. En revanche, pour les programmes de courte durée, les performances restent de l'ordre de grandeur de

l'exécution séquentielle. Ceci s'explique facilement par le fait qu'un processeur inactif ne demande qu'une seule fois du travail, et donc ne sature pas le système avec de nouvelles requêtes. Donc si le travailleur initial n'a pas assez de travail, il l'épuisera seul, soit parce qu'il n'atteindra pas le seuil d'exportation, soit parce que le temps de réaction à une demande d'exportation est plus long que le temps de surcharge, faisant échouer par la suite le transfert. Ce dernier cas est dû en partie à la non préemption de l'évaluation, qui joue alors un rôle positif.



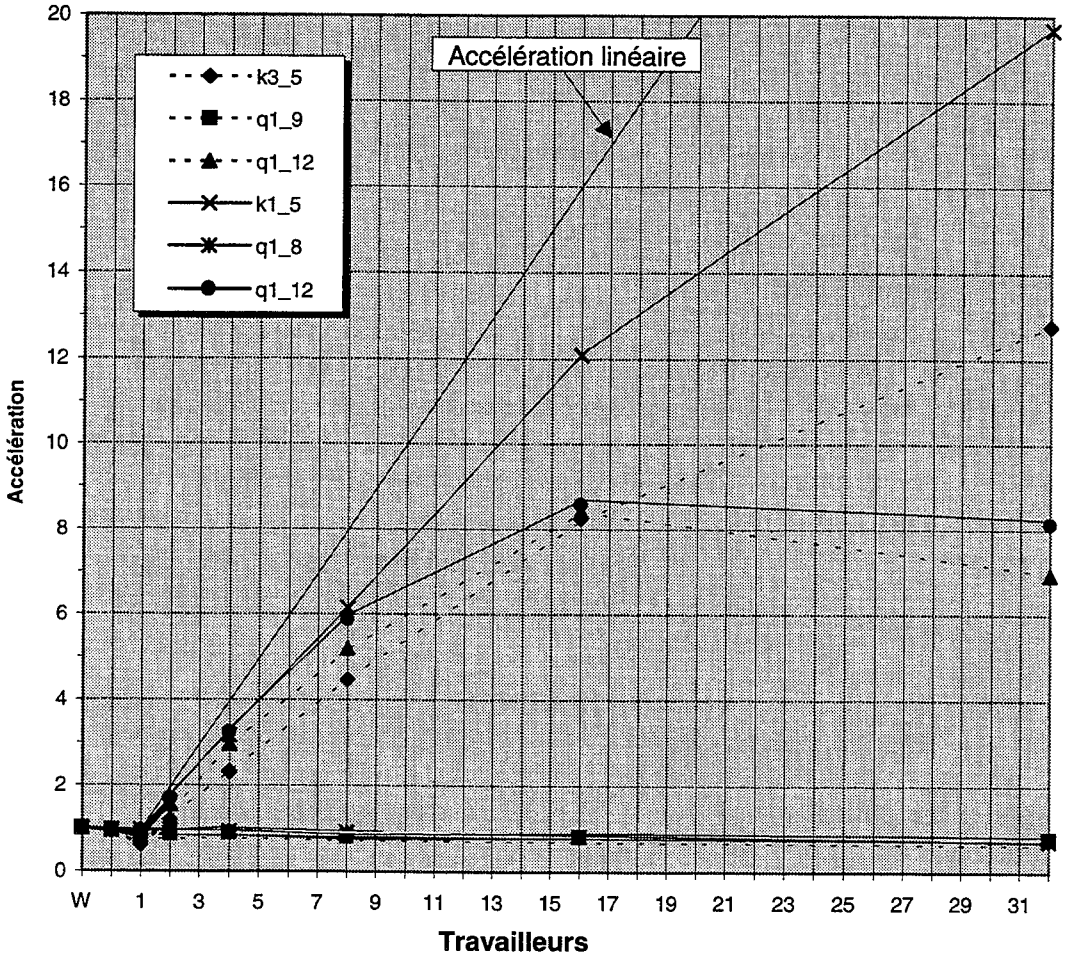
Les courbes en pointillés représentent les accroissements de performances obtenus avec la version 1 du prototype, et les courbes en traits pleins celles de la version 1.5 du prototype. Les deux premières abscisses sont respectivement l'exécution par WAMCC et par la version séquentielle de PLoSys.

Figure 6-1- Accélérations relatives pour quelques programmes

Le surcoût de temps d'exécution de PLoSys en version parallèle n'est pas constant d'un programme à l'autre. Ce phénomène est tout à fait normal, étant donné que ce surcoût est dû à la simulation de la fonction d'échantillonnage, qui est gérée dans les fonctions de manipulation de point de choix. Le surcoût est donc plus important dans les

programmes où le traitement de ces fonctions représente une plus grande part du calcul, tels les différents programmes de parcours d'un échiquier par un cavalier.

L'accélération relative est donnée pour quelques programmes (Figure 6-1), dont les exécutions prennent en compte le choix des paramètres de la fonction de régulation selon le nombre de travailleurs employés afin d'obtenir les meilleures performances possibles.



Les courbes en pointillés représentent les accroissements de performances obtenus avec la version 1 du prototype, et les courbes en traits pleins celles de la version 1.5 du prototype.

Figure 6-2 - Accélérations réelles pour quelques problèmes

Les courbes d'accélération nous montrent clairement que le système fonctionne selon deux modes : soit il accroit notablement les performances, soit celles-ci ne sont pas améliorées. Elles font aussi apparaître que si le nombre de processeurs est trop grand, l'accélération est moins importante ou diminue, soit par manque de travail, soit par saturation de l'ordonnanceur.

Ainsi, malgré les simplifications de l'implantation et les contraintes imposées par la bibliothèque de communication, l'accélération relative est très bonne pour les problèmes de grande taille. Les programmes trop petits sont peu ralentis, car ils s'exécutent sur un seul travailleur, et les éventuels travailleurs inactifs ne consomment pas de ressources car ils sont en attente d'une importation de points de choix.

Les courbes des accélérations réelles (Figure 6-2), mesurées par rapport au temps d'exécution de WAMCC, sont évidemment moins favorable que les précédentes. Cependant pour les programmes de grande taille, les accélérations demeurent bonnes pour la version 1.5 du prototype. Dans tous les cas, l'écart de performances entre les deux versions du prototype est en faveur de la version 1.5. Il est explicable par le moindre surcoût de la fonction d'échantillonnage de la version 1.5 du prototype. L'écart le plus important se situe avec les longs temps d'exécution et un grand nombre de travailleurs.

6.3.3 Une exécution détaillée

Nous présentons quelques caractéristiques d'une exécution parallèle d'un programme à travers une série de graphiques que nous analyserons pour faire apparaître les faiblesses de PLoSys. Les informations illustrées ci-dessous ont été obtenues à l'aide des fonctions de traces intégrées dans la version 1 de PLoSys. Nous n'avons pas mesuré l'influence de ces prises de traces sur le comportement de l'exécution, mais elle doit être relativement faible (moins de 10%), du moins sur le SP/1, où le temps peut être obtenu depuis un registre du processeur. L'influence des traces semble plus importante dans la version 1.5. Le système de prise de traces doit être revu pour être utilisé avec cette version.

Les graphiques suivants ont été obtenus à partir de l'exécution parallèle du programme Q1 avec 12 reines à placer. Ce programme est de taille moyenne en temps séquentiel, 94.5 secondes sur le SP/1, et génère 37771955 points de choix. L'utilisation des piles est relativement faible. La fonction de régulation est celle décrite en 6.3.1 avec pour paramètres : $M=2$, $D=2$, $F=0.1$ et $N=1$.

6.3.3.1 Informations de charge

La figure suivante nous donne le nombre de messages d'information de charge envoyés par chaque travailleur. La durée d'exécution est à peu près de 13 secondes pour 16 travailleurs, avec une fréquence d'échantillonnage de période 0.1 seconde. Le graphique montre aussi une bonne répartition des messages envoyés, que l'on peut attribuer à une bonne répartition du travail (Figure 6-3).

Si on n'utilisait pas de seuil de variation du nombre de points de choix en attente dans la fonction de régulation ($D = 0$), le nombre de messages envoyés serait d'environ 130 par travailleur, au lieu de 45 environ. Cela pourrait créer un goulot d'entranglement au niveau de la tâche ordonnanceur, car le temps de traitement du service de collecte est de quelques millisecondes. Ainsi, pour 130 messages par travailleurs, on obtiendrait 2080 messages, soit autant d'appels au service de collecte de la charge, et un temps de

traitement dépassant la dizaine de secondes, ce qui est très proche du temps total d'exécution.
Néanmoins, cela nous permet de penser qu'il est possible d'améliorer les performances, en diminuant la valeur de la période d'échantillonnage jusqu'à 0,05s (au lieu de 0,1s). Ainsi, le temps de prise en compte d'une exportation devrait être réduit, et donc le gain de l'exécution parallèle serait amélioré.
Or ceci n'est rentable que pour un nombre de travailleurs supérieur à 8, et le gain est alors inférieur à 10%. En effet, sur chaque travailleur, l'attente due à l'arrêt de l'évaluation pendant l'envoi de la charge devient plus fréquente, et réduit alors les performances séquentielles.

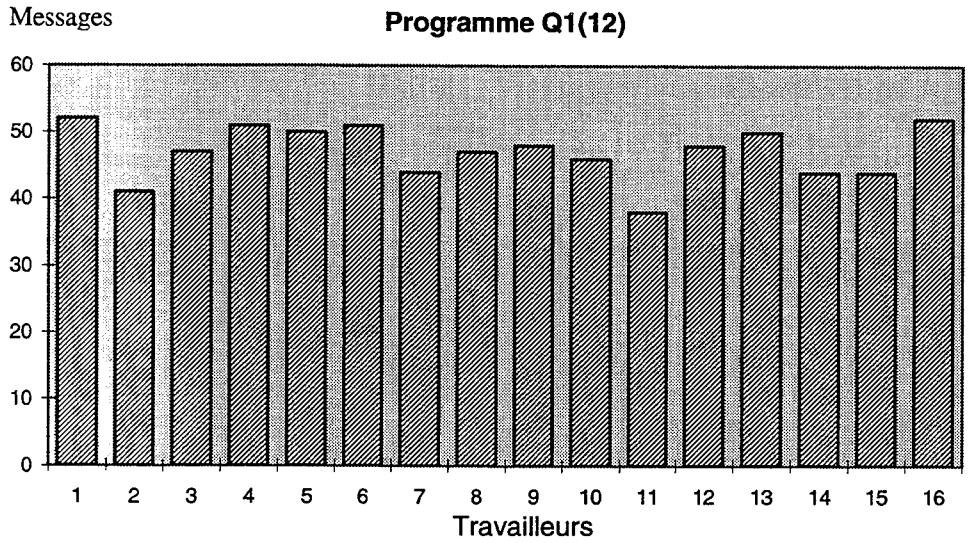


Figure 6-3 - Messages de charge envoyés par chaque travailleur

Dans les mêmes conditions, avec la version 1.5 du prototype, le nombre de messages envoyés devient plus important, environ 95. Ceci peut explique, que l'accélération reste du même ordre entre les deux versions. La version 1 a un traitement séquentiel plus lent en exécution parallèle, favorisant la parallélisation, mais la version 1.5, est ralentie par le nombre de messages échangés.

Enfin, si l'on reprend le résultat de l'exécution du programme *K1_5* (Tableau 6-4) avec 32 processeurs avec la version 1.5 du prototype, et si on diminue la période de l'échantillon, le temps d'exécution croît légèrement. Par contre, si on augmente la période de l'échantillon jusqu'à 0,2s (au lieu de 0,1s), on obtient un temps de 34.5s, ce qui tend à confirmer que le temps perdu lors de l'envoi bloquant de la charge est très important.

La détermination de la fréquence d'échantillonnage est donc délicate, car elle concerne des modification de caractéristiques divergentes du système. Elle constitue actuellement le premier point critique de notre système. Celui-ci disparaîtra en partie quand la prise

en compte de l'exportation ne sera plus conditionnée par l'arrêt du noyau Prolog, permettant ainsi un envoi de charge non bloquant, et donc moins coûteux. Cela ne sera possible qu'avec l'utilisation de la préemption des processus.

6.3.3.2 Transferts de points de choix

La figure Figure 6-4 présente le nombre d'exportations et d'importations réalisées par chaque travailleur du système lors d'une exécution avec 16 travailleurs.

Le nombre d'exportations varie assez fortement selon les travailleurs. Le travailleur 1 ayant en charge le début de l'exécution parallèle réalise le plus grand nombre d'exportation. Comme la politique de régulation actuelle vise à réduire la charge du travailleur ayant le plus de points de choix en attente, il semble donc que le premier travailleur accumule beaucoup de points de choix.
Pour savoir s'il ne disperse pas assez vite son travail, il faudrait pouvoir obtenir l'évolution de sa charge durant l'exécution.

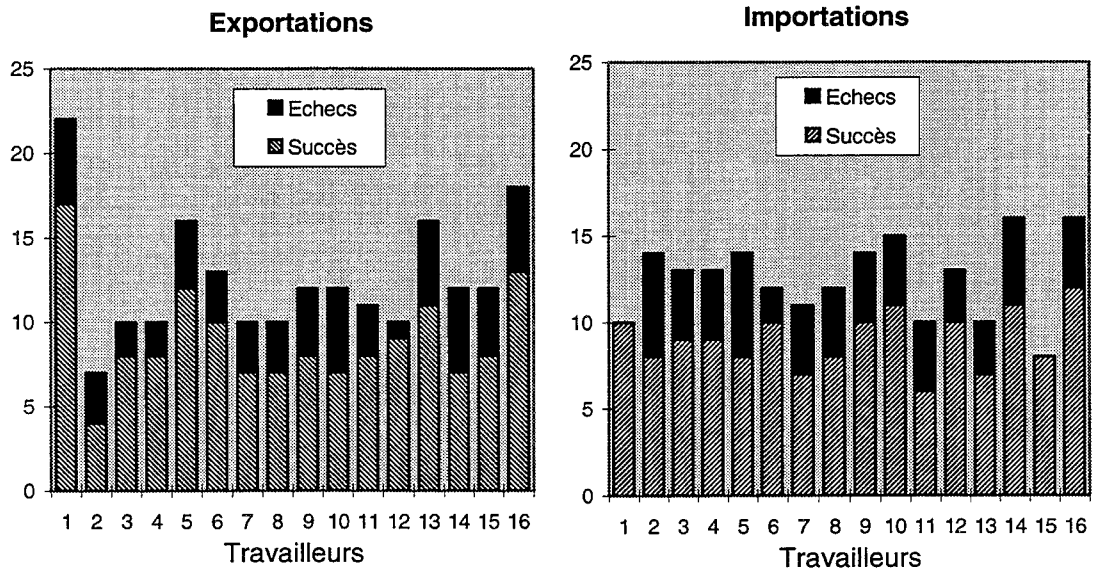


Figure 6-4 - Transferts par travailleur pour le programme q1(12)

Les importations semblent plus régulièrement réparties sur l'ensemble des travailleurs du système. Un point assez étonnant, est que le travailleur 1, qui a commencé l'exécution parallèle, a aussi effectué des importations. Ce phénomène s'est répété pour toutes les exécutions tracées. Il semble donc que la politique de régulation de charge conduise à trop décharger ce travailleur, et peut être les autres. Il se peut aussi, que ceci soit dû à la phase finale de l'exécution, où un grand nombre d'échanges seraient dus au manque de travail global. Le seuil d'exportation doit être ajusté pour limiter ces échanges.
Pour vérifier ces hypothèses, il faudrait pouvoir obtenir la chronologie des échanges.

Il existe aussi un nombre important d'échecs lors des tentatives de transfert. Ces échecs semblent assez régulièrement répartis, tant pour les importations que pour les exportations. Ils peuvent provenir de la lenteur de réaction aux requêtes d'importation, due à la non-préemption des processus.

La figure Figure 6-5 donne le nombre total d'octets transférés lors de l'exécution à 16 travailleurs. Pour ce programme, cette quantité est relativement faible. Si on la rapporte au nombre de transferts effectués, on obtient une moyenne de 835 octets par transfert. Dans ce cas, la latence de communication (1.1ms) est presque plus plus pénalisante que le transfert lui-même (~1ms). Pour ce type de programme, les performances d'un réseau dédié pourraient augmenter celles de notre système.

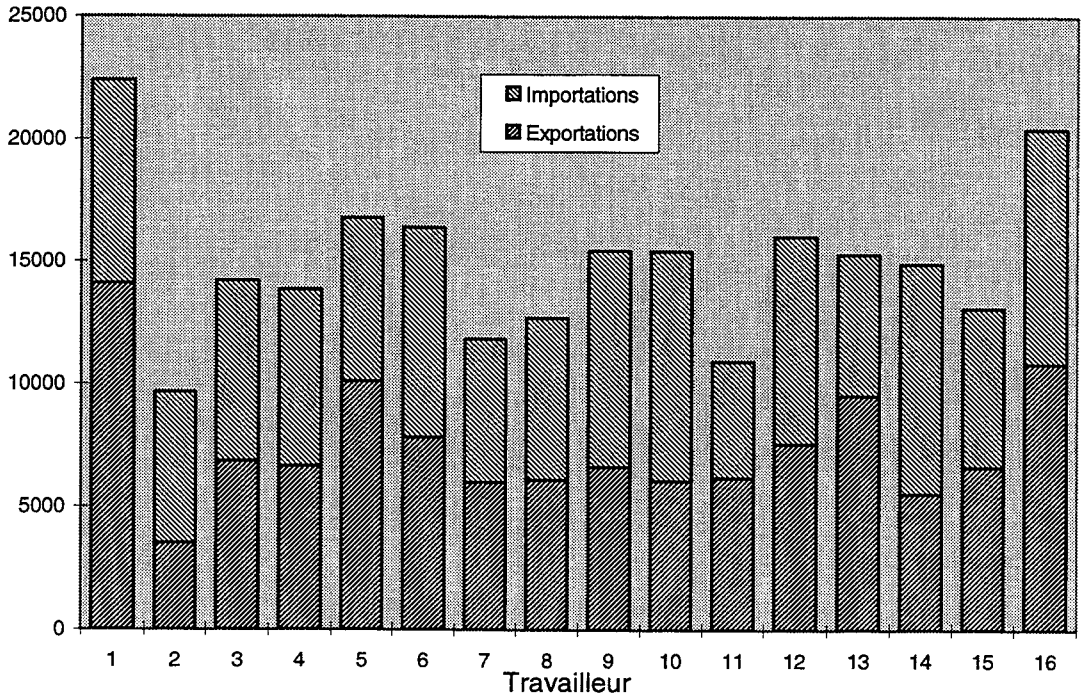


Figure 6-5 - Quantité de données transférées pour le programme q1(12)

6.3.3.3 Efficacité du parallélisme

Les figures suivantes présentent la répartion du temps des différentes phases d'un travailleur, notre système de prise de trace en différencie quatre :

- Inactivité : phase durant laquelle travailleur ne produit aucun travail. Elle est représentée essentiellement par le temps passé à attente la réponse de l'ordonnanceur à requête d'importation.
- Importation : cette phase débute avec la réception, depuis l'ordonnanceur, de l'identification d'un travailleur pouvant exporter des points de choix. Elle se termine, soit lors de le réception de l'échec de l'importation, soit juste avant le

lancement du calcul de la machine abstraite. La durée donnée correspond à la somme du temps de transfert de données et du temps de prise en compte de la demande d'importation. Actuellement ce dernier peut être très variable, non mesurable.

- Communication : dans cette phase se trouvent réunis le temps de communication dû à l'envoi de la charge et le temps d'exportation éventuelle. Ceci est dû au fait qu'une exportation ne peut avoir lieu que pendant l'envoi de la charge, et qu'il n'est pas possible de connaître la durée réelle d'une exportation ou d'un envoi de charge. Les deux actions sont donc confondues.
- Calcul (WAM) : c'est la phase durant laquelle le travailleur exécute le programme Prolog, à l'exclusion de toute autre action.

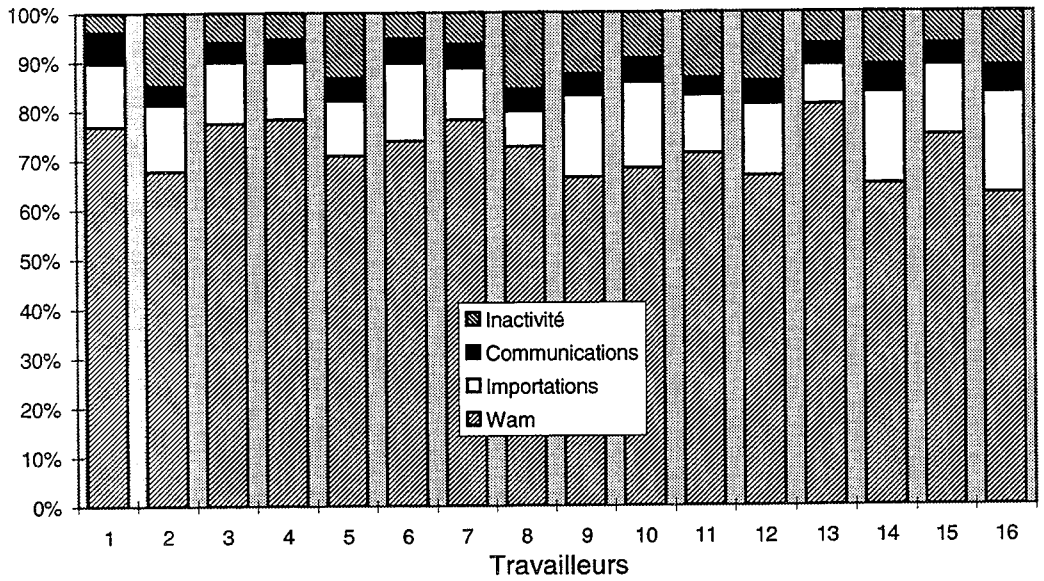


Figure 6-6 - Répartition du temps d'exécution pour le programme q1(12)

La figure Figure 6-6 donne les différents durées des phases d'un travailleur pour une exécution parallèle sur 16 travailleurs.

Ce graphique nous fait apparaître, sur chaque travailleur, un temps d'importation qui est anormalement grand. Par exemple il est de 12% du temps total d'exécution pour le premier travailleur. Or celui-ci n'a effectué que 10 importations (Figure 6-4), avec une moyenne de 832 octets par transfert (Figure 6-5), ce qui ne devrait prendre que quelques dizaines de millisecondes, soit environ cent fois moins que le temps mesuré (1s 67). De plus ce temps est, en moyenne, supérieur au temps total d'envoi de charge et d'exportation, il ne peut donc être attribué uniquement au transfert des données. Dans l'état actuel du prototype, on peut donc estimer que le temps de prise en compte d'une requête d'exportation par un travailleur est de l'ordre de la centaine de millisecondes, ce qui correspond à la durée de la période de l'échantillon.

Le graphique nous montre aussi une assez bonne répartition des temps de calcul entre les différents travailleurs du système. La régulation de la charge, si elle n'est pas encore optimale, semble relativement performante. Il faut cependant pouvoir diminuer le temps d'inactivité, ce qui permettrait de réduire encore le temps d'exécution total. En effet, le graphique des efficacités (Figure 6-7) nous montre que le temps d'inactivité devient proportionnellement très important lorsque qu'on augmente le nombre de travailleurs. Celui-ci a plusieurs causes probables, soit de mauvais paramètres pour la fonction de régulation de charge, soit la saturation de l'ordonnanceur ou éventuellement une fréquence d'échantillonnage trop faible. Seule une étude plus poussée pourra déterminer, laquelle ou lesquelles peuvent être réduites ou éliminées. Néamoins, s'il s'agît de l'ordonnanceur, ce problème sera en partie résolu par la distribution de celui-ci.

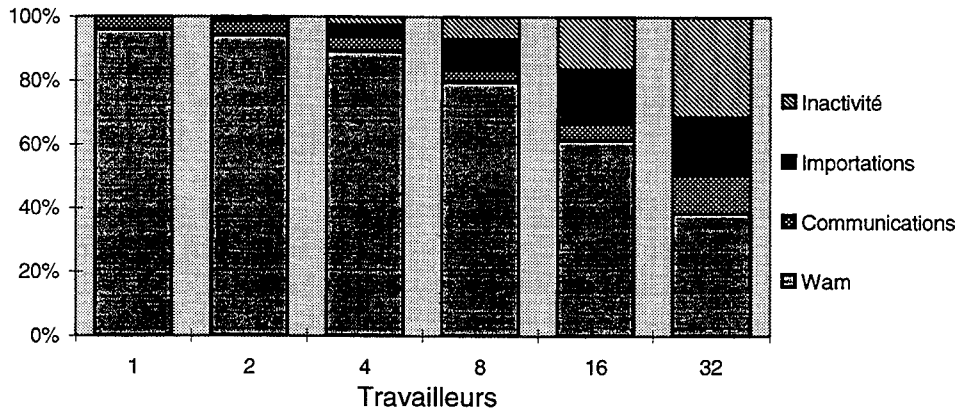


Figure 6-7 - Efficacités moyennes mesurées pour le programme q1(12)

6.3.3.4 Cohérence du système de traces

Pour pouvoir fournir un outils d'évaluation des performances qui soit utilisable, il faut assurer que celui-ci reflète bien le fonctionnement du système.

Certaines valeurs de traces, comme les nombres d'importations, d'exportations, les quantités de données transférées sont fiables. D'autres, comme les durées des différentes phases peuvent avoir une certaine marge d'erreur.
Si toutes les durées de phases ont une signification par rapport à la nature de l'exécution, le temps de calcul (temps WAM) nous semble le plus important. En effet, il représente ce que l'on nomme classiquement l'efficacité de l'exécution parallèle.
Donc, si l'on considére que l'efficacité du système est le pourcentage de temps de calcul par rapport au temps total d'exécution, les valeurs mesurées doivent se rapprocher des valeurs obtenues par la formule généralement utilisée :

$$Efficacité = \frac{Tséq\ .}{Tpar. \times Nb.\ processeurs}$$

Les valeurs obtenues par notre système de traces semblent être assez proches de la réalité. Par exemple pour l'exécution du problème des 12 reines sur 4 processeurs, l'efficacité calculée est de 93% pour 89% mesurés; et pour 32 elle est de 36% pour 39% mesurés.

6.4 Performances et effets de bords

Le prototype actuel ne dispose pas de la gestion des effets de bord, cependant de nombreux mécanismes associés sont déjà implantés et fonctionnent avec les deux versions du prototype. Les résultats présentés précédemment ont été obtenus à partir d'exécutions incluant ses différents mécanismes.

Le premier mécanisme, le point central de la gestion des effets de bord, est le maintien de l'ordre séquentiel. Celui est actif dans le prototype, est représente un coût si faible qu'il n'est pas mesurable, les variations de temps d'exécution parallèles dues à d'autres facteurs, comme l'activité du système, étant bien plus importantes. L'objectif d'obtenir une information sur l'ordre séquentiel est donc réalisé à un coût pratiquement nul.

La gestion des tampons associés aux effets de bord est déjà active pour les opérations d'entrées et de sorties. Son coût est lui aussi négligeable car cette gestion est réalisée en parallèle de la requête de travail émise vers l'ordonnanceur.
Les tampons liés aux autres effets de bord bénéficieront des mêmes conditions de gestion, et ne devraient donc pas ralentir l'exécution. La seule contrainte pour ne pas accroître le temps d'exécution est de pouvoir les gérer dans les phases d'attente forcée du travailleur. Le point le plus délicat restera le traitement des effets de bord. Ce coût supplémentaire n'est pas encore évalué. Mais à partir des mesures actuelles, on peut estimer que la validation ou l'invalidation d'une coupure pourra être effectuée en moins de deux millisecondes (SP/1, Ethernet) par groupe de point de choix. Les entrées et sortie, les prédicats collecteurs génèrerons des communications d'une durée de une à une dizaine de millisecondes.
Pour les gestion des prédicats dynamiques, l'évaluation est très difficile, et seule l'implantation permettra d'estimer les performances obtenues.

7. Conclusion et perspectives

Nous présentons maintenant un rappel des objectif du projet, ainsi que les résultats obtenu par ce travail de thèse dans ce cadre. Nous donnons ensuite les principales améliorations à apporter au système réalisé. Enfin nous ouvrons les perspectives à plus long terme du projet.

7.1 Conclusion

L'objectif du projet était de fournir un environnement de programmation Prolog avec un système d'exécution gérant automatiquement l'exécution parallèle langage, avec le respect de l'exécution séquentielle standard. Un autre objectif important concernait l'efficacité de l'exécution parallèle, enfin le dernier objectif était d'obtenir un système facilement portable et extensible.

7.1.1 Environnement de programmation

Les principales composantes de l'environnement PLoSys sont réalisées. Il reste cependant un travail important d'intégration, notamment entre la compilation des programmes et l'exécution parallèle de ceux-ci. De plus, il faut encore modifier le compilateur parallèle, qui ne prend pas encore en compte toutes les caractéristiques de l'exécution parallèle.
L'ensemble des outils de mise au point et de prise de traces sont déjà utilisables, et ont permis de d'analyser l'exécution de quelques programmes tests.

7.1.2 Exécution parallèle

Nous avons décrit une méthode originale pour gestion des effets de bord dans un système sans mémoire commune, avec le respect de la sémantique d'exécution séquentielle. Cette méthode est donnée pour un système avec un ordonnancement centralisé, mais peut être assez facilement étendue pour fonctionner avec un ordonnancement distribué.
Pour le prototype réalisé, la gestion des effets de bord n'est pas complète, et n'a pas pu être évaluée en terme de performances. Seule la partie concernant le maintien de l'ordre séquentiel a pu être implantée, ainsi que quelques mécanismes associés aux effets de bord.

7.1.3 Performances

Les performances de l'exécution séquentielle du système dépendent principalement du système Prolog utilisé, car les modifications apportées pour l'exécution parallèle ont peu d'impact. Les performances de l'exécution parallèle obtenues par les programmes de tests sont très bonnes en général. Nous avons noté que la perte importante de performance est très limité si le programme ne peut être exécuté en parallèle, et dans ce cas notre système utilise peu de ressources.
Les résultats obtenus peuvent encore être améliorés, en particulier le traitement de l'exportation. Il faut aussi fournir des fonctions de régulation de charge plus élaborée que celle que nous avons utilisé pour les premiers test du prototype.

7.1.4 Portabilité

Pour simplifier le portage de notre système nous avons utilisé une bibliothèque de communications portable et un noyau Prolog très simple. Ce choix semble avoir été correct, car la version actuelle du prototype est disponible pour trois types de machines, et est construite à l'aide du même jeu de modules de code source. Le développement du prototype a été conduit sur plusieurs types d'architectures sans poser de problème d'incompatibilité.
L'adaptation du prototype à une nouvelle architecture ne dépend que de l'adaptation de la bibliothèque de communication et du noyau Prolog, car nous n'avons utilisé aucune particularité des machines utilisées.

7.2 Perspectives

Même si la plupart des objectifs sont atteints, il reste un important travail à fournir pour obtenir un système complet. Ce travail comprend trois phases d'évolution. La première, à court terme, est l'amélioration du prototype et sa finalisation. La deuxième phase, à moyen terme, est le passage d'un fonctionnement centralisé vers un mode entièrement distribué, et l'intégration complète de l'environnement PLoSys. Enfin la dernière phase, à plus long terme, est l'extension du prototype vers le support des contraintes.

7.2.1 Amélioration du système

Le point le plus important pour l'amélioration globale de PLoSys est l'utilisation d'une bibliothèque réellement adaptée et performante pour notre type d'application. La bibliothèque actuelle impose des limitations sévères de performances et de possibilités de gestion des effets de bord car elle ne dispose pas de toutes les fonctionnalités nécessaires à l'implantation du système d'exécution parallèle de PLoSys.
En effet, l'absence de préemption ou d'interruption, alourdit considérablement le mécanisme d'échantillonnage de la charge du système, en créant une perturbation importante et très coûteuse en ressource de calcul. Cette absence rend le système lent à réagir aux événements, comme la prise en compte d'une exportation ou d'un traitement d'effet de bord.
De plus, la gestion des effets de bord n'est pas réalisable efficacement sans l'utilisation d'interruptions lors de la réception d'un message, et sans la possibilité d'envoyer des signaux entre les travailleurs. Ces absences deviennent encore plus pénalisantes si l'on veut atteindre un mode d'exécution complètement distribué.
Une prochaine évolution de la librairie devrait combler ce manque, mais il existe aussi d'autres librairies, telle que PM^2 [Namyst95], ou MPI2 [MPIF95]. Il faut noter que l'on n'utiliserait pas la possibilité de migration des processus légers offert par PM^2 avec le modèle actuel de PLoSys. Une comparaison détaillée des différentes bibliothèques et systèmes exécutifs parallèles est donnée dans [Christaller96].

Evolution vers un système entièrement distribué

Pour que l'implantation de PLoSys soit complètement distribuée il faut répartir le fonctionnement des modules de l'ordonnanceur sur chaque travailleur du système. Cela revient globalement à distribuer l'algorithme de régulation de charge et à résoudre le problème de la gestion des effets de bord en milieu distribué.

En ce qui concerne la régulation de charge, plusieurs projets ont déjà permis d'obtenir des résultats convaincants [Benjumea93], [Araujo94].
Par contre, pour la gestion des effets de bord le problème reste entier. L'ébauche de la méthode décrite dans le chapitre des effets de bord nous laisse supposer que cette étape sera réalisable à partir du moment où notre prototype fonctionnera correctement avec l'ordonnaceur centralisé.

7.2.2 Les extensions du système

7.2.2.1 Environnement de développement

Les deux axes du projets PLoSys, exécution parallèle de Prolog et évaluation des fonctions de régulation de charge, ne sont pas encore réunis au sein d'un même environnement de programmation. La première extension du projet est l'adoption d'une interface commune pour le système de développement et d'exécution parallèle, et la plate-forme d'évaluation des fonctions de régulation. Les données nécessaires à chaque partie seront alors disponibles sans traitement ou intervention manuelle de la part du programmeur.
A plus long terme, il faudra intégrer la plate-forme à l'environnement de développement, et gérer toutes les composantes du système par une interface graphique appropriée.

7.2.2.2 L'intégration d'un resolveur de contrainte

La programmation logique à travers son expression simple permet d'exprimer aisément les problèmes de recherche heuristique ou combinatoire. Seulement, la résolution de ces problème n'est pas toujours efficace, du fait que le principe du calcul est basé sur le modèle « genérer et tester » qui réduit l'espace de recherche qu'après parcours exhaustif des domaines de valeurs.
L'accélération fournie par la parallèlisation ne suffit pas toujours pour permettre des calculs dans des temps raisonnable. Le moyen de rendre efficace la résolution, est en fait de pouvoir déterminer a priori quels chemins ne conduiront pas vers une solution, et donc d'éviter des énumérations inutiles. Pour cela il faut pouvoir propager au fur et à mesure, les connaissances dont on dispose sur le domaine de recherche, afin de limiter celui-ci le plus possible. Ainsi les contraintes qui peuvent être définies sur les valeurs des variables du problème, même si elles sont faible, contribuent à une réduction importante du domaine de recherche.
Actuellement il existe de nombreuses extensions proposées pour la gestion des contraintes dans un système séquentiel. Cependant, il n'existe encore pas de système tel que celui présenté dans notre projet, qui intègre aussi un resolveur de contraintes. Cette approche nous semble pourtant très prometteuse, et pourra faire l'objet d'un travail futur à partir du système PLoSys. Ceci sera possible avec peu de modification de moteur Prolog du prototype actuel, car celui-ci est déjà utilisé pour les contraintes sur les domaines finis [Diaz95].

Bibliographie

[AitKaci90] H. Aït-Kaci, «The WAM: A (Real) Tutorial», Research Report 5, Digital Paris Research Laboratory, January 1990.

[Ali87] K.A.M. Ali, «A Method for Implementing Cut in Parallel Execution of Prolog», *Proceedings of of the 1987 Symposium on Logic Programming*, San Fransisco, pp. 449-456, September 1987.

[Ali90] K.A.M. Ali & R.Karlsson, «The MUSE Or-parallel Prolog Model and Its Performances», *In Proceedings of North-American Conference on Logic Programming 90*, Austin, pp. 757-776, 1990.

[Ali93] K.A.M. Ali & R.Karlsson, «A novel Method for Parallel Implementation of *findall*», *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Sciences n° 714, pp 235- 245, August 1993.

[Araujo94] L.Araujo and J.J. Ruz, «PDP: Prolog Distributed Processor for Independant-AND/OR Parallel Execution of Prolog», *Proceedings of International Conference on Logic Programming*, pp 142-156, MIT Press, June, 1994.

[Baetke95] F. Baetke, «The CONVEX Exemplar SPP1000 Series - MPC-Systems with a New Virtual Shared Memory Concept», *Proceedings of the second Austrian-Hungarian Workshop on Transputer Applications*, pp 14-22, Budapest, Hungary, 1994.

[Balaniuk94] A. Balaniuk and T. Muntean, «Programming with Shared Data in Parallel Loosely Coupled Machines : The Shared Memory Approach», *Proceedings of the IEEE/USP International Workshopon High Performance Computing*, p129-142, March, 1994.

[Beaumont91] A. Beaumont, S. Muthuraman, P. Szeredi and D.H.D. Warren, «Flexible scheduling of OR-parallelisme in Aurora : The Bristol Scheduler », *Parallel Architectures and Languages Europe PARLES'91*, Lecture Notes in Computer Science, vol 506, pp403-420, 1991.

[Beaumont93] T. Beaumont and D.H.D. Warren, «Schedulling Speculative Work in Or-Parallel Prolog Systems», *Proceedings of the 10th International Confernece on Logic Programming*, , pp 135-149, D.S. Warren (Ed.), 1993.

[Benjumea93] V. Benjumea and J.M. Troya, «An OR Parallel Prolog Model for Distributed Memory Systems», *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Sciences n° 714, pp 291- 301, August 1993.

[Boizumault88] P. Boizumault, «Prolog : l'implantation», Masson, Paris, juin 1988.

[Briat90] J. Briat, M. Favre et C. Geyer, «OPERA: Ou Parallélisme et régulation adaptative et Prolog», *Journées Programmation en Logique*, CNET, 1990.

[Briat91] J. Briat et *al*, «Scheduling of Or-parallel Prolog on a scalable, reconfigurable, distributed memory multiprocessor», *PARLE'91 Parallel Architectures and Languages Europe*, pp385-402, 1991.

[Briat92] J.Briat et *al*, «OPERA : OR-Parallel Prolog System on SuperNode», *Implementation of distributed Prolog*, P. Kacsuk and M.J. Wise (Editors), pp 45-63, Wilhey, Sussex, England, 1992.

[Buettner88] K.A. Buettner and J.W. Mills, «Assertive deamons», *Proceedings of the Fifth International Conferenceand Symposium on Logic Programming*, pp 1403, The MIT Press, Cambridge, Massachusetts, 1988.

[Carlsson90] M. Carlsson, «Design and Implementation of an OR-Parallel Prolog Engine», *Dissertation submitted for the Degree of Doctor of Philosophy*, Royal Institute of Technology, Stockholm, Sweden, March 1990.

[Charm92] The CHARM (3.2) Programming Language Manual, University of Illinois, Urbana Champaign, Illinois.

[Chassin94] J. Chassin de Kergommeaux et P. Codognet, «Parallel Logic Programming Systems», *ACM Computing Surveys*, Vol. 26 (3), September, 1994.

[Christaller94a] M. Christaller, «Athapascan-0a sur PVM3: définition et mode d'emploi», *Rapport Technique n°11*, LMC-IMAG, Grenoble, 1994.

[Christaller94b] M.Christaller, J.Briat et M.Rivière, «Athapascan-0 : concepts structurants simples pour une programmation parallèle efficace», *Calculateur Parallèles*, Vol. 7(2), pp173-196,1995.

[Christaller96] M. Christaller, «Athapascan-0 : vers un support exécutif pour applications parallèles irrégulières efficacement portables», Thèse, Université Joseph Fourier Grenoble 1, Grenoble, 1996.

[Clocksin88] W.F. Clocksin and H. Alshawi, «A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors», *New Generation Computing*, n° 5, pp361-376, 1988.

[Colmerauer72] A. Colmerauer, H. Kanoui, R. Pasero et P. Roussel, «Un système de Communication Homme-Machine en Français», GIA, Université d'Aix-Marseille II, 1972.

[Conery87] J.S. Conery, « Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors», *1987 Symposium on Logic Programming*, pp457-467, IEEE Computer Society Press, San Fransisco, USA, 1987.

[Costa91] V.S. Costa, D.H.D. Warren et R. Yang, «ANDORA I: A parallel Prolog system that transparency exploits both AND and OR parallellism», *ACM SIGPLAN Symposium on Principles ans Practice of Parallel Programming*, pp83-93, April, 1991.

[Costa94] V.S. Costa, G. Gupta, M. Hermenigildo and E. Pontelli, «ACE: And/Or parallelism Copying-based Execution of Logic Programs», *Proceedings of International Conference on Logic Programming*, pp 93-109, MIT Press, June, 1994.

[Covington93] M.A. Covington, «A summary of the Draft Proposed Standard», Artificila Intelligence Programs, The University of Geogia, Athens, Georgia, USA, June 1993.

[DaCosta93] C. Maciel Da Costa, «Environnement d'exécution parallèle : conception et architecture», Thèse de Université Joseph Fourier Grenoble 1, Grenoble, 1993.

[Dazy89] J.F. Dazy et J.M. Pozas, «Décompilation de clause, traitement des prédicats assert/1, retract/1, clause/2,...: pour une implantation compilée de 'tout' Prolog», *Actes du Huitième Séminaire de Programmation en Logique*, Trégastel, pp523-540, CNET, mai 1989.

[Debray90] S.K. Debray, «Towards Banishing the Cut from Prolog», *IEEE International Conference on Computer Languages*, pp 2-12, Miami, Florida, October 1986.

[Diaz93] D. Diaz ans P. Codognet, «A minimal extension of the wam for clp(fd)», *Proceedings of ICLP'93 10th International Conference on Logic Programming*, MIT Press, Budapest, Hungary, 1993.

[Diaz94] D. Diaz, «Wamcc Prolog User's Manual», INRIA, July 1994.

[Diaz95] D. Diaz, «Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système clp(FD) », Thèse de l'Université d'Orléans, janvier 1995.

[Dijkstra75] E.W. Dijkstra, «Guarded commands, non-determinacy, and formal derivation of programs», *Communication of ACM*, Vol. 18 (8), 1975.

[Escalada95] G. Escalada-Imaz and A. M. Martínez-Enríquez, «Efficient Interpretation of Logic Programs Handling Multi-Bindings», *Compulog Net Workshop on Parallelism and Implementation Technologies*, Utrecht, September 1995.

[Favre92] M. Favre, «OPERA : un système Prolog OU-Parallèle», Thèse, INPG, Grenoble, 1992.

[Forst94] A. Forst, E. Kühn, H. Pohlai and K. Schwarz, «Logic Based and Imperative Coordination Langauages», *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, ISCA, IEEE, Las Vegas, October 6-8, 1994.

[Foster89] I. Foster and S. Taylor, «Strand: New Concepts in Parallel Programming», Prentice-Hall, Englewood Cliffs, 1989.

[Foster92] I. Foster, R. Olson and S. Tuecke, «Productive Parallel Programming: The PCN Approach», *Scientific Programming*, 1992.

[Gelernter89] D. Gelernter, «Generative Communication in Linda», *ACM Transaction on Programming Languages and Systems*, vol 7(1), pp 80-112, January 1991.

[Geist93] A. Geist, A. Beguelin, A. Dongara, J.Jiang, W. Manchek, and V. Sunderam, «PVM3 user's guide and reference manual», Technical Report TM-12187, Oak Ridge National Laboratory, 1993.

[Geyer91] C. Geyer, «Une contribution à l'étude du parallélisme OU en Prolog sur des machines sans mémoire commune», Thèse de l'Université Joseph Fourier, Grenoble, Octobre 1991.

[Green90] K.J. Green and M. Hermenegildo, «&-Prolog and its performance: Exploiting independant And-parallelism», *Proceedings of the 7th International Conference on Logic Programming*, MIT Press, pp 253-268, Cambridge, Massachusetts, 1990.

[Gregory87] S. Gregory, «Parallel Logic Programming in Parlog. The Langage and its Implementation», Addison-Wesley, England, 1987.

[GuptaACH] G. Gupta, K.A.M. Ali, M. Carlsson and M.V. Hermenegildo, «Parallel Execution of Prolog Programs: A Survey», preliminary version.

[Gupta91] G. Gupta and M. Hermenegildo, «ACE: And/Or parallelism Copying-based Execution of Logic Programs», *in Lecture Notes in Computer Science 569*, Springer Verlag, 1991.

[Gupta92] G. Gupta and V. S. Costa, «Cut and Side-Effects in And-Or Parallel Prolog», extended version of «Complete and Efficient Method for Supporting Cut and Side-Effects in And-Or Parallel Prolog» *in Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, December 1992.

[Gupta93] G. Gupta and B. Jayaraman, «Analysis of Or-Parallel Execution Models», *ACM Transactions on Programming Languages and Systems*, vol 15(4), pp 659-680, September 1993.

[Gupta95] G. Gupta and E. Pontelli, «An overview of the ACE project», *Compulog Net Workshop on Parallelism and Implementation Technologies*, Utrecht, September 1995.

[Harp86] J.G. Harp, C.R. Jesshope, T. Muntean and C. Whitby-Steven, «The Development and Application of a Low Cost High Performance Multiprocessor Machine», *Processeur. ESPRIT'86 : Results and Achievements*, Elsevier Science Publishers, 1986.

[Hausman90a] B. Hausman, «Handling of speculative work in Or-parallel Prolog: evaluation results», *Logic Programming Proceedings of the 4th International Conference*, J.L. Lassez (Ed.), pp 721-735, MIT Press, 1987.

[Hausman90b] B. Hausman, «Pruning and scheduling speculative work in Or-parallel Prolog», Proceedings of PARLE 89, *Conference on Parallel Architectures ans Language Europe*, vol 2, pp 133-150, June 1989.

[Hausman90c] B. Hausman, «Pruning and speculative work in Or-parallel Prolog», Dissertation submitted for the Degree of Doctor of Technology, Royal Institute of Technology, Stockholm, Sweden, March 1990.

[Hausman90d] B. Hausman, A. Ciepielewski and A. Calderwood, «Cut and side effects in Or-parallel Prolog», *Procceedings of International Conference on Fifth Generation Computer System*, pp 831-840, November 1990.

[HPFF93] High Performance Fortran Forum, «High Performance Fortran language specification v 1.0», *Scientific Programming*, Vol. 2(1-2), 1993.

[Kacsuk92] P. Kacsuk, *Implementation of distributed Prolog*, P. Kacsuk and M.J. Wise (Editors), Wilhey, Sussex, England, 1992.

[Kacsuk92a] P. Kacsuk, «3DPAM = WAM+DATAFLOW», *Proceedings of the Post-Confernece Joint Workshop on Distributed and Parallel Implementation*, pp 32-53, Washingtown, November 1992.

[Kacsuk92b] P. Kacsuk, «Distributed Data Driven Prolog Abstract Machine», *Implementation of distributed Prolog*, P. Kacsuk and M.J. Wise (Editors), pp 45-63, Wilhey, Sussex, England, 1992.

[Kacsuk93] P. Kacsuk, «Cut Implementation in a Massively Parallel Prolog System», *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pp 96-103, Gran-Canaria, 1993.

[Kacsuk94] P. Kacsuk, «Wavefront Scheduling in LOGFLOW», *2nd Euromicro Workshop on Parallel and Distributed Processing*, pp 71-80, Malaga, 1994.

[Kale89] L.V. Kale, B. Ramkumar and W. Shu, «Parallel Prolog on the Intel iPSC/2», *The 4th Conference on Hypercubes, concurrent computers, and applications*, vol 1, pp 525-536, March 1989.

[Kannat94] S.E. Kannat, E. Morel, J.P. Kitajima, and J. Briat, «A Plateform to Study Dynamic Load Balancing Functions for Parallel Logic System», *IEEE Proceedings of the First International Workshop on Parallel Processing*, pp 580-585, Bengalore, India, December 26-31, 1994.

[Kannat96] S.E. Kannat, «Une contribution à l'étude de la régulation de charge pour les systèmes logiques parallèles», Thèse, INPG, Grenoble, 1996.

[Karlsson92] R. Karlsson, «A High Performance OR-Parallel Prolog System», *Dissertation submitted for the Degree of Doctor of Technology*, Royal Institute of Technology, Stockholm, Sweden, March 1992.

[Kowalski74] R. Kowalski, «Predicate Logic as a Programming Langage», Information Processing 74, IFIP Congress, pp569-574, 1974.

[Kuhn93] E. Kühn, H.Pohlai and F. Puntigam, «Concurrency and Backtracking in V.P.L.», *Computer Langages*, Vol. 19 (3), July 1993.

[Lindoholm87] T.G. Lindoholm and R. O'Keefe, «Efficient implementation of a defensible semantics for dynamic Prolog code», *Proceedings of the 4th International Conference on Logic Programming Languages*, pp 21, MIT Press, Cambridge, Massashusetts, 1987.

[Lusk92] E. Lusk, S. Mudambi, R. Overbeek and P. Szeredi, «Application of the Aurora Prolog system to computational molecular biology», Proceedings of the JICSLP'92 Post Conference Workshop on Distributed and Parallel Implementation of Logic Programming Systsems, Washington DC, USA, 1992.

[Masuzawa86] H. Masuzawa, «Kabu Wake parallel inference mechanism and its evaluation», *1986 FJCC*, IEEE, pp 858-876, New York, 1986.

[Morel96] E.Morel, J.Chassin de Kergommeaux, J.Briat, «PLoSys : OU parallélisme et effets de bord sur système parallèle», *Cinquièmes Journée Francophones de Programmation Logique et programmation par Contraites*, pp 131-145 , Ed. HERMES, Clermont-Ferrand, France, 5-7 juin, 1996.

[MPI93] P.S. Pachero, «A User's guide to MPI», University of San Fransisco, March 1993.

[MPIF95] Message Passing Interface Forum, «MPI-2 : Extension to the message passing interface standard», *Technical report*, University of Tenessee, Knoxville, Tenessee, 1995.

[Namyst95] R. Namyst at J.F. Méhaut, «PM^2 parallel multithreaded machine : A multithreaded environment on top of PVM», *Proceedings of Second European PVM User's Group Meeting*, pp 179-184, Lyon, France, 1995.

[OKeefe90] R.A. O'Keefe, «The craft of Prolog», MIT Press, Cambridge Massachusetts, 1990.

[Quiniou90] R. Quiniou et L. Trilling, «Les prédicats collectifs un moyen d'expression du contrôle du parallélisme», *Rapport Interne n° 548*, IRISA, Septembre 1990.

[Ramkumar94a] B. Ramkumar and L. V. Kalé, «Machine Independant AND and OR Parallel Execution of Logic Programs: part 1 - The Binding Environnement», *IEEE Transaction on Parallel and Distributed Systems*, vol 5(2), pp 170-180, February 1994.

[Ramkumar94b] B. Ramkumar and L. V. Kalé, «Machine Independant AND and OR Parallel Execution of Logic Programs: part 2 - Compiled Execution», *IEEE Transaction on Parallel and Distributed Systems*, vol 5(2), pp 181-193, February 1994.

[Seznec95] A. Seznec et Y. Mével, «Evolution des gamme de processeurs MIPS,DEC Alpha, PowerPC, SPARC et xxx86», *Publication Interne n° 957*, IRISA, Décembre 1995.

[Singhal89] A. Singhal and Y. Patt, «Unification parallelism: How much can it be exploited?», *Proceeding of the North American Conference on Logic Programming*, pp 1135-1148, June 1993.

[Shapiro83] E. Shapiro, «A subset of Concurrent Prolog and its interpreter», *Technical report*, Weizmann Institue of Science. 1983.

[Shapiro86] E. Shapiro and L. Sterling, «The art of Prolog», MIT Press, Cambridge Massachusetts, 1986.

[Smith92] Donald A. Smith, «MultiLog: Data Or-Parallel Logic Programming», *Proceedings of the Post-Confernece Joint Workshop on Distributed and Parallel Implementation*, Washingtown, pp 159-175, November 1992.

[Smith93] Donald A. Smith, «MultiLog: Data Or-Parallel Logic Programming», *Proceedings of the 10th International Confernece on Logic Programming*, D.S. Warren (Ed.) , pp 314-331, 1993.

[Stevens90] W.R. Stevens, «Unix network programming», Prentice Hall, 1990.

[Sutcliffe90] G. Sutcliffe and J. Pinakis, «Prolog-Linda - An embedding of Linda in muProlog», *Proceedings of AI'90 - the 4th Australian Conference on Artificial Intelligence*, pp 331-340, Perth (Australia), 1990.

[Tarau91] P. Tarau, «A Compiler and a Simplified Abstract Machine for The Execution of Binary Metaprograms», *Proceedings of the Logic Programming Conference '91*, pp 119-128, Tokyo, September, 1991

[Tarau95] P. Tarau and B. Demoen, «Higher-Order Programming in an OR-intensive Style», *Compulog Net Workshop on Parallelism and Implementation Technologies*, Utrecht, September 1995.

[Tick91] E. Tick, «Parallel Logic Programming», TheMIT Press, Cambridge, Massachusetts, England, 1991.

[VanRoy93] Peter Van Roy, «1983-1993: The Wonder Years of Sequential Prolog Implementation», *Research Report 36*, Digital Paris Research Laboratory, December 1993.

[Yang93] R. Yang, T. Beaumont, I. Dutre, V. S. Costa and D.H.D. Warren, «Performance of the Compiler-based Andorra-I System», *Proceedings of the 10th International Confernece on Logic Programming*, D.S. Warren (Ed.) , pp 150-166, 1993.

[Yasuhara84] H. Yasuhara & K. Nitadori, « Orbit: A Parallel computing Model of Prolog », *New Generation Computing*, vol 2, pp. 277-288, 1984.

Annexe

Programme : ham3.pl

```
/*  Hamiltonian Graphs...   from ECRC benchmark   */

mham_top :-
    cycle_ham([s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12],X), write(X), nl,
    fail.

cycle_ham([X|Y],[X,T|L]):-
        chain_ham([X|Y],[],[T|L]),
        edge(T,X).

chain_ham([X],L,[X|L]).
chain_ham([X|Y],K,L):-
        delete(Z,Y,T),
        edge(X,Z),
        chain_ham([Z|T],[X|K],L).

delete(X,[X|Y],Y).
delete(X,[U|Y],[U|Z]):-
        delete(X,Y,Z).

edge(X,Y):-
        connect(X,L),
        el(Y,L).

el(X,[X|L]).
el(X,[Y|L]):-
      el(X,L).

connect(s0,[s1,s2,s3,s5,s7,s10]).
connect(s1,[s0,s3,s4,s5,s6,s7,s9]).
connect(s2,[s0,s1,s2,s3,s4,s5,s6,s8]).
connect(s3,[s0,s1,s2,s4,s6,s7,s8,s9]).
connect(s4,[s1,s2,s3,s5,s6,s7,s8,s9]).
connect(s5,[s0,s1,s2,s4,s7,s8,s9,s11]).
connect(s6,[s1,s2,s3,s4,s7,s8,s9,s12]).
connect(s7,[s0,s1,s3,s4,s5,s6,s11]).
connect(s8,[s2,s3,s4,s5,s6,s9,s12]).
connect(s9,[s1,s3,s4,s5,s6,s8,s10]).
connect(s10,[s0,s9,s11,s12]).
connect(s11,[s5,s7,s10,s12]).
connect(s12,[s6,s8,s10,s11]).
```

Programme : q1_n.pl

```
/* Queens : générer et tester incrémental */

queens(N,Qs) :-
      range(1,N,Ns),
      queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
      sel(UnplacedQs,UnplacedQs1,Q),
      not_attack(SafeQs,Q),
      queens(UnplacedQs1,[Q|SafeQs],Qs).

not_attack(Xs,X) :-
      not_attack(Xs,X,1).

not_attack([],_,_).
not_attack([Y|Ys],X,N):-
      X =\= Y+N,
```

```
        X =\= Y-N,
        N1 is N+1,
        not_attack(Ys,X,N1).

sel([X|Xs],Xs,X).
sel([Y|Ys],[Y|Zs],X):-
        sel(Ys,Zs,X).

range(N,N,[N]).
range(M,N,[M|Ns]):-
        M < N,
        M1 is M+1,
        range(M1,N,Ns).

go(N) :- queens(N,L), write(L), nl, fail.
```

Programme : k1_5.pl

```
/* Parcours naïf du cavalier sur un échiquier de 5x5 */

portee(X,Y) :- Y is X-21.
portee(X,Y) :- Y is X-12.
portee(X,Y) :- Y is X-19.
portee(X,Y) :- Y is X-8.
portee(X,Y) :- Y is X+8.
portee(X,Y) :- Y is X+12.
portee(X,Y) :- Y is X+19.
portee(X,Y) :- Y is X+21.


ote(L,X,LS):-
        X<56, X>10,
        oter(L,X,LS).

oter([],_,_):-
        fail.
oter([X|L],X,L).
oter([Y|L],X,[Y|LS]):-
        X =\= Y,
        oter(L,X,LS).

coup([],CC,[CC|[]]).
coup(LRes,CC,[CC|LCou]):-
        portee(CC,Z),
        ote(LRes,Z,LResS),
        coup(LResS,Z,LCou).

caval(Start,L):-
        oter([ 11,12,13,14,15,
               21,22,23,24,25,
               31,32,33,34,35,
               41,42,43,44,45,
               51,52,53,54,55],
               Start,LC),
        coup(LC,Start,L).

go(Start) :- caval(Start,L), write(L), nl, fail.

:- go(11).
```

Programme : k3_5.pl

```
/* Parcours optimisé du cavalier sur un échiquier de 5x5 */

portee(X,Y) :- Y is X-21.
portee(X,Y) :- Y is X-12.
portee(X,Y) :- Y is X-19.
portee(X,Y) :- Y is X-8.
portee(X,Y) :- Y is X+8.
portee(X,Y) :- Y is X+12.
portee(X,Y) :- Y is X+19.
portee(X,Y) :- Y is X+21.


ote(L,X,LS):-
      X<56, X>10,       oter(L,X,LS).

oter([],_,_):-
      fail.
oter([X|L],X,L).
oter([Y|L],X,[Y|LS]):-
      X \== Y,
      oter(L,X,LS).

coup(blanche,_,[],CC,[CC]).
coup(noire  ,[],_,CC,[CC]).
coup(blanche,LB,LN,CC,[CC|LCou]):-
      oter(LN,Z,LNS),
      portee(CC,Z),
      coup(noire,LB,LNS,Z,LCou).
coup(noire,LB,LN,CC,[CC|LCou]):-
      oter(LB,Z,LBS),
      portee(CC,Z),
      coup(blanche,LBS,LN,Z,LCou).

caval(Color,Start,L):-
      oter([11,13,15,22,24,31,33,35,42,44,51,53,55],Start,LB),
      coup(Color,LB,[12,14,21,23,25,32,34,41,43,45,52,54],Start,L).

go :- caval(blanche,11,L), write(L), nl, fail.
```

Programme : q2_9.pl

```
/* Queens : générer et tester simple */

:- main.

go :- q([1,2,3,4,5,6,7,8,9],_), fail.

q(L,C):-
      perm(L,P),
      pair(L,P,C),
      safe([],C).

perm([],[]).
perm(Xs,[Z|Zs]):-
      sel(Z,Xs,Ys),
      perm(Ys,Zs).

sel(X,[X|Xs],Xs).
sel(X,[Y|Ys],[Y|Zs]):-
      sel(X,Ys,Zs).

pair([],[],[]).
pair([X|Y],[U|V],[p(X,U)|W]):-
      pair(Y,V,W).

safe(X,[]).
safe(X,[Q|R]):-
      test(X,Q),
      safe([Q|X],R).

test([],X).
test([R|S],Q):-
      test(S,Q),
      nd(R,Q).

nd(p(C1,R1),p(C2,R2)):-
      C  is C1-C2,
      R  is R1-R2, C=\=R,
      NR is R2-R1, C=\=NR.
```

Résumé

Cette thèse présente l'étude de l'implantation d'un système Prolog parallèle sur une architecture sans mémoire commune dans le cadre du projet PLoSys (Parallel Logic System). L'exécution exploite le parallélisme de manière implicite. Le système repose sur un modèle OU multiséquentiel. Le partage de l'état d'exécution est assuré par copie des données. Le langage Prolog supporté est complet, et intègre les effets de bord classiques du langage. La gestion parallèle fait l'objet d'une étude complète pour préserver la compatibilité avec l'exécution séquentielle du langage Prolog. En particulier, une méthode originale est présentée pour la gestion parallèle des effets de bord.
Enfin, ce document présente la réalisation d'un prototype portable, ainsi que l'analyse des résultats obtenus.

Abstract

This thesis discribes the implementation of a parallel Prolog system on distributed memory architecture. This work is a part of the PLoSys (Parallel Logic System) project.
Parallelisation of a program is done implicitly using. The program execution uses an OR-parallel multisequential model. Data copying is used for sharing execution state among processors.
The supported language represent all Prolog predicates, including common side-effects. An original method is described to implement the parallel management of side-effects which preserves the sequential semantic of a Prolog program execution.
Finaly, this document presented a portable prototype with its evaluation, with the analysis of the first results obtained.