Régulation dynamique de charge dans les systèmes logiques parallèles

Salah Eddine Kannat

▶ To cite this version:

Salah Eddine Kannat. Régulation dynamique de charge dans les systèmes logiques parallèles. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1996. Français. NNT : . tel-00004997

HAL Id: tel-00004997
https://theses.hal.science/tel-00004997

Submitted on 23 Feb 2004

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

Salah Eddine KANNAT

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(*arrêté ministériel du 30 mars 1992*)

Spécialité : Informatique

Régulation Dynamique de Charge dans les Systèmes Logiques Parallèles

Date de soutenance : 4 novembre 1996

Composition du jury

Président :	Brigitte PLATEAU
Rapporteurs :	Bernard TOURSEL
	Michel TREHEL
Examinateurs :	Jacques BRIAT
	Gilles BERGER-SABBATEL
Directeur de Thèse :	Jacques BRIAT

Thèse préparée au sein du
Laboratoire Systèmes et Réseaux LSR-IMAG
et du
Laboratoire de Modélisation et Calcul LMC-IMAG
(Institut de Mathématiques Appliquées de Grenoble)

Table des matières

I Introduction **11**
- I.1 Domaine de recherche : Motivations . . . 11
  - I.1.1 Quel langage de programmation ? . . . 12
  - I.1.2 Quelle type de machine parallèle ? . . . 14
  - I.1.3 Programmation logique parallèle . . . 15
- I.2 Problématique . . . 16
- I.3 Cadre de travail : Projet PLoSys . . . 17
- I.4 Travail de thèse . . . 18
- I.5 Structure du document . . . 19

II PROgrammation LOGique : Principe et Implantation **21**
- II.1 Introduction . . . 21
- II.2 Présentation du langage . . . 21
- II.3 Bases théoriques . . . 23
  - II.3.1 Clauses de Horn . . . 24
  - II.3.2 L'unification . . . 25
  - II.3.3 Résolution et interprétation . . . 26
  - II.3.4 Stratégie de résolution . . . 27
- II.4 Effets de bord . . . 29
- II.5 Implantation de Prolog : la WAM . . . 30
  - II.5.1 Organisation mémoire . . . 31
  - II.5.2 Jeu d'instructions . . . 33
- II.6 Efficacité séquentielle . . . 34
- II.7 Conclusion . . . 34

III Systèmes logiques parallèles **37**
- III.1 Introduction . . . 37
- III.2 Parallélisme en programmation logique . . . 38
  - III.2.1 Les sources de parallélisme . . . 38
    - a) Le parallélisme OU . . . 38
    - b) Le parallélisme ET . . . 39

c) Le parallélisme d'unification 40
III.2.2 Types d'exploitation 40
a) Parallélisme de données 41
b) Parallélisme de contrôle 43
III.2.3 Parallélisme explicite 43
III.2.4 Parallélisme implicite 45
III.3 Les modèles multi-séquentiels 47
III.3.1 Le parallélisme OU 47
III.3.2 Le parallélisme ET 48
a) Parallélisme ET-Indépendant 48
b) Parallélisme ET-Dépendant 50
III.3.3 Parallélisme ET-OU combiné 52
III.3.4 Exploitation des effets de bord 52
III.4 Le problème d'efficacité 54
III.4.1 Notion de tâche et de Granularité 54
III.4.2 Régulation de charge 56
a) Gestion mémoire 58
b) Contrôle du grain de parallélisme 59
c) Impact des effets de bord 60
III.5 Conclusions 61

IV Régulation de charge dans les systèmes OU multi-séquentiels 63
IV.1 Introduction 63
IV.2 Régulation de charge 64
IV.2.1 Composants d'une fonction de régulation de charge 65
IV.2.2 Architecture d'une fonction de régulation 68
a) Gestion de l'état de charge du système 68
b) Transfert de charge 69
c) Stratégie de régulation 70
IV.3 Régulation de charge dans les systèmes OU Multi-séquentiels . . . 71
IV.3.1 Formulation du problème de régulation 73
IV.3.2 Conception et mise en œuvre : problèmes & solutions . . . 75
a) Stratégies de régulation 75
b) Efficacité d'une fonction de régulation 78
IV.4 Transfert de la charge de calcul 79
IV.4.1 Extraction du contexte d'exécution 79
IV.4.2 Coûts d'installation 81
IV.5 Evaluation de la charge du système 83
IV.5.1 Estimation des coûts de calcul 83
IV.5.2 Estimation de la charge 85
IV.6 Contrôle du grain 87

IV.6.1 Transformation du programme 89
IV.6.2 Approches heuristiques 90
IV.7 Quelques cas de fonctions de régulation 92
IV.7.1 Régulateurs de charge du système *Aurora* 92
IV.7.2 Régulateur de charge du système *Muse* 94
IV.7.3 Régulateur de charge du système *Opera* 95
IV.8 Conclusion 95

V *PLoSys* : Modélisation et Plate-forme d'évaluation 97
V.1 Introduction 97
V.2 Régulation de charge dans *PLoSys* 98
V.2.1 Le modèle *PLoSys* 98
a) Le travailleur 98
b) Le régulateur 99
c) Le schéma de coopération 100
V.2.2 Evaluation de la charge du système 100
a) Evaluation de la charge d'un *travailleur* 100
b) Maintien de l'état de charge du système 101
V.2.3 Mécanisme de transfert de la charge 102
V.2.4 Stratégies de régulation de charge 103
V.3 Incidence des paramètres de régulation 104
V.3.1 La fréquence d'échantillonnage 104
V.3.2 Contrôle de la granularité 105
V.3.3 Seuillage de l'indice de charge 105
V.3.4 Bilan 108
V.4 Modéliser pour évaluer 109
V.5 Technique de modélisation 110
V.5.1 Principe de modélisation 110
V.5.2 Extraction du graphe de tâches 111
V.5.3 Avantages et limites du modèle 112
V.6 Environnement d'évaluation 113
V.6.1 Description de l'architecture matérielle 113
a) Les ressources de calcul : *le Transputer* 115
b) Réseau d'interconnexion 116
V.6.2 Organisation de la plate-forme 117
a) Le travailleur 117
b) Le gestionnaire de communication 118
c) Le régulateur 118
V.6.3 Schéma d'une exécution parallèle 119
V.7 Collecte d'informations de mesures 121
V.8 Interface utilisateur 122

V.9 Conclusion 123

VI Contexte expérimental : Premiers résultats **125**

VI.1 Introduction 125

VI.2 Indices de performances 126

VI.2.1 Accélération et efficacité 127

VI.2.2 Facteur de distribution 128

VI.3 Programmes de test 129

VI.3.1 Extraction du graphe de tâches 130

VI.3.2 Calibrage des programmes de test 130

VI.4 Evaluation de stratégies de régulation 131

VI.4.1 Gestion de l'état de charge du système 131

a) Evaluation de la charge d'un processeur 132

b) Maintien de l'état de charge du système 132

VI.4.2 Algorithme de régulation 134

a) Seuillage de l'indice de charge 135

b) Contrôle de la granularité 136

c) Stratégie de localisation 138

VI.5 Conclusion : Bilan 140

VII Conclusions et Perspectives **143**

VII.1 Conclusions sur le travail effectué 143

VII.2 Perspectives : Problèmes ouverts 147

A Programmes Prolog d'essai **149**

A.1 Programme du 4-gasp 149

A.2 Programme des 8-reines 150

B Graphes de tâches **153**

C Interface Utilisateur : *PVT* **157**

Liste des Figures

II.1 *Arbre ET/OU : Stratégie de parcours.* 28
II.2 *Organisation mémoire de la WAM.* 32
II.3 *Instructions WAM pour p(X,Y) :- q(X,Z),r(Z,Y).* 33

III.1 *Exemple de programme.* 38
III.2 *Sources de parallélisme en Prolog.* 39
III.3 *OU-parallélisme de données.* 41
III.4 *ET-parallélisme de données.* 42
III.5 *Parallélisme explicite.* 44
III.6 *OU-parallélisme.* 47
III.7 *Parallélisme ET-Indépendant.* 49
III.8 *Parallélisme de flot.* 51
III.9 *Travail spéculatif.* 53

IV.1 *Régulation de charge.* 64
IV.2 *Eléments d'une fonction de régulation.* 66
IV.3 *Eléments d'une stratégie de régulation.* 67
IV.4 *Stratégie multi-séquentielle.* 72
IV.5 *Conditions minimales de parallélisation efficace.* 74
IV.6 *Sections publiques et privées.* 76
IV.7 *Stratégies de sélection des points de choix.* 77
IV.8 *Technique de recopie de contexte.* 81
IV.9 *Technique de recalcul fondée sur les oracles.* 82
IV.10 *Analyse de complexité.* 85
IV.11 *Heuristique d'estimation de charge.* 87
IV.12 *Contrôle du grain.* 88
IV.13 *Regroupement des alternatives.* 91

V.1 *Arbre équilibré complet.* 106
V.2 *Arbre dégénéré.* 106
V.3 *Extraction du graphe de tâches.* 111
V.4 *Architecture de l'environnement d'évaluation.* 114

V.5 *Architecture du module de base d'un Supernode.* 115
V.6 *Topologie du réseau de processeurs.* 116
V.7 *Organisation de la plate-forme.* 117
V.8 *Transitions d'états.* 120
V.9 *Schéma de coopération parallèle.* 121
V.10 *Caractéristiques d'une exécution* 122
V.11 *Interface utilisateur* 123

VI.1 *Problème des 8 reines.* 129
VI.2 *Problème du gasp.* 130
VI.3 *Transitions d'état d'un processeur.* 132
VI.4 *4-gasp : Impact de la fréquence d'échantillonnage.* 134
VI.5 *8-reines : Impact de la fréquence d'échantillonnage.* 135
VI.6 *4-gasp* & *8-reines : Impact du seuillage.* 136
VI.7 *Impact de la granularité sur l'accélération.* 137
VI.8 *Impact de la granularité sur le facteur* $\mathcal{D}_f$. 138
VI.9 *4-gasp :Impact de la topologie sur* S_N *et* $\mathcal{D}_f$. 139
VI.10 *8-reines :Impact de la topologie sur* S_N *et* $\mathcal{D}_f$. 140

B.1 *Graphe de tâches du programme des 8-reines.* 153

C.1 *Paramètres de régulation* 157
C.2 *Paramètres de configuration* 158

Liste des Tables

IV.1 *Stratégies de régulation de charge dans les systèmes* OU *parallèles.* 92
IV.2 *Performances d'Aurora (8-reines) sur Sequent Symmetry.* 93
IV.3 *Performances de Muse et Opera pour le problème des 8-reines.* . . . 95

V.1 *X-reines : Nombre de points de choix Vs. Temps d'exécution.* . . . 101
V.2 *X-gasp : Nombre de points de choix Vs. Temps d'exécution.* 101

VI.1 *Caractéristiques des programmes d'essai.* 130
VI.2 *Fréquences d'évaluation de la charge d'un processeur.* 133

B.1 *Informations de trace du programme 4-gasp.* 154
B.2 *Caractéristiques des programmes d'essai.* 155

Chapitre I

Introduction

Historiquement, on a recouru à deux méthodes pour augmenter les performances des applications informatiques : (a) utiliser des processeurs plus rapides (b) paralléliser l'application sur plusieurs processeurs. La pénétration croissante de l'informatique dans les secteurs d'applications de plus en plus complexes comme ceux du "Grand Challenge" [Nor96] (météorologie, aéronautique, génome humain, physique des particules) nécessite l'utilisation conjointe de ces deux approches pour faire face à une demande sans cesse réitérée : traiter plus vite des problèmes plus grands.

I.1 Domaine de recherche : Motivations

Paradoxalement aujourd'hui, le problème n'est plus seulement de disposer d'une puissance de calcul importante mais aussi d'avoir des environnements de développement et des outils d'exploitation permettant, d'une part, d'utiliser efficacement toute la puissance de calcul offerte par les architectures parallèles et facilitant, d'autre part, le portage des applications initialement écrites pour des machines séquentielles.

L'exploitation des machines parallèles est une activité complexe et coûteuse ce qui restreint aujourd'hui encore leur utilisation à des secteurs spécifiques. Ceci s'explique en partie par le fait que la plupart des langages de programmation *impératifs* existants sont caractérisés par une sémantique fortement couplée à l'ordre d'apparition des instructions (constructions) dans le langage. De tels langages ne permettent pas à l'utilisateur d'avoir une vue uniforme indépendamment de l'architecture de la machine cible (séquentielle ou parallèle).

L'approche que nous pensons prometteuse et que nous prônons dans cette thèse consiste à utiliser des langages de programmation dits *"de plus haut niveau"* qui permettent à l'utilisateur de s'abstraire de l'aspect contrôle du parallélisme. Le programmeur doit uniquement spécifier les fonctionnalités de son application

et non l'exécution de celle-ci sur une architecture particulière. C'est l'exécutif du langage qui assure l'enchaînement des actions nécessaires à l'exécution du programme. Là où le programmeur/compilateur devait préciser en détail toutes les allocations de ressources (processeur, mémoire) dans un langage impératif, c'est maintenant à l'exécutif du programme de gérer dynamiquement ces ressources.

I.1.1 Quel langage de programmation ?

Actuellement, il n'existe pas de langage parallèle ni d'environnement de programmation communément acceptés. Tous les constructeurs proposent une programmation fondée sur un langage conventionnel comme C ou Fortran, enrichi de constructeurs syntaxiques pour exprimer le parallélisme. La programmation d'applications avec ce type de langage nécessite généralement l'intervention d'un spécialiste de la programmation parallèle. Il n'existe pas non plus de consensus sur les modèles de programmation parallèle. Un grand nombre de modèles de programmation fondés sur l'échange de messages et/ou sur le concept de mémoire virtuelle partagée ont été développés ces dernières années mais aucun standard susceptible de satisfaire les besoins de toutes les applications ne s'est jusqu'à présent dégagé.

Aussi, les travaux de recherche actuels tentent-ils de combler ces lacunes grâce au développement d'environnements de programmation permettant aux utilisateurs une mise en œuvre plus aisée de leurs applications et une meilleure exploitation des possibilités offertes par ces machines. Cet objectif nécessite en particulier d'offrir aux utilisateurs des langages de programmation qui permettent de décrire de manière simple leurs applications et de faciliter le portage des applications existantes de manière transparente.

L'un des points clés qui permet de distinguer les différentes approches dans ce domaine est l'expression du parallélisme dans le modèle de programmation. Les travaux ont donné naissance à deux courants de pensée :

- l'expression **explicite** du parallélisme dans le langage,
- l'extraction du parallélisme **implicite** du programme.

Le parallélisme explicite consiste à fournir au programmeur des mécanismes de base pour qu'il exprime et gère explicitement le parallélisme dans son programme. Il s'agit alors de concevoir un nouveau langage ou d'étendre un langage de programmation classique au moyen de constructeurs parallèles, avec l'introduction de notions de processus, de communication et de synchronisation dans le langage, ou dans son environnement d'exécution. L'utilisation à bon escient du parallélisme dans cette approche reste intégralement à la charge du programmeur.

La parallélisation implicite, par contre, vise l'exploitation de la puissance des machines parallèles en masquant autant que possible la complexité des problèmes de gestion du parallélisme. Cette voie privilégie la productivité et cherche à décharger le programmeur des tâches de bas niveau, en extrayant le parallélisme inhérent de l'application de façon transparente pour celui-ci. Dans cette optique, deux approches sont distinguées.

La première consiste à utiliser des compilateurs capables d'extraire, des programmes séquentiels, les parties pouvant être exécutées en parallèle. Les recherches dans cette voie portent essentiellement sur les techniques de compilation permettant de déceler le parallélisme existant dans le programme au niveau syntaxique. La parallélisation se fait de façon statique, au moment de la compilation, par analyse de la dépendance des données et par la restructuration du code. Ainsi, les problèmes de la parallélisation automatique de Fortran traitent principalement de la parallélisation de boucles.

La deuxième approche consiste à utiliser des langages de programmation **déclaratifs**. Ces langages détiennent une propriété intéressante qui les différencie des langages impératifs, par la séparation de la sémantique du programme du contrôle d'exécution de ce programme. La sémantique du programme décrit la résolution d'un problème donné, indépendamment de la stratégie de contrôle d'exécution utilisée pour son évaluation. La description d'un problème fait souvent apparaître des étapes de résolution pouvant être exécutées dans un ordre quelconque, sans affecter la signification du programme. Ces étapes en particulier peuvent donc être exploitées par un évaluateur parallèle, dans le but d'améliorer les performances d'exécution de l'application.

On connaît deux grandes classes de langages déclaratifs : les langages **fonctionnels** et les langages **logiques**. Les langages fonctionnels ont évolué plus tôt que les langages logiques et l'exécution parallèle des langages fonctionnels a fait l'objet de vastes travaux de recherche dès la fin des années 70 et jusqu'à nos jours. En revanche, les travaux en programmation logique ne se sont multipliés et n'ont acqui un intérêt considérable qu'avec le projet de machines dites “de cinquième génération” lancé au Japon par l'ICOT [Kur88].

Aujourd'hui, la programmation logique et plus particulièrement le langage Prolog sont utilisés dans plusieurs domaines tels que l'intelligence artificielle (IA) [Nil90, Sho94], pour le traitement des langages naturels [GM96], le séquencement génétique [Sea75, Sea92] et le développement de systèmes experts [SH96], ainsi qu'en recherche opérationnelle (RO) [RK90, Sze91a] pour l'expression de gros problèmes combinatoires, ou encore la mise en œuvre de systèmes de gestion de bases de données réparties, et le prototypage rapide d'applications [Abb90, BW90, Els90, Fil94, SJ96].

Le besoin croissant de puissance dans ces différents domaines a motivé le développement de techniques d'évaluation efficaces pour Prolog. Cette recherche

d'efficacité se fait en amont, par le développement de techniques de compilation et d'optimisation, ainsi qu'en aval par le développement de mécanismes d'évaluation parallèle.

Les techniques de parallélisation efficace du langage Prolog restent à ce jour un important domaine de recherche. C'est l'objectif de notre travail, comme celui de beaucoup d'autres équipes de recherche.

I.1.2 Quelle type de machine parallèle ?

La recherche d'un support efficace de calcul parallèle a donné lieu à une vaste gamme d'architectures matérielles, allant des systèmes fortement couplés comme les machines multiprocesseurs à mémoire commune, jusqu'aux architectures faiblement couplées composées de machines autonomes (réseau de station de travail), éventuellement hétérogènes, interconnectées par un réseau de communication. On répartit ces architectures en plusieurs classes suivant le moyen de communication utilisé pour l'échange de données (mémoire partagée ou échange de messages) et suivant la gestion des flux de données [Fly79] :

- synchrone pour les architectures SIMD *(Single Instruction Multiple Data)*
- asynchrone pour les architectures MIMD *(Multiple Instruction Multiple Data)*

La mise en œuvre d'un système parallèle ne peut pas être abordée de la même façon sur une machine parallèle disposant d'un modèle de fonctionnement SIMD que sur une machine possédant un modèle de fonctionnement MIMD. Dans le premier cas, seule une répartition des données a un sens, étant donné que le code est séquencé par un contrôleur unique. Dans le second cas, on peut choisir de répartir les données et les traitements à effectuer sur ces données. Chaque unité exécute des instructions différentes sur des données différentes. Il n'y a plus de contrôle unique. Le support de communication utilisé pour l'échange de données entre les processeurs permet de distinguer les machines MIMD à mémoire commune et celles à mémoire distribuée. L'espace mémoire adressable dans les machines à mémoire commune est le même pour chaque processeur et il porte sur toute la mémoire partagée. Le seul mode de communication est par mémoire partagée. On connaît deux types d'architectures de machines à mémoire commune : UMA et NUMA. Les machines UMA *(Uniform Memory Architecture)* préservent un temps uniforme d'accès à la mémoire. Dans les machines NUMA *(Non Uniform Memory Acces)*, les accès à la mémoire sont plus ou moins rapides selon que la mémoire adressée est locale ou non au processeur.

Dans les machines MIMD sans mémoire commune (NORMA : *No Remote Memory Acces*), l'espace mémoire adressable est local à chaque processeur. Les

communications entre processeurs se font par échanges de messages via un réseau d'interconnexions spécifique.

Les machines de type UMA/NUMA permettent une programmation parallèle confortable. Cependant, elles souffrent de la limitation du degré de parallélisme induit par le goulot d'accès à la mémoire. L'utilisation de caches locaux réduit l'importance de ce goulot mais pose le problème du maintien de leur cohérence.

Par contre, les machines NORMA présentent un intérêt particulier du fait qu'elles possèdent un grand potentiel d'extensibilité d'une part et bénéficient d'un bon compromis coût/performance d'autre part [Bed94]. En effet, la notion de machine NORMA est générique. Elle inclut aussi bien un ensemble de stations de travail connectées par un réseau local qu'une machine composée de microprocesseurs interconnectés par un réseau de communications spécifique.

Dans la classe d'architectures qui nous intéresse, les processeurs sont connectés par des liens de communication suivant différents types de topologies fixes (grille, arbre, hypercube) ou reconfigurables (crossbar, réseau de clos, réseau de benes). De nombreuses architectures matérielles ont été réalisées, des prototypes de recherche comme des produits commerciaux :

- *iPSC/1* et *iPSC/2* hypercube d'Intel,
- *Symmetry* de Sequent,
- *Tnode* et *Meganode* de Telmat
- *Paragon* d'Intel,
- *CM-5* de TMC
- *SP/1* et *SP2* d'IBM

I.1.3 Programmation logique parallèle

Deux principales approches ont été menées conjointement pour le développement de systèmes logiques parallèles :

- l'une repose sur les **langages gardés**, qui sont des langages de programmation logique intégrant un **contrôle explicite** pour exprimer le parallélisme,
- l'autre propose différents schémas d'évaluation permettant d'extraire et d'exploiter diverses formes de parallélisme **implicite** inhérentes à la sémantique des programmes logiques.

Il existe plusieurs sources implicites de parallélisme en programmation logique mais les deux sources principales exploitées par ces deux approches sont :

- le *parallélisme* OU qui survient lorsqu'un appel de prédicat fait apparaître plusieurs alternatives (clauses). Ces chemins multiples sont alors parcourus en parallèle,
- le *parallélisme* ET qui correspond à l'évaluation simultanée des littéraux d'un corps de clause.

L'exploitation automatique du parallélisme OU inhérent aux programmes Prolog **purs** constitue la préoccupation essentielle de notre étude, et ce pour les raisons suivantes :

- le parallélisme OU est présent dans une vaste gamme d'applications [RK90],
- les tâches (branches) évaluées en parallèle sont indépendantes.

C'est dans ce temps de foisonnement de l'évolution des machines parallèles et dans le domaine conjoint du **Parallélisme** et de la **Programmation Logique** que se situent les travaux présentés dans cette thèse. Nous nous sommes appliqués à étudier les mécanismes et stratégies d'évaluation parallèles qu'il est nécessaire de mettre en œuvre pour la conception et la réalisation d'un environnement de programmation Prolog parallèle **efficace** sur des architectures sans mémoire commune. Ces travaux sont une contribution nécessaire pour :

- faire de Prolog une solution à la maîtrise du parallélisme et à l'exploitation d'une large gamme de machines de manière uniforme,
- faire du parallélisme une solution pour l'accroissement de l'efficacité des systèmes Prolog séquentiels,
- faciliter le portage et le développement d'applications parallèles à coût moindre par rapport aux autres langages actuels.

I.2 Problématique

Si les programmes Prolog se prêtent bien à une parallélisation automatique, il reste néanmoins à savoir exprimer, extraire et exploiter efficacement ce potentiel.

La mise en œuvre d'un système qui permette d'exploiter de façon automatique le parallélisme OU impose, dans le cadre de l'implémentation actuelle sur les machines à mémoire distribuée, des défis non-négligeables. Elle fait d'ailleurs toujours l'objet d'une recherche active. L'un de ces enjeux et certainement l'un des plus importants est le développement des techniques d'équilibrage de charge, aux fins de guider l'exécution parallèle des programmes Prolog.

Plus généralement, l'exécution d'un programme logique engendre de nombreuses tâches indépendantes qui, à un instant donné, peuvent être éligibles pour un traitement parallèle. Il se pose alors les problèmes suivants :

- quelles tâches doit-on choisir ?
- quels processeurs doit-on sélectionner ?
- combien de tâches doit-on transférer ?

Ces trois points définissent le problème d'**ordonnancement**, de **placement** et celui du **contrôle de la granularité** inhérents à la mise en œuvre de nombreux systèmes parallèles. La manière dont les tâches sont regroupées et assignées aux processeurs a un impact critique sur la performance globale du système. Cette opération est réalisée par la fonction de **régulation de charge**.

Toutes les applications ne sont pas égales devant le problème de placement. Pour une application présentant un graphe de tâche statique, on connaît à la compilation le nombre de tâches qu'elle met en œuvre et la charge de calcul et de communication que chacune d'elles représente. En réalité, on ne connaît généralement que des valeurs approximatives ou moyennes. Cependant, même si le compilateur peut déterminer précisément ces informations, l'équilibre de la charge et de la communication est difficile à mettre en œuvre pour obtenir **un placement statique** garantissant un temps d'exécution optimal. Ce problème est réputé NP-complet [Bok81, Kle85, CT93].

Contrairement à ce type d'applications, l'élaboration d'une fonction de régulation de charge pour un système logique parallèle nécessite la prise en compte du caractère hautement dynamique de l'évaluation des programmes Prolog, qui ne permet pas de partitionner a priori l'exécution pour un placement statique sur les processeurs. Le problème du placement devient alors plus difficile car il faut le traiter **dynamiquement** au moment même de l'exécution. Sa mise en œuvre représente un coût qui vient s'ajouter au coût d'exécution du programme. Les surcoûts engendrés par la recherche d'une bonne répartition ont un impact critique sur les performances du système. Ceci est particulièrement vrai sur les machines à mémoire distribuée. Si ces coûts ne sont pas maîtrisés, ils peuvent faire chuter les performances du système au point d'obtenir un temps d'exécution plus important qu'en séquentiel. Il convient donc de bien les étudier et de les prendre en compte lors de l'implémentation.

I.3 Cadre de travail : Projet PLoSys

Le projet PLoSys (*Parallel Logic System*) s'inscrit dans une activité initiée en 1986 dans le cadre du projet ESPRIT 1085 Supernode [HJM86] et a été

poursuivi dans le cadre du projet CMAP (Calcul Massivement Parallèle). Ce travail a conduit à la réalisation d'un environnement de programmation Prolog parallèle : OPERA (*Ou Parallélisme et Régulation Adaptative*) [Gey91, Fav92, BFGdK92].

***OPERA*:** La plate-forme OPERA est une implantation parallèle de Prolog sur un TNODE (module de base du Supernode), un réseau reconfigurable de *Transputers.* Elle exploite automatiquement le parallélisme OU d'un programme Prolog *pur* selon une stratégie multi-séquentielle. Cette stratégie est une combinaison dynamique d'une stratégie en largeur et de la stratégie séquentielle classique (en profondeur) pour la construction et le parcours de l'arbre de recherche défini par l'exécution du programme Prolog. Le nombre de processus parallèles y est constamment borné par le nombre de processeurs disponibles sur la machine cible. Les premiers résultats obtenus avec cette maquette se révèlent très prometteurs [Gey91, Fav92, BFGdK92].

Le projet PLoSys repose sur l'expérience et les résultats acquis avec OPERA pour la conception et la réalisation d'un environnement de programmation Prolog parallèle *portable.* Alors que la plupart des recherches dans ce domaine [CH86, dK89, Car90, LK91, Gup91, GSC92b, GJ93b, BT92] choisissaient de travailler sur des machines à mémoire partagée, nous nous sommes orientés vers des architectures parallèles sans mémoire commune de type réseaux de processeurs communiquant par échanges de messages. Un premier point a été de disjoindre les études nécessaires pour la définition et l'évaluation d'une fonction de régulation de charge, afin de garantir l'efficacité du parallélisme d'une part et les études spécifiques de la parallélisation de Prolog pour un exécutif parallèle d'autre part. Ce dernier aspect fait l'objet d'une présentation appropriée dans la thèse d'Eric Morel [Mor96].

I.4 Travail de thèse

Le projet PLoSys est le résultat d'un travail d'équipe. Ma participation au projet m'a conduit à m'intéresser particulièrement à l'étude de fonctions (stratégies) de régulation de charge nécessaires pour le système PLoSys.

L'objet de mes travaux consiste à apporter des éléments de réponse aux questions suivantes :

- Dans quelle mesure une fonction générale de régulation de charge est-elle bonne ?
- Existe-t-il une fonction appropriée à un programme donné ? A une classe de programmes ?

- Est-ce que cette fonction peut s'adapter dynamiquement à différentes classes de programmes ?

La plate-forme OPERA conçue à l'origine et qui devait permettre l'évaluation et la mise au point d'une fonction de régulation de charge a dû être abandonnée en raison de problèmes techniques d'exploitation de la machine hôte. Il a été nécessaire de développer un nouveau cadre d'expérimentation en vue de définir et d'évaluer une fonction de régulation de charge pour le système PLoSys.

Le travail exposé dans cette thèse est le résultat de deux réflexions. La première réflexion portait sur le problème de savoir comment évaluer une fonction de régulation de charge sans le système à réguler. Elle a conduit à la définition et à la réalisation d'un modèle et d'un environnement d'expérimentation qui permettent l'évaluation de fonctions (stratégies) de régulation de charge pour le système PLoSys. La seconde réflexion portait sur l'étude d'une fonction de régulation appropriée au système PLoSys. Une fonction a été élaborée et nous nous sommes attachés à évaluer son impact sur les performances du système à l'aide de notre environnement d'expérimentation.

La définition d'un modèle d'évaluation du système PLoSys a été présentée dans [KMB94, KKMB94, Kan95]. Nous avons choisi l'utilisation d'une technique de modélisation simple dans le but d'*émuler* (simuler) le comportement réel du système sur une architecture parallèle existante. Notre approche repose sur une modélisation de l'exécution d'un programme Prolog par un *graphe de tâches acyclique*. L'exécution de ce graphe de tâches permet l'émulation de l'évaluation d'un programme Prolog.

La définition de notre fonction de régulation de charge s'inspire des travaux [Gey91, Fav92, BFGdK92] initialement effectués dans le cadre du projet OPERA, sans toutefois reprendre certaines optimisations liées à l'architecture de la machine cible. Cette fonction, bien qu'élémentaire, s'est révélée performante pour des programmes Prolog comportant du parallélisme OU et présentant un comportement équilibré. Une étude théorique de notre fonction de régulation a été dès lors entreprise sur différents comportements de programmes Prolog. Cette analyse a été présentée dans [KM94, KMB95]. Cette évaluation a été poursuivie dans un cadre expérimental sur une plate-forme développée à cet effet [KC95]. La plate-forme proposée se veut suffisamment indépendante de la machine cible pour permettre le portage du système sur des architectures voisines. Les premiers résultats de mesure obtenus ont été présentés dans [KCMB96].

I.5 Structure du document

Le chapitre suivant rappelle brièvement les fondements de la programmation logique et les techniques d'implantation du langage Prolog sur des machines sé-

quentielles. Cette présentation est destinée à définir les éléments auxquels nous ferons souvent référence dans la suite du document.

Le chapitre III fait un tour d'horizon des différentes approches qui visent l'exploitation du parallélisme en programmation logique et présente les principaux modèles de parallélisme. Nous y abordons également les aspects liés au problème de l'efficacité et l'obtention de gain de performance dans un cadre général. Nous reprenons plus en détail ces problèmes dans le chapitre IV pour une classe de systèmes Prolog parallèles à laquelle appartient notre système PLoSys : les systèmes OU multi-séquentiels. Nous analysons particulièrement les difficultés de conception et de mise en œuvre liées à l'élaboration d'une fonction de régulation dynamique de charge pour de tels systèmes sur une architecture parallèle sans mémoire commune.

Nous décrivons les choix retenus pour le système PLoSys dans le chapitre V et nous proposons un environnement d'évaluation pour l'étude de notre fonction de régulation de charge.

L'évaluation des différents paramètres de notre fonction de régulation de charge fait l'objet du chapitre VI où nous analysons les premiers résultats obtenus sur cette plate-forme d'évaluation.

Viennent en suite trois annexes qui concluent ce document : l'annexe A présente les programmes du jeu de test utilisés pour les mesures de performance, l'annexe B décrit la méthode que nous avons utilisée pour l'extraction des graphes de tâches et l'annexe C présente l'interface utilisateur développée pour faciliter l'utilisation et l'exploitation de notre environnement d'évaluation.

Chapitre II

PROgrammation LOGique : Principe et Implantation

II.1 Introduction

Un des axes d'évolution de l'informatique porte sur la recherche d'alternatives aux langages de programmation classiques dans lesquels l'utilisateur doit spécifier l'ordre d'exécution des opérations de son programme. Il s'agit de concevoir et d'implanter des langages de plus haut niveau d'abstraction, où le programmeur n'a plus qu'à exprimer les relations existantes entre données et résultats. La solution d'un problème y serait décrite par l'expression des propriétés de la solution souhaitée [Kow79a].

Cette évolution a donné le langage **Lisp** fondé sur le λ calcul (1960) et le langage **Prolog** fondé sur la logique des prédicats du premier ordre (1970).

Après avoir abordé les aspects les plus intéressants du langage Prolog, nous présenterons brièvement, dans la première partie de ce chapitre, les bases théoriques de la programmation logique. La seconde partie est consacrée aux techniques d'implantation du langage sur des machines séquentielles. Cette description est d'abord destinée à initier le lecteur aux bases du langage et de son implantation mais elle permettra également de fixer la terminologie que nous utiliserons souvent dans la suite du document.

II.2 Présentation du langage

Le langage Prolog est un langage ayant un très haut niveau d'abstraction comparativement aux langages procéduraux classiques. L'une de ses principales caractéristiques réside dans le fait que l'algorithme nécessaire à la résolution d'un problème donné n'a pas besoin d'être fourni explicitement par le programmeur.

Le travail du programmeur consiste essentiellement à décrire un **domaine**, composé d'**objets**, ainsi que les **connaissances** qu'il a sur ce domaine, représentées sous forme de relations entre les objets.

Supposons par exemple que l'on ait besoin de traiter un problème portant sur les liens familiaux. On dispose d'un certain nombre de faits constituant la filiation suivante :

Jean est le père de Paul
Jean est le père de Marc
Paul est le père de Pierre

Cet ensemble de connaissances peut être représenté en Prolog par l'ensemble des **faits** suivants :

```
père(paul, jean).
père(marc, jean).
père(pierre, paul).
```

qui est une représentation des relations liant les objets Paul, Jean et Pierre.

D'autre part, nous savons que le grand-père d'une personne est défini comme le père du père de cette personne. Cette connaissance s'exprime en Prolog sous forme de la **règle** suivante :

```
grand-père(X, Y) :- père(X, Z), père(Z, Y).
```

où les variables X, Z et Y représentent respectivement un fils, un père et un grand-père. Cette règle se lit : "Y est le grand-père de X si Z est le père de X et Y et le père de Z".

Pour savoir qui est le père de Pierre, il suffit en Prolog d'énoncer la requête suivante :

```
?- père(pierre, X).
```

où la variable X représente un objet quelconque vérifiant la relation.

L'interprète Prolog répond alors : *X = paul.*

Si nous désirons connaître le père de Jean, la question sera :

```
?- père(pierre, X).
```

et la réponse : *faux.*

Ceci exprime le fait qu'il n'y a pas de père pour Jean dans la mesure où il n'a pas été défini dans les faits.

Pour connaître le fils de Jean, la question se formule par :

```
?- père(X, jean).
```

à laquelle l'interprète répond par les deux solutions : $X = paul$ *et* $X = marc.$

Ces exemples illustrent l'intérêt de la programmation relationnelle, et particulièrement de la propriété de **réversibilité** qu'elle offre, dans la mesure où la relation père se comporte fonctionnellement aussi bien pour obtenir un père qu'un fils. Ainsi, un même programme permet de résoudre des problèmes différents.

Si nous désirons savoir qui est le grand-père de Pierre, la question est :

```
?- grand-père(pierre, X).
```

et provoque la réponse : $X = jean.$

La règle définissant un grand-père peut également être considérée comme définissant un petit-fils et utilisée comme telle. Ainsi, la question :

```
?- grand-père(X, Jean).
```

voit la réponse : $X = pierre.$

Enfin, il est possible d'obtenir tous les couples de personnes tels que l'une est le grand-père de l'autre en posant la question :

```
?- grand-père(X, Y).
```

qui produit la solution unique : $X = pierre,\ Y = jean.$

Ces exemples très simples mettent en évidence les avantages principaux du langage Prolog, où l'expression d'un problème sous forme de relations permet :

- de ne pas exprimer explicitement l'algorithme nécessaire à la résolution du problème,
- de définir en un seul exposé le moyen de résoudre plusieurs types de problèmes.

II.3 Bases théoriques

La programmation logique est un modèle de programmation dérivé de la logique des prédicats du premier ordre. Celle-ci est l'un des formalismes utilisés en logique pour la représentation et le traitement d'axiomes et de propositions dans des domaines scientifiques divers [Kow79b, Col83, Llo87].

Nous rappelons ici la terminologie utilisée en programmation logique. Les concepts utilisés sont présentés par la logique des clauses de Horn, un sous-ensemble de la logique des prédicats du premier ordre. Certaines définitions peuvent être trouvées dans leur contexte général dans [CL73], ou de façon plus orientée vers Prolog dans [SS86].

II.3.1 Clauses de Horn

On suppose disposer des ensembles suivants :

- un ensemble de variables,
- un ensemble de symboles fonctionnels dotés d'une arité,
- un ensemble de symboles de prédicats.

Dans tout ce qui suit, on dénotera les variables par des chaînes alphanumériques commençant par une majuscule et les symboles fonctionnels par des chaînes alphanumériques commençant par une minuscule.

Soit T l'ensemble des **termes** définis par :

- une variable est un terme,
- si f est un symbole fonctionnel d'arité n et t_1, ..., t_n sont des termes, alors $f(t_1, ..., t_n)$ est un terme.

Un **terme structuré** est un terme comportant au moins un symbole fonctionnel d'arité non nulle.

Un **terme clos** est un terme ne comportant aucune variable. L'ensemble des termes clos forme l'**Univers de Herbrand**.

Un **littéral positif** (ou formule atomique) est de la forme $p(t_1, ..., t_n)$ où p est un prédicat d'arité n et $t_1, ..., t_n$ des termes appelés arguments du prédicat.

Un littéral négatif est un littéral positif précédé du symbole de négation $\neg$.

Une **clause** est une disjonction de littéraux positifs et négatifs dont les variables sont universellement quantifiées :

$$\forall X_1, ..., \forall X_s \quad (A_1 \vee A_2 \vee ... \vee A_n \vee \neg B_1 \vee \neg B_2 \vee ... \vee \neg B_m)$$

où A_i et B_j sont des formules atomiques. Les B_j sont les **conditions** de la clause et les A_i sont ses **conclusions**. On simplifie la notation de cette clause de la façon suivante :

$$A_1, A_2, ..., A_n \text{ :- } B_1, B_2, ..., B_m$$

Le symbole *":-"* correspond à l'implication et peut se prononcer *'si'*.

$$\begin{cases} \text{Si } m = 0 & \text{la formule est une } \textbf{assertion inconditionnelle}. \\ \text{Si } n = 0 & \text{il s'agit d'une } \textbf{réfutation}. \\ \text{Si } m = 0 \text{ et } n = 0 & \text{il s'agit de la clause } \textbf{vide} \text{ notée } []. \end{cases}$$

Une **clause de Horn** est une clause contenant au plus un littéral positif ($n \leq 1$). Sa forme générale est donc de trois types :

$$\begin{cases} A \text{ :- } B_1, ..., B_m. & (1) \\ A. & (2) \\ \text{:- } B_1, ..., B_m. & (3) \end{cases}$$

On appelle **programme Prolog** un ensemble formé de clauses de ces trois types. Une clause de type (1) est dite composée d'une **tête** A et d'un **corps** qui est la conjonction des B_i. Une clause **unitaire** (fait) de type (2) est une clause dont le corps est vide (le symbole :- étant omis). La résolution du programme est généralement déclenchée par la formulation d'une clause de type (3) qui est appelée **question** (ou requête).

II.3.2 L'unification

L'unification est l'opération la plus importante dans l'exécution d'un programme Prolog. Elle consiste à déterminer si deux termes sont identiques par instanciation de leurs variables.

a) Substitution et instanciation

Une substitution σ est un ensemble fini de couples (X, t) noté aussi X/t, où X est une variable et t un terme quelconque.

X/t signifie que le terme t peut être substitué à la variable X :

$$\sigma = \{X_i/t_i, \quad i = [1...n] \quad avec \quad X_i \neq X_j \quad si \quad i \neq j\}$$

On dit qu'un terme t_1 est l'instanciation du terme t par σ si t_1 s'obtient en remplaçant simultanément dans t chaque X_i par t_i et on écrit : $\sigma(t) = t_1$

Exemple *si* $\sigma = \{X/a, Z/X, Y/b\}$ *alors* $\sigma(f(g(X), Y)) = f(g(a), b)$.

b) Unification

On dit que deux termes t_1 et t_2 sont unifiables s'il existe une substitution σ telle que : $\sigma(t_1) = \sigma(t_2)$. σ est alors dit unificateur de t_1 et t_2.

Exemple *considérons* $t_1 = f(X, Y)$ *et* $t_2 = f(a, Z)$, *alors la solution* $\sigma = \{X/a, Y/Z\}$ *est l'unificateur de* t_1 *et* t_2 *car :* $\sigma(t_1) = \sigma(t_2) = f(a, Z)$.

Une substitution circulaire est une substitution contenant un élément X/t tel que t contient au moins une occurrence de X. De telles substitutions ne peuvent jamais être des unificateurs. Le test dit d'**occurrence** *(occur check)* permet de les détecter et de provoquer l'échec de l'unification [Pel87]. Peu d'interprètes mettent en œuvre le test d'occurrence en raison du coût exponentiel de son utilisation.

II.3.3 Résolution et interprétation

L'exécution d'un programme Prolog composé d'un ensemble de clauses de Horn est une **réfutation** (preuve par l'absurde) utilisant le principe de résolution [Rob65] comme règle d'inférence.

Soient les deux clauses C_1 et C_2 suivantes :

$$\begin{cases} \text{:- } A_1, A_2, ..., A_m, ..., A_k. & (C_1) \\ A \text{ :- } B_1, ..., B_q. & (C_2) \end{cases}$$

G est une **résolvante** de C_1 et C_2 par résolution si :

$$\begin{cases} A_m & \text{est l'atome sélectionné de } C_1. \\ \sigma & \text{est l'unificateur le plus général de } A_m \text{ et } A. \\ G & \text{est le but :- } \sigma(A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ...A_k). \end{cases}$$

On dit que G dérive de C_1 et C_2.

La résolution est une inférence qui se décompose en deux étapes :

- Choix d'un sous-but dans la résolvante *(règle d'évaluation)*,
- Choix d'une clause dont l'entête est unifiable avec le sous-but considéré *(règle de sélection)*.

Ces deux étapes forment un pas de résolution que l'on appelle **inférence logique**. Le choix d'une règle d'évaluation particulière des sous-buts d'une clause n'affecte pas le résultat. Si un succès est produit pour un ordre donné d'exécution des sous-buts, il est possible de produire le même succès pour tous les autres buts. Prouver par réfutation une conjonction de littéraux revient à effacer tous les littéraux qui la composent, par applications successives du principe de résolution, jusqu'à dérivation de la clause vide. La substitution associée à la dernière résolvante (clause vide) représente alors une solution au problème.

Par contre, le choix de la clause ou règle de sélection est important. Selon la règle de sélection, on peut, pour un même programme et un même but, produire l'un des trois résultats suivants: *succès*, *échec*, *boucler indéfiniment*. La règle de sélection est importante car on peut ne pas trouver de solutions au programme (même si elles existent) si les boucles ne sont pas détectées. Ce problème est indécidable dans son cas général [Llo87].

La résolution *SLD* :
En considérant que l'on s'intéresse à au moins une solution du problème et que le "bon choix" de la clause n'est pas évident, il faut prévoir une **stratégie de recherche** *(règle de recherche)* en cas d'échec :

- quel sous-but précédemment unifié choisir ?
- quelle nouvelle clause choisir (ré-application de la règle de sélection en éliminant la clause qui a échoué) ?

Cette action est nommée **retour-arrière** (*backtracking*). La combinaison, d'une règle d'évaluation et d'une stratégie de recherche (règle de sélection incluse), est nommée **procédure de résolution SLD** [KV74].

La résolution SLD, résolution Linéaire avec fonction de Sélection pour les clauses Définies (de Horn), est la méthode de preuve par réfutation utilisée en programmation logique. Le processus de résolution consiste à dériver du but initial (requête) la clause vide :

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

II.3.4 Stratégie de résolution

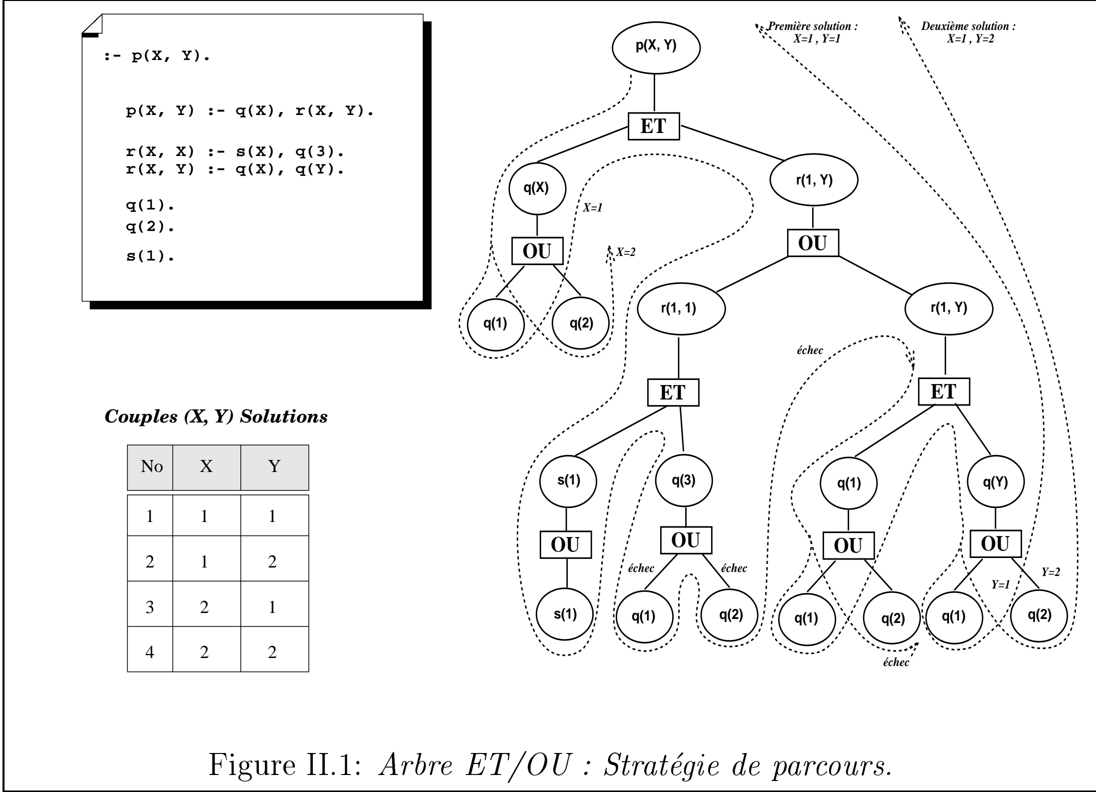
Etant donnée une règle de sélection, on peut représenter une réfutation SLD (preuve) d'un programme par un arbre ET-OU.

Considérons par exemple le programme de la figure II.1. La preuve de ce programme peut être représentée par le parcours d'un arbre ET-OU (cf. figure II.1). Chaque nœud ET représente la conjonction des sous-buts à prouver et contient la résolvante courante. La racine est un nœud ET et contient le but originel. Chaque branche ET (sortant d'un nœud ET) correspond à une sélection de sous-buts.

Chaque nœud OU représente l'ensemble des alternatives de choix d'une clause possible. Chaque branche OU (sortant d'un nœud OU) correspond à une sélection de clause.

Le calcul d'une solution peut être schématisé par le parcours d'une partie de cet arbre en appliquant les règles suivantes :

- passer par tous les arcs issus des nœuds ET,
- et passer par un des arcs issus des nœuds OU.



No	X	Y
1	1	1
2	1	2
3	2	1
4	2	2

Figure II.1: *Arbre ET/OU : Stratégie de parcours.*

Un parcours de l'arbre depuis la racine, en respectant ces deux règles, jusqu'à une feuille (clause vide), est une solution du programme. Un programme ayant plusieurs solutions correspond à l'existence de plusieurs alternatives de parcours aboutissant à une feuille.

Etant donné que la règle d'évaluation n'affecte pas les résultats, on peut maintenir une seule branche par nœud ET. En figeant une règle d'évaluation, les nœuds ET sont omis. L'arbre ET-OU se transforme alors en un arbre OU.

L'exécution du programme consiste en un parcours exhaustif de son arbre d'évaluation ET-OU. Un effort important a été consacré à la définition de stratégies de parcours de cet arbre (procédure de résolution SLD). On distingue deux grandes classes de stratégies :

a) Les stratégies *aveugles*

Ces stratégies fixent un ordre de parcours *'a priori'* de l'arbre [Bra88b, Row88, Sho94]. Elles diffèrent de par les propriétés garanties par cet ordre. Les stratégies rencontrées dans l'implantation de Prolog sont :

- parcours en **profondeur d'abord** *(depth first)*,
- parcours en **largeur d'abord** *(breadth first)*,

- parcours **mixte** *(deep iterative deepenning).*

La stratégie *en profondeur d'abord* est généralement utilisée dans toutes les implantations de systèmes Prolog actuels. Elle définit la règle de résolution (SLD) suivante :

- sélection des sous-buts dans l'ordre de leur écriture,
- sélection des clauses dans l'ordre de leur écriture,
- retour-arrière sur la clause (alternative) suivante (dans l'ordre d'écriture) à celle ayant échouée.

b) Les stratégies *intelligentes*
Le parcours de l'arbre de preuve avec ses stratégies dépend d'heuristiques portant sur les propriétés structurelles du parcours (conditions globales) ou sur l'état des variables (conditions locales) [Bra88b, Row88, Sho94].

Ces approches ont conduit à des variantes de langages proposant, soit une extension du langage Prolog pour exprimer ces fonctions de sélection [DL84], soit essayant d'intégrer des algorithmes de recherche heuristique (de type A^*), ou bien encore offrant un canevas permettant d'exprimer des systèmes de contraintes à satisfaire sur un parcours [DHS+88, Col90].

II.4 Effets de bord

Il serait faux de déduire des sections précédentes que l'implantation du langage Prolog peut être considérée comme une mise en œuvre d'un démonstrateur de théorèmes par l'application du principe de résolution. Prolog est un langage de programmation à part entière, dans la mesure où il comprend des opérateurs de communication avec le monde extérieur (entrées/sorties), des opérateurs de contrôle du processus de résolution pour influer sur la stratégie, et des opérateurs de retrait ou d'ajout de clauses. Ce passage d'un modèle logique à un langage de programmation est réalisé par l'adjonction de prédicats dits *non-logiques* ou à effets de bord.

- *Opérateurs méta-logiques* : Ces opérateurs sont utilisés pour la manipulation des termes Prolog et l'interrogation de l'état de la résolution (l'opérateur *var/1* permet de déterminer si une variable est liée ou pas).
- *Opérateurs extra-logiques* : La plupart des implantations de Prolog offrent divers prédicats prédéfinis de communication avec le monde extérieur. Ces prédicats permettent la sortie de termes *(write/1)*, ainsi que l'entrée de

termes *(read/1)*. D'autres opérateurs permettent de modifier dynamiquement les prédicats pendant l'évaluation du programme. Des faits peuvent être ajoutés *(assert/1)* ou retranchés *(retract/1)* de la base, modifiant ainsi le comportement du programme.

- *Opérateurs de contrôle* : Le retour-arrière permet de retourner sur un nœud OU après un succès ou un échec. Ce comportement est très utile car il décharge le programmeur d'expliciter une recherche exhaustive de solutions. Cependant, du fait de sa généralité, le retour-arrière peut se révéler inefficace. Il est donc important d'avoir un moyen qui permette de le contrôler ou de l'interdire. La plupart des systèmes Prolog offrent pour cela des opérateurs permettant de contrôler la stratégie de résolution (parcours de l'arbre d'évaluation) : la **coupure** notée *!* et l'**échec** forcé noté *fail*.

II.5 Implantation de Prolog : la WAM

La réalisation par l'équipe d'A. Colmerauer [Rou75] du premier interprète Prolog remonte à 1972. Les performances relativement faibles des premiers interprètes ont motivé de nombreux travaux sur sa compilation.

L'idée à la base de la compilation de Prolog consiste à utiliser la connaissance que l'on a des clauses constituant le programme, de sorte que l'on puisse faire une spécialisation de l'algorithme d'unification qui, dans le cas de l'interprétation, était le plus général. L'aboutissement de ces travaux a été la réalisation en 1977 du premier compilateur Prolog par D.H. Warren [War77, War83].

Warren a défini une machine abstraite *(la WAM)* dont l'efficacité par rapport aux techniques d'interprétation a considérablement marqué les développements ultérieurs. Cette machine abstraite est devenue le *"standard de fait"* de la compilation Prolog.

L'utilisation de cette technique d'implantation répond à un double objectif :

- faciliter la portabilité du compilateur du langage,
- servir de base à la réalisation de machines Prolog spécialisées ou de systèmes logiques parallèles.

La WAM est définie par une organisation de sa mémoire et un jeu d'instructions. La description qui suit présente le processeur abstrait et les principales catégories de primitives de cette machine. Nous conseillons au lecteur intéressé par plus de détails de se référer à l'article original de D.H.D Warren [War77] et au rapport de H. Ait Kaci [AK92a].

Le code WAM généré peut être traité de trois manières :

- par un émulateur sur une machine spécifique [Jem92],
- par traduction en code natif d'une machine spécifique [Roy84],
- ou par un processeur dédié [Her86a].

II.5.1 Organisation mémoire

Lors de l'exécution d'un programme, la machine abstraite manipule cinq zones mémoires dans lesquelles sont gérées les données (cf. figure II.2) :

- la zone de **code** : elle contient l'ensemble des instructions générées à la compilation du programme ;
- la pile **locale** : utilisée pour le contrôle des appels de prédicats et la gestion du retour-arrière ;
- la pile **globale** *(le tas)* : utilisée pour la gestion des termes structurés ;
- la pile **traînée** : utilisée pour la mise à jour des variables lors du retour-arrière ;
- la pile **d'unification** : utilisée comme pile de travail durant le processus d'unification de deux termes.

La machine abstraite manipule également un nombre important de registres ; certains représentent l'état d'exécution du programme *(registres d'état)* et d'autres sont utilisés durant l'unification et l'invocation de prédicats *(registres d'arguments)*.

La pile locale contient deux types de structures de données :

- les `environnements` : un environnement est créé chaque fois qu'une clause contenant plusieurs sous-buts est invoquée. Il est utilisé pour la sauvegarde des variables accédées par les différents buts dans le corps de la clause *(variables permanentes)*. Sa taille est déterminée par le nombre de variables de la clause. Il correspond à la notion de bloc d'activation de procédure dans la compilation des langages classiques. Il contient les informations nécessaires pour le retour à la clause appelante. A la fin de l'exécution de la clause, l'environnement correspondant est libéré.
- les `points de choix` : un point de choix est créé si le prédicat appelé par un sous-but est non-déterministe (nœud OU). Il contient une sauvegarde de l'état de la machine lors d'un retour-arrière sur une alternative inexplorée. Le retour-arrière s'effectue à partir du point de choix le plus récent. l'espace alloué pour un point de choix est récupéré lorsque la dernière alternative du prédicat est invoquée.

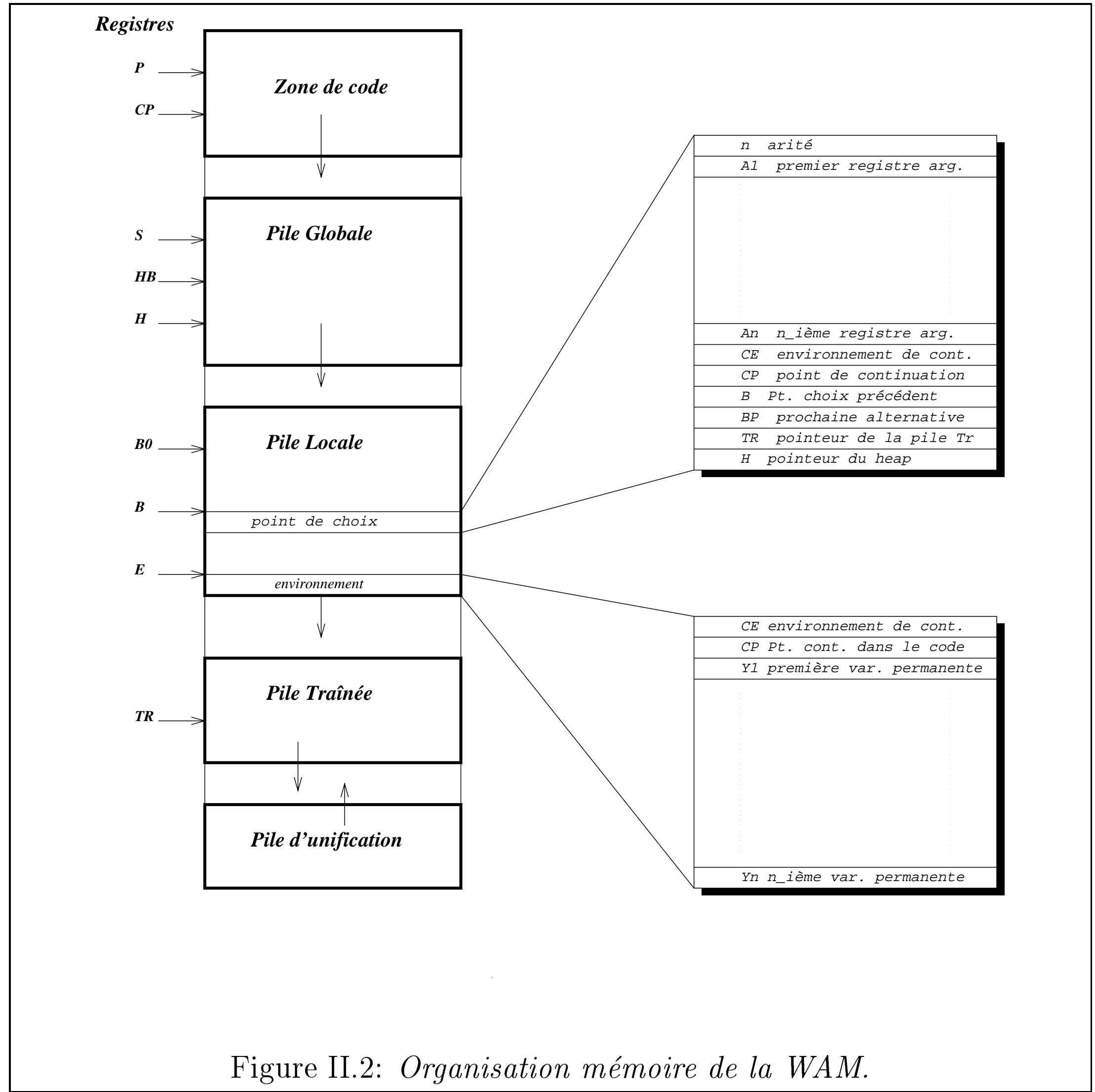


Figure II.2: *Organisation mémoire de la WAM.*

Les termes structurés, nécessaires à l'exécution du programme, sont construits dans la pile globale. L'espace alloué n'est récupéré que lors d'un retour-arrière sur un point de choix, créé avant le terme.

Une variable logique est représentée par une cellule de mémoire, dans la pile locale, si elle est locale à une clause ; ou dans la pile globale, si elle est un sous-terme d'un terme structuré. L'initialisation de la cellule (à sa création) la met à libre *(valeur inconnue)*. Lors d'une substitution, la variable prend une valeur. Cette substitution est représentée par une référence *(un pointeur)* vers le terme qui représente la valeur de la variable. Cette opération est appelée **liaison**. Si un point de choix sépare la création d'une variable de l'instant de sa première liaison, cette liaison est dite **conditionnelle**. Dans le cas d'un retour-arrière sur

ce point de choix, la liaison doit être défaite. Les liaisons conditionnelles sont maintenues dans la pile traînée.

II.5.2 Jeu d'instructions

On distingue deux grandes catégories d'instructions : les instructions de contrôle liées à l'enchaînement des opérations logiques (conjonction/disjonction) et les instructions d'unifications (cf. figure II.3) qui gèrent la création et l'accès aux données (termes).

```
p/2 :   allocate                %   p
        get_variable X2, A1     %    (X,
        get_variable Y1, A1     %       Y) :-
        put_value X2, A1        %   q(X,
        put_value Y2, A2        %       Z
        call q/2                %        ),
        put_value Y2, A1        %   r(Z,
        put_value Y1, A2        %       Y
        call r/2                %        )
        deallocate              %   .
```

Figure II.3: *Instructions WAM pour p(X,Y) :- q(X,Z),r(Z,Y).*

Instructions de contrôle
Le contrôle est implanté de façon à respecter la stratégie d'évaluation séquentielle choisie pour Prolog. Le contrôle dit *déterministe* correspond au parcours de *gauche à droite* pour évaluer les conjonctions d'une clause, et le contrôle du *non-déterminisme* correspond aux alternatives d'évaluation des différentes clauses d'un prédicat dans l'ordre de leur écriture dans le programme source.

Instructions d'unification
L'unification s'effectue entre les paramètres d'une clause (d'un prédicat) et les arguments d'appel d'un but. L'unification de chaque type d'argument (une variable, une constante ou une structure) est réalisée avec une instruction spécifique. Chaque instruction est donc une spécialisation de l'algorithme d'unification.

II.6 Efficacité séquentielle

Prolog est considéré comme un langage propre à permettre le développement rapide d'applications [Row88, Nil90, Sea75, Sea92, Sho94, Fil94]; par contre, comparé aux langages algorithmiques, Prolog apparaît plutôt inefficace. Des études [Rei88, Paa88, Roy90] montrent un facteur de 1 à 10 des temps d'exécution de Prolog et de ceux du langage C. L'utilisation d'un système compilé et le développement de techniques d'optimisation [Kan90] ont contribué à réduire ce facteur, mais une différence significative persiste.

Aujourd'hui, les travaux visant à améliorer l'efficacité des systèmes Prolog se poursuivent dans différentes directions :

- l'exploitation des propriétés extraites par l'analyse de programme [CC77, Oud87, Bru91, CC92, Deb92, BCHP95],
- le développement de nouveaux modèles d'exécution [KN90, Tar95],
- l'extension aux contraintes [Col90, JJ94, GSUJ94, BH95, CH95],
- l'exploitation du parallélisme [CK81, Lin88, FT90, Kac90, Tic91, Cra92].

II.7 Conclusion

Comparé aux langages classiques de programmation algorithmique conçus pour le calcul numérique, un langage de programmation logique présente deux différences fondamentales :

- Il est destiné à la programmation *symbolique* : les données de base sont des *termes* (objets) qui ne représentent qu'eux-mêmes et ne sont pas interprétés par l'exécution des programmes.
- Il ne spécifie pas directement des algorithmes à exécuter pour produire des résultats mais il spécifie des objets, des propriétés d'objets et des relations entre objets.

L'utilisation de Prolog comme un langage à part entière en informatique est longtemps restée confrontée à un problème d'efficacité. Toutefois, les avantages du langage et la compétition industrielle déclenchée par le projet japonais d'ordinateurs de cinquième génération [Kur88], dans lequel Prolog ou un de ses dérivés servirait de base pour l'exploitation de machines parallèles, ont provoqué le lancement de nombreuses études destinées à corriger ses inconvénients par le développement de techniques efficaces à son implantation.

Historiquement, cette recherche d'efficacité portait d'abord sur l'amélioration des interprètes, et ensuite sur les compilateurs. Le résultat principal a été la définition d'une machine abstraite adaptée à la résolution de Prolog, *"la Warren Abstract Machine"* [War83], cible de la plupart des compilateurs actuels. Cette machine permet la prise en compte d'optimisations élémentaires comme l'optimisation de l'appel terminal [Kan90] ou la spécialisation de l'unification en fonction de contextes précis. Toutefois, les optimisations possibles dans le cadre de la machine de Warren [AK92a] et de la compilation *"naïve"* qui lui correspondent semblent avoir atteint leurs limites.

Actuellement, les travaux les plus récents portent d'une part sur l'amélioration de la compilation elle-même et l'exploitation des propriétés extraites par l'analyse de programme, [DLH90, Deb92, DR94] et d'autre part, sur l'exploitation du parallélisme par le développement de mécanismes et de stratégies d'évaluation parallèles efficaces. Ce dernier aspect ouvre de nouvelles voies à la programmation logique mais rajoute également plusieurs dimensions au problème d'implantation. La suite de cette thèse y est consacrée.

Chapitre III

Systèmes logiques parallèles

III.1 Introduction

L'idée du parallélisme est simple : faire effectuer par plusieurs machines la tâche normalement dévolue à une seule, dans le but d'améliorer l'efficacité des applications informatiques. Cependant, si l'augmentation croissante du nombre de processeurs dans les machines permet de satisfaire les besoins des applications en vitesse de calcul et en place mémoire, il reste néanmoins à transformer cette puissance de calcul parallèle en une amélioration des performances d'exécution d'un programme utilisateur.

En effet, l'utilisation générale du parallélisme dans le traitement informatique nécessite de répondre à un certain nombre de questions fondamentales [DDG85] [BST89] :

- comment découper le traitement à effectuer en entités pouvant être traitées en parallèle ?
- comment exploiter et mettre en œuvre ce découpage sur l'ensemble des processeurs d'une machine parallèle particulière ?
- comment contrôler l'exécution de l'ensemble de ces entités afin de tirer profit au mieux des ressources de calcul disponibles ?

L'exploitation du parallélisme en programmation logique est confrontée également à ces mêmes interrogations. Ce chapitre présente les éléments de réponse apportés par les différents travaux de recherche effectués dans le domaine. Nous présentons tout d'abord les principales sources de parallélisme en programmation logique et les problèmes généraux liés à leur exploitation. Nous décrivons par la suite les différentes approches existantes. La dernière partie du chapitre est consacrée à la présentation de notre approche.

III.2 Parallélisme en programmation logique

Comme nous l'avons vu dans le chapitre précédent, la sémantique des langages de programmation logique n'est pas a priori séquentielle. La stratégie SLD-résolution n'impose pas de stratégie pour l'exploration de l'arbre de réfutation associé à un programme. Pour cette raison, les langages logiques sont souvent considérés comme les meilleurs candidats pour la programmation de machines parallèles.

Si un système Prolog séquentiel met en œuvre une stratégie de parcours *(en profondeur d'abord, en largeur d'abord, ou heuristique)* de l'arbre ET/OU généré par l'évaluation du programme, un système Prolog parallèle exploite le parallélisme suggéré par la structure de cet arbre et met en œuvre une stratégie de parcours parallèle pour l'évaluation simultanée de plusieurs branches de cet arbre.

III.2.1 Les sources de parallélisme

Intuitivement, on divise les différentes sources de parallélisme de la programmation logique en deux grandes classes : le parallélisme OU, qui permet l'évaluation en parallèle des clauses d'un même prédicat, et le parallélisme ET qui consiste à évaluer en parallèle la conjonction des sous-buts d'une clause. A côté de ces deux sources de parallélisme, on peut également penser à paralléliser l'opération d'unification des termes.

Nous présentons brièvement ces trois formes de parallélisme (figure III.2) à travers l'exemple de programme de la figure III.1 et l'arbre d'évaluation ET/OU correspondant.

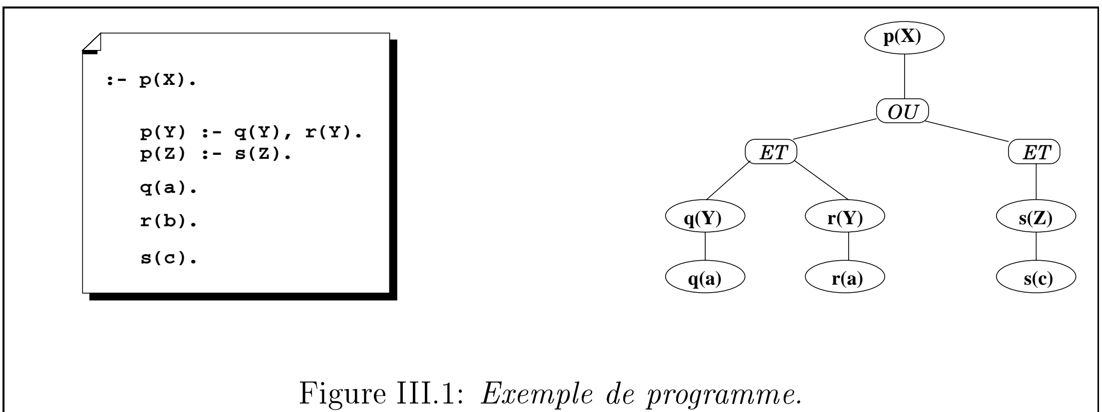


Figure III.1: *Exemple de programme.*

a) Le parallélisme OU

Cette forme de parallélisme résulte du non-déterminisme inhérent aux programmes Prolog. Un prédicat peut être défini par plusieurs clauses. Chacune de

ces clauses constitue une alternative de résolution à l'appel de ce prédicat.

Dans un système séquentiel, le paquet de clauses est exploré une clause après l'autre, dans l'ordre de leur apparition dans le programme, en utilisant le mécanisme du retour-arrière. Dans un système OU-parallèle, plusieurs alternatives sont essayées simultanément pour résoudre un but donné. Les résolutions correspondant à ces alternatives sont indépendantes mais partagent l'état de la résolvante (conjonction de sous-buts à résoudre) initiale à l'appel du but considéré.

Pour le programme figure III.2, l'appel du prédicat p génère un point de choix (nœud OU) avec deux alternatives de résolution. Les deux résolvantes à ce point de choix sont respectivement la conjonction des sous-buts : $\{q(X),\ r(X).\}$ et $\{s(X).\}$ Les deux résolvantes partagent la même variable logique X mais leurs continuations sont indépendantes. La première résolvante produit un échec à l'appel de r alors que la seconde génère la solution $X = c$. Un système OU-parallèle exploite cette caractéristique d'indépendance et traite ces deux résolvantes en parallèle.

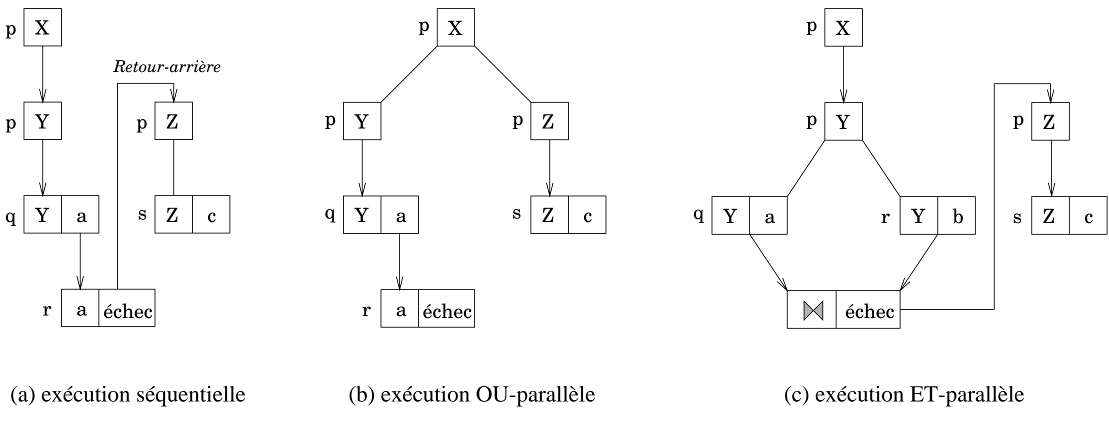


Figure III.2: *Sources de parallélisme en Prolog.*

Le parallélisme OU est exploitable pour de grandes classes de problèmes qui sont intrinséquement non-déterministes. On peut citer, par exemple, des algorithmes de recherche dans les graphes qui sont essentiels en intelligence artificielle, dans l'analyse des langages naturels ou des traitements de bases de données. Enfin, notons que les systèmes experts se programment aisément en Prolog dont l'interprète est déjà un moteur d'inférence, le programme jouant le rôle de base de règles.

b) Le parallélisme ET

La stratégie de résolution de Prolog choisit un littéral (le plus à gauche) de la résolution courante et essaie de satisfaire ce but. Le parallélisme ET consiste à

prendre plusieurs littéraux et à mener leur résolution en parallèle.

Dans le programme figure III.2, l'évaluation de la première clause du prédicat p consiste à réfuter la résolvante $\{q(X),\ r(X).\}$. Un système ET-parallèle tente d'évaluer les deux sous-buts $q(X)$ et $r(X)$ en parallèle. Le problème qui se pose dans ce cas provient de la dépendance de certains littéraux (la variable logique X est commune aux deux sous-buts q et r). Cette dépendance peut provoquer des résultats incompatibles (la résolution de $q(X)$ génère la solution $X = a$ alors que la résolution de $r(X)$ génère la solution $X = b$). Remédier à cela passe généralement par le calcul d'une jointure des sous-ensembles solutions générés par chaque sous-but ($X = \{a\} \cap \{b\} = \emptyset$).

Le parallélisme ET apparaît dans tous les programmes logiques. Cependant, ainsi que nous le verrons dans la section suivante, le problème d'interdépendance des variables dans les sous-buts rend difficile son exploitation efficace.

c) Le parallélisme d'unification

L'unification pourrait être une source de parallélisme dans la mesure où l'on pourrait envisager d'effectuer les unifications des sous-termes de structure en parallèle. Après ces unifications, comme dans le cas du parallélisme ET, il faudrait vérifier la cohérence des solutions produites par chacun des sous-termes unifiés de façon indépendante.

Considérons par exemple l'unification des deux termes suivants : $p(a,\ X,\ b)$ et $p(Z,\ Y,\ b)$, les trois unifications peuvent être effectuées simultanément. Par contre, pour les deux termes $p(a,\ X,\ b)$ et $p(Z,\ Z,\ b)$, l'unification des arguments X et Z ne peut avoir lieu que si la première a réussi.

Les avis concernant la viabilité de l'exploitation de cette source de parallélisme sont partagés : certains considèrent que cette opération est intrinséquement séquentielle [DKM84], d'autres ont proposé une forme parallèle d'implantation de l'algorithme [Bar90]. Il y a eu également quelques études de faisabilité pour des architectures de processeurs spécialisés réalisant l'unification [SP89].

III.2.2 Types d'exploitation

Le parallélisme présent dans les programmes Prolog peut être exploité aussi bien sur des architectures SIMD que sur des architectures MIMD. Cependant, comme pour les autres langages, la classe de programmes pouvant tirer parti des architectures SIMD est restreinte par rapport à celles qui peuvent être exploitées sur des architecture MIMD.

Il est possible d'exploiter toutes les sources de parallélisme implicites présentes dans les programmes Prolog. Cependant, aucun système parallèle n'a atteint ce but dans la mesure où :

- il est difficile d'exploiter efficacement une seule source,
- il est difficile d'exploiter de façon conjointe toutes les sources.

Ainsi, c'est encore un objectif de recherche que de concevoir un système Prolog parallèle efficace. La vaste littérature en programmation logique parallèle, à travers les publications dans les journaux, congrès et conférences, est un indicateur significatif de la recherche effectuée dans le domaine. Cette thèse ne suffirait pas à couvrir tous les aspects du domaine. Nous présentons dans la suite du chapitre une synthèse des approches existantes dans l'exploitation du parallélisme en programmation logique.

a) Parallélisme de données

Des travaux récents proposent des systèmes parallèles considérant les deux sources principales de parallélisme en Prolog (OU et ET) comme un parallélisme de données. Cette approche vient de la constatation que certaines classes de programmes présentent une structure composée de données sur lesquelles on doit répéter une même action.

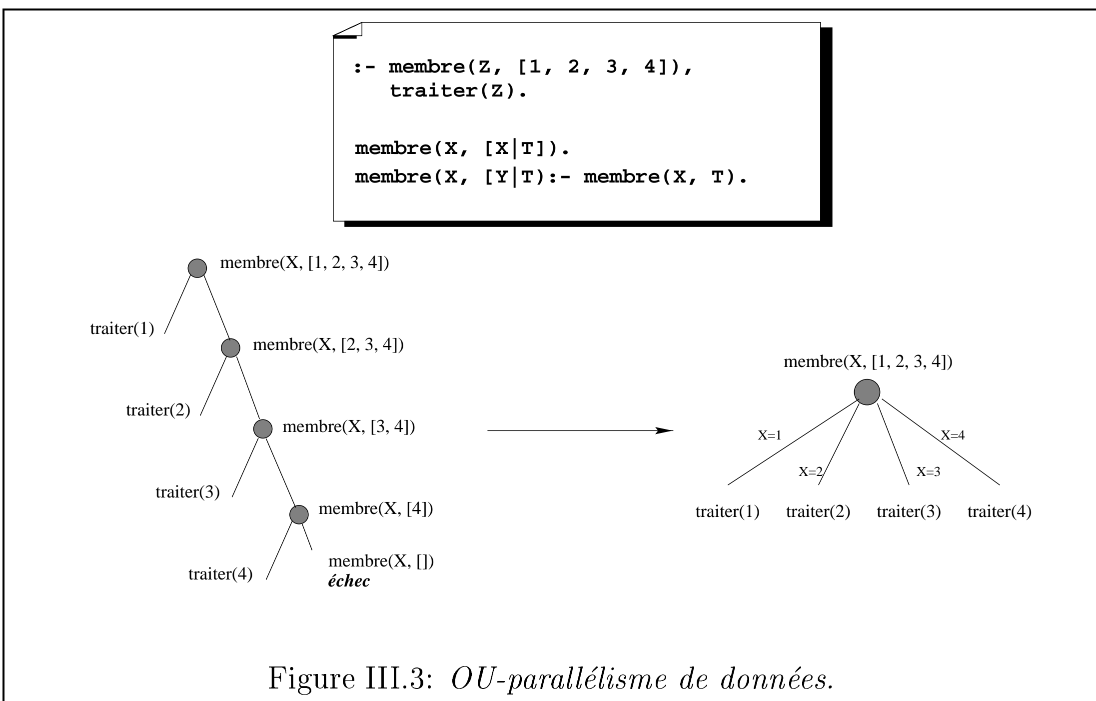


Figure III.3: *OU-parallélisme de données.*

OU-parallélisme de données

Dans cette approche de systèmes, le but est d'exploiter la régularité du OU-parallélisme sur des architectures SIMD. Considérons l'exemple de programme de la figure III.3.

Le but `membre` est identifié dans ce type de système comme générateur de solutions. La résolution de ce but produit un ensemble de solutions représentées par les substitutions possibles de la variable `Z`. Le but `traiter` est alors exécuté sous forme de parallélisme de données OU-Parallèle, pour chaque liaison de la variable `Z`. L'exécution des différents buts `traiter` diffère uniquement par la valeur de la variable `Z`. Un seul flux de contrôle est nécessaire pour opérer sur les valeurs différentes de `Z`.

MultiLog [Smi93] est un système qui exploite cette approche. Pour des programmes se conformant à certains schémas d'exécution, MultiLog affiche de bonnes performances.

ET-parallélisme de données

L'idée du parallélisme de données peut également être appliquée au traitement parallèle d'une conjonction de sous-buts. Les clauses contenant des appels récursifs sont aplaties *(unfolded)* et les buts résultants sont traités en parallélisme de données.

Considérons l'exemple de programme de la figure III.4.

```
:- rev([1, 2, 3], Z).

   rev([], []).
   rev([H|T], R) :- rev(T, Tr),
                    concat(Tr, [H], R).
```

Figure III.4: *ET-parallélisme de données.*

la mise à plat du but `rev([1, 2, 3], Z)` génère la requête suivante :

```
:- rev([], Tr2),
   concat(Tr2, [3], Tr1),
   concat(Tr1, [2], Tr),
   concat(Tr, [1], Z).
```

Les trois sous-buts `concat/3` dans la requête sont identiques, excepté pour les données traitées. L'exécution de la requête peut se faire en deux étapes de parallélisme de données.

Reform Prolog [BLM93] est un exemple de systèmes exploitant cette approche. Il affiche également de bonnes performances pour certains types de programmes.

b) Parallélisme de contrôle

La plupart des travaux se sont concentrés sur l'exploitation du parallélisme sur des architectures MIMD, en considérant le OU-*parallélisme* et le ET-*parallélisme* comme une forme de parallélisme de contrôle. Cette approche vient de la constatation que le parcours de l'arbre d'évaluation effectué par un interprèteur Prolog peut être effectué simultanément par plusieurs processeurs travaillant sur des parties disjointes de l'arbre ET/OU.

Cependant les difficultés techniques rencontrées à différents niveaux, notamment pour l'extraction et l'exploitation efficace de ce potentiel, ont conduit au développement de systèmes exploitant une forme particulière du parallélisme. On distingue principalement deux approches :

- L'une repose sur la modification du langage par l'introduction d'annotations dans le programme pour exprimer le parallélisme. On obtient alors un nouveau langage où l'expression du parallélisme est **explicite**.

- L'autre propose des schémas d'évaluation permettant d'extraire des formes particulières de parallélisme. Le parallélisme est **implicite**.

La première approche vise à introduire dans le langage des mécanismes de synchronisation qui soient similaires aux mécanismes formalisés dans le modèle *CSP* [Dij68] et que l'on trouve dans un langage impératif parallèle comme *Occam*. La seconde approche caractérise les systèmes parallèles dits **non-déterministes** ou multi-solutions de la programmation logique *"pure"*.

III.2.3 Parallélisme explicite

Un aspect de cette approche concerne les **langages gardés** [Ued86]. Les langages gardés fournissent au programmeur des mécanismes pour expliciter le contrôle du parallélisme à travers le concept de flot et de synchronisation (voir parallélisme ET-Dépendant dans la section section III.3.2). ***Parlog*** [CG86], ***Concurrent Prolog*** [Sha87, Sha89] et ***Guarded Horn Clauses*** [Ued86] sont trois exemples de systèmes adoptant cette approche.

La caractéristique principale de ses trois langages est l'introduction du concept de **gardes** [Ued86]. Dans un langage gardé une clause a la forme suivante :

$$A \text{ :- } G_1, ..., G_m | B_1, ..., B_n. \quad m, n \geq 0$$

où A est la tête de la clause, et $G_1, ..., G_m$ sont les gardes de la clause, le symbole | est appelé opérateur d'engagement *(commit)* et $B_1, ..., B_n$ représentent le corps de la clause.

Les gardes sont des conditions qui doivent être vraies pour que, lors d'un appel, la clause fasse partie du sous-ensemble de clauses qui peuvent être activées. Le contrôle du non-déterminisme est effectué sur ce sous-ensemble : une seule clause est aléatoirement choisie pour continuer l'exécution. Les autres clauses sont arrêtées et le retour-arrière n'est jamais utilisé. On élimine alors la possibilité offerte par Prolog d'une recherche exhaustive de solutions. Le principe d'engagement, qui écarte définitivement les clauses alternatives même si celles-ci peuvent conduire à une solution, risque de donner un échec à une requête qui, au vu de Prolog, possède une solution.

Le non-déterminisme (parallélisme OU) se limite à l'exécution simultanée des unifications, avec les têtes de clauses et l'évaluation de leurs gardes. Le non-déterminisme **libre** de la programmation logique *(don't know non-determinism)* est ainsi remplacé par le non-déterminisme **contrôlé** *(don't care non-determinism)*.

Le parallélisme ET est géré en introduisant des mécanismes de synchronisation de type *producteur/consommateur* entre les différents littéraux du but courant. Cette synchronisation est réalisée en précisant, pour chaque prédicat (dans Parlog [CG86] et dans Guarded Horn Clauses [Ued86]) ou dans chaque littéral, (dans Concurrent Prolog [Sha87, Sha89]) le mode d'accès à ses arguments. Une occurrence d'une variable (non liée) en entrée d'un littéral indique que celui-ci est *consommateur*. L'exécution parallèle de ce littéral est alors mise en attente tant que cette variable n'a pas été liée par un littéral *producteur*. Ainsi le problème des liaisons conflictuelles à une même variable est écarté.

```
Procédure a/1
DébutPar
   a(X) :- b(Y), c(X,Y), d(X).
   DébutSeq
      a(X) :- e(X), f(X).
      a(X) :- g(X).
   FinSeq
FinPar
```

Figure III.5: *Parallélisme explicite.*

Un autre aspect de cette approche consiste à introduire, dans le langage ou sous forme de **bibliothèques** *(prédicats pré-définis)*, des mécanismes de commu-

nication et de synchronisation que l'on trouve dans un langage impératif parallèle comme *Occam* (CS-Prolog [Fut93] et Delta-Prolog [ACMP86] adoptant ce type d'approche. Shared-Prolog [Cia93] introduit également des primitives de communication fondées sur le principe du *tableau noir (blackboard)*. On est alors amené à l'expression d'un programme qui est défini comme un ensemble de processus communicants dans une optique similaire à celle formalisée dans les modèles *CSP* [Dij68].

La figure III.5 représente un exemple d'extension du langage Prolog [Pas87] pour expliciter et imposer une structure d'exécution parallèle. Des annotations que l'on retrouve dans les modèles *CSP* [Dij68] sont utilisées pour contrôler explicitement le parallélisme. Dans cet exemple particulier de la figure III.5, ces annotations permettent de contrôler le nombre de processus OU-parallèle initialisés à l'appel du prédicat *a/1*. Deux tâches (processus) sont activées en parallèle à l'appel de *a/1*. L'une traite la première clause alors que l'autre traite en séquence la seconde et la troisième. Les solutions sont consommées par le processus appelant *(père)*, dans l'ordre de leur arrivée. L'un des problèmes de ce type d'approche est que l'exécution du programme produit un graphe de tâches évoluant dynamiquement et de taille non bornée a priori (du fait de la récursivité des appels de prédicat).

Une critique majeure de ces approches réside dans le fait que l'on s'éloigne de la programmation logique et surtout que l'on reporte sur le programmeur les problèmes d'expression et de gestion du parallélisme.

III.2.4 Parallélisme implicite

Contrairement aux langages gardés, les systèmes non-déterministes sont capables de fournir, de la même façon que les implantations séquentielles de Prolog, toutes les solutions d'une requête. Ces modèles ne proposent pas d'extension de la sémantique de Prolog dans la direction du parallélisme. En particulier, ils ne fournissent pas de communications explicites ou implicites entre processus parallèles et ils visent à une relative compatibilité avec les implantations séquentielles de Prolog.

Plusieurs modèles de parallélisation *(de calcul)* ont été proposés dans le cadre de l'implantation de systèmes non-déterministes. Un modèle de calcul est caractérisé par la forme de parallélisme exploité et par la façon dont l'exécution parallèle est contrôlée. La stratégie de contrôle de l'exécution parallèle permet encore de distinguer deux classes de modèles non-déterministes : les modèles théoriques et les modèles multi-séquentiels.

a) Les modèles théoriques

Le but principal des **modèles théoriques** est d'extraire et d'exploiter *"tout"* le

parallélisme potentiel des programmes Prolog. Ils sont souvent qualifiés dans la littérature de **modèles de parallélisme massif**. Dans ces modèles, l'accent n'est pas mis sur la faisabilité d'une implémentation efficace sur les matériels existants mais plutôt sur l'importance du parallélisme extrait.

L'exemple classique de ce type d'approche est le modèle ET/OU proposé par Conery [CK81, Con83, CK83, Con87b] pour l'exécution de programmes logiques *purs*. Dans son modèle [CK81], chaque étape de l'exécution donne lieu à la création de deux types de processus : les processus **ET** et **OU**. Sont créés autant de processus ET qu'il y a de sous-buts dans une clause, alors qu'un processus OU est chargé de résoudre un littéral à l'intérieur d'une clause. Chaque processus crée d'autres processus (ses *fils*) au fur et à mesure que l'exécution avance. Le réseau de processus engendré par ce modèle peut donc être représenté par l'arbre ET/OU, en considérant que chaque nœud est un processus. Les processus ET assurent l'appariement des solutions trouvées par leurs processus fils de type ET, tandis que les processus OU collectent l'ensemble des solutions produites par leurs processus fils de type ET. Des algorithmes complexes permettent à l'exécution de déterminer l'ordre d'évaluation des sous-buts pour éliminer les problèmes liés aux variables partagées (liaisons conflictuelles).

D'autres modèles [Kal85, Kal87] ont également été proposés, dans le but de diminuer le coût des algorithmes de détection du parallélisme à l'exécution, par rapport au modèle proposé par Conery. Ces modèles sont fondés sur des techniques de détermination du parallélisme ET, à partir d'une analyse statique du programme.

Ces modèles ne semblent cependant pas exploitables efficacement sur les architectures parallèles existantes pour les raisons suivantes :

- Ils mettent en jeu un très grand nombre de processus, très supérieur au nombre de processeurs existants sur les architectures actuelles. Il est nécessaire alors de recourir à la multiprogrammation. Compte tenu des coûts d'installation et de gestion des processus (ordonnancement, synchronisation, changement de contexte) sur un nœud de calcul, il est préférable de privilégier la stratégie séquentielle à la stratégie parallèle.

- Ces processus échangent entre eux de nombreux messages, ce qui entraîne un surcoût très important dû aux communications.

b) Les modèles multi-séquentiels

A la différence des modèles théoriques qui cherchent à exploiter tout le parallélisme potentiel des programmes Prolog, les **modèles multi-séquentiels** ont été développés dans le but principal d'augmenter la vitesse d'exécution des programmes Prolog par rapport au modèle séquentiel, en tenant compte des caractéristiques des architectures existantes [Ali86].

Le principe de fonctionnement de ces modèles repose sur le contrôle des processus parallèles créés pour le traitement des branches de l'arbre ET/OU en fonction des ressources de calcul (processeurs, mémoire) disponibles. Un processus est considéré comme étant une instance d'exécution d'une machine Prolog. Le nombre de processus actifs à un instant donné est borné par le nombre de processeurs de la machine cible (parallélisme **paresseux**). La suite de nos travaux est consacrée à l'étude de cette classe de systèmes.

III.3 Les modèles multi-séquentiels

Plusieurs modèles multi-séquentiels ont été proposés. La plupart de ces modèles ont été développés pour une implantation sur une architecture particulière. Les techniques de mise en œuvre des systèmes multi-séquentiels diffèrent, suivant la source de parallélisme exploitée et le type de l'architecture cible. Nous présentons brièvement les problèmes liés à la mise en œuvre du parallélisme exploitant ces sources.

III.3.1 Le parallélisme OU

Rappelons que le parallélisme OU survient lorsqu'un sous-but peut être unifié à plus d'une clause. Dans ce cas, le corps de ces clauses peut être exécuté en parallèle.

Considérons l'exemple de programme et l'arbre d'exécution OU-parallèle correspondant de la figure III.6.

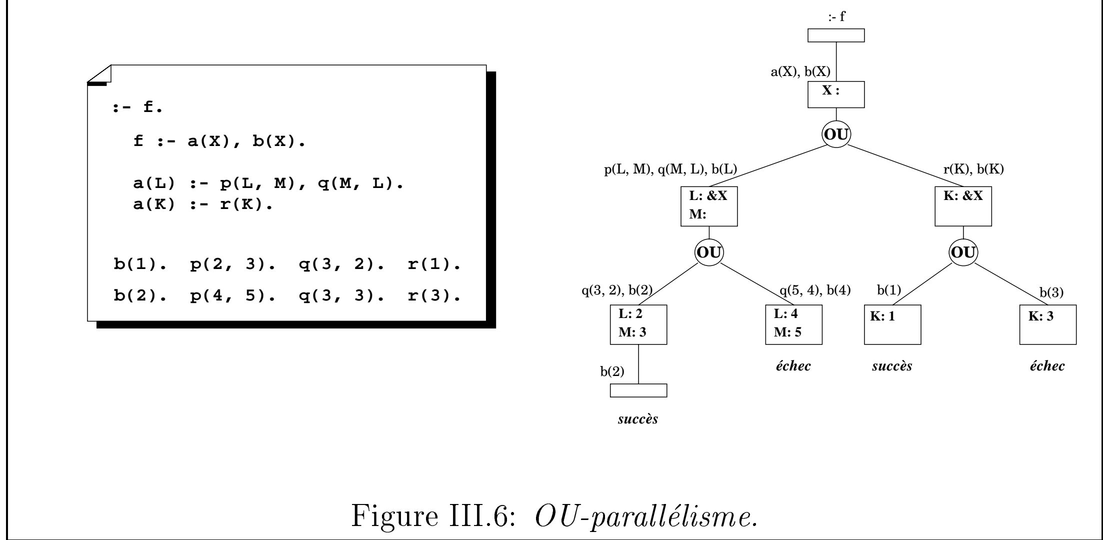


Figure III.6: *OU-parallélisme.*

Une résolvante différente est attribuée à chacune des activités créées après un nœud OU-parallèle. Dans l'exemple précédent, deux activités OU-parallèles sont créées pour résoudre en parallèle les résolvantes :

p(L, M), q(M, L), b(L). *et* *r(K), b(K).*

La variable partagée X (liée à L dans la première résolvante et à K dans la seconde) est instanciée différemment. Les environnements des deux résolvantes doivent être organisés de telle manière que les liaisons effectuées par chacune d'elles soient identifiables. L'un des principaux problèmes que pose l'implantation d'un système Prolog OU-parallèle est la gestion des résolvantes multiples.

Dans la plupart des implantations Prolog séquentielles, une résolvante est représentée par une pile d'environnements contenant des variables logiques liées entre elles lors d'unifications intervenues depuis le début de l'exécution du programme. A l'initialisation d'une nouvelle activité, il existe plusieurs possibilités pour construire la résolvante associée à l'activité, selon le type de machine ciblée (cf. section IV.4).

III.3.2 Le parallélisme ET

Si l'on veut paralléliser l'exécution d'un programme Prolog déterministe, il est nécessaire d'exploiter le parallélisme ET. La mise en œuvre du parallélisme ET pose cependant deux types de problème :

- le premier est lié au non-déterminisme des programmes *(retour-arrière)* : il s'agit de déterminer comment combiner les ensembles de solutions générées par l'exécution de chacun des sous-buts ;
- le second provient des variables communes aux sous-buts exécutés en parallèle : il s'agit là de concilier les liaisons engendrées à la même variable par les exécutions parallèles des sous-buts, et d'en assurer la cohérence d'accès.

L'une des solutions est d'exécuter indépendamment dans une première phase les sous-buts d'une clause et de s'assurer dans une seconde phase de la cohérence des variables partagées par une opération de jointure relationnelle. Cependant, le coût induit par la deuxième phase de calcul fait que cette approche reste pour l'instant peu réaliste. Il existe toutefois plusieurs solutions envisageables à partir d'hypothèses simplificatrices fondées sur des règles de synchronisation des sous-buts, lors de l'accès aux variables partagées. Nous présentons brièvement les différentes approches.

a) Parallélisme ET-Indépendant

Cette approche adoptée par plusieurs modèles de calcul [CK83, Lin88, Lin91, DeG84, Her86b] consiste à restreindre l'évaluation parallèle au seul cas où les

sous-buts sont indépendants, c'est à dire aux sous-buts n'ayant pas de variables en commun. Considérons par exemple le programme de la figure III.7 qui calcule la fonction de *Fibonacci*.

```
fib(0, 1).
fib(1, 1).
fib(N, M):- [ N1 is N-1, fib(N1, M1) ],        (1)
            [ N2 is N-2, fib(N2, M2) ],        (2)
            M is M1+M2.                        (3)
```

Figure III.7: *Parallélisme ET-Indépendant.*

Les deux listes de sous-buts (1) et (2) ne partagent pas de variables communes. Elles peuvent donc être traitées en parallèle. Mais le dernier sous-but (3) dépend des solutions générées par le traitement parallèle des deux listes précédentes. Son exécution ne peut être effectuée que si M_1 et M_2 ont été calculés.

Le parallélisme ET indépendant est simple mais la détection de cette indépendance n'est pas simple. Considérons par exemple la clause suivante :

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

A première vue, les sous-buts `r` et `s` sont indépendants dans cette clause ; ils pourraient donc être traités en parallèle. Cependant, il est possible que les variables `X` et `Y` soient interdépendantes par une liaison commune à une même autre variable. C'est le cas par exemple si l'appel de la clause est `p(Z, Z)`. Les deux variables `X` et `Y` deviennent dépendantes par la liaison effectuée à `Z`. Cet exemple montre qu'une analyse **syntaxique** n'est pas suffisante pour la détection des **liaisons conflictuelles**.

Plusieurs approches ont été proposées pour pallier à ce problème, depuis celles qui effectuent une analyse de programme à la compilation jusqu'à celles différant cette détection à l'exécution du programme. Un compromis doit être trouvé entre le taux de parallélisme exploité et le surcoût engendré par une analyse dynamique du programme. Quatre approches sont possibles :

- annotation du programme *(modage)* : une façon d'éviter d'avoir à détecter les dépendances de données est de demander au programmeur de spécifier le **mode** des variables pour chaque prédicat. Trois types d'annotations sont utilisées pour spécifier le mode d'une variable : $+, -, ?$. Une variable `X` d'un prédicat est dite *'entrante'* (notée $+$) si, à l'appel, cet argument est

toujours lié (clos). Une variable `Y` d'un prédicat est dite *'sortante'* (notée $-$) si, au succès de la résolution du prédicat, elle est toujours liée. Le symbole ? est utilisé pour les arguments dont on ne peut rien dire. Cette approche est adoptée dans l'implantation des langages logiques concurrents [Sha87, Sha89].

Des travaux récents montrent que ces annotations peuvent être générées automatiquement par un compilateur se fondant sur l'**interprétation abstraite** [CC77, CC92, MH92, JL92, DLH90, Deb92, DR94]. L'interprétation abstraite d'un programme consiste en une exécution symbolique où l'on ne s'intéresse pas à la valeur des termes calculés mais uniquement à des propriétés de ceux-ci, comme par exemple déterminer si certains termes sont interdépendants (partagent des variables) ou si certains termes sont clos.

- analyse **statique** : dans cette technique, les dépendances entre littéraux sont calculées statiquement par une analyse faite à la compilation [CDD85]. Dans le cas général, une telle analyse ne peut pas détecter le parallélisme maximal possible (car on doit choisir le cas le plus général d'instanciation des variables).

- analyse **dynamique** : cette approche consiste à détecter les dépendances des littéraux pendant l'exécution du programme. Les dépendances entre littéraux sont représentées par des graphes indiquant les relations de production/consommation des variables entre les différents sous-buts. Ces dépendances sont recalculées au fur et à mesure que l'on avance dans l'exécution. Cette approche est adoptée dans le modèle ET/OU de Conery [CK81, Con83, CK83, Con87b].

- analyse **mixte** : cette approche combine les avantages des deux précédentes. La phase d'analyse statistique est effectuée dans le but de transformer chaque clause en une expression conditionnelle exhibant le parallélisme possible. Cette expression est évaluée à l'exécution, pour tester la dépendance entre les variables. Cette approche est adoptée dans le modèle RAP *(Restricted AND-parallelism)* [DeG84] repris dans [Her86b].

b) Parallélisme ET-Dépendant

Deux sous-buts ayant des dépendances de données peuvent, sous certaines conditions, être exécutés en parallèle. L'approche essentiellement adoptée dans ce cas consiste à n'autoriser qu'un seul sous-but à lier une variable partagée (le producteur). Le second sous-but ne peut que consulter cette valeur (le consommateur). La variable est alors un canal de communication entre les deux sous-buts.

Ce type de parallélisme est aussi appelé **parallélisme de flot** *(Stream Parallelism)* [Wis86].

Le parallélisme de flot exploite les dépendances de données entre les buts d'une clause pour les lancer en parallèle. La synchronisation de l'ensemble s'effectue par l'indication du mode d'accès (lecture ou écriture) des arguments. Ces annotations contrôlent les canaux de communication entre les sous-buts d'une clause.

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

   p([a|Y]) :- r(Y).

   q([a|Y], Z) :- s(Z).
   q([b|Y], Z) :- t(Z).
```

Figure III.8: *Parallélisme de flot.*

Considérons par exemple le programme de la figure III.8. Dans le but initial, p et q ont en commun la même variable X. Le parallélisme de flot consiste à dire : si l'appel du prédicat p (le but producteur) lie toujours la variable X, alors le prédicat q (le but consommateur) peut a priori s'exécuter sans attendre que r soit terminé. Le problème consiste alors à contrôler l'exécution des sous-buts consommateurs. Il s'agit alors de :

- déterminer les instances *productrices* et *consommatrices* de chaque variable partagée[1],
- avoir des mécanismes permettant de suspendre et de réveiller un sous-but.

[She92a] est un système logique parallèle qui exploite cette caractéristique du parallélisme de flot.

Une communication bidirectionnelle est également possible entre deux sous-buts, via les dépendances communes. Considérons par exemple les deux buts suivants :

```
a(X), b(X).
```

[1] *Les techniques de détection des variables partagées, présentées dans la mise en œuvre du parallélisme* ET*-Indépendant sont également utilisées pour déterminer ces instances.*

Si le but `a` lie la variable `X` à la structure `f(Y)`, où `Y` est une variable non liée, cette liaison peut être consommée par le but `b` qui, à ce moment, lie la variable `Y`. Le but `a` peut alors consommer de nouveau cette liaison, créant ainsi un canal bidirectionnel entre les deux sous-buts. La mise en œuvre de cette approche est plus délicate que la précédente, du fait du non-déterminisme. Les langages gardés [Ued86, CG86, Sha89, FT90] contrôlent ce non-déterminisme par la sélection arbitraire d'un seul choix possible à un instant donné. D'autres approches sont également proposées dans [HJ90, CWY91, Bah91].

III.3.3 Parallélisme ET-OU combiné

La mise en œuvre d'un système exploitant simultanément les deux sources de parallélisme se trouve confrontée à la résolution conjointe des principaux problèmes rencontrés dans la mise en œuvre de chacun des systèmes exploitant une seule source de parallélisme.

Quelques systèmes proposent des limitations à l'exploitation de l'une ou l'autre source de parallélisme, de façon à simplifier le contrôle de l'exécution. Un exemple de ce type d'approche est donné par le modèle *ROPM* [Kal85] et le système *PEP-Sys* [Wes87, Bar88, Rob88, dK89]. D'autres modèles plus récents exploitant également cette approche sont proposés : le modèle *AO-WAM* [Gup91, GJ93b] et le modèle *ACE* [GH91, GP95, GHP95]. *IDIOM* [GSCYH91] est aussi un modèle combinant le OU-parallélisme et le ET-parallélisme indépendant, avec une extension au ET-parallélisme dépendant pour les buts déterministes.

Les simulations [SH91, She92b] effectuées sur l'exploitation simultanée de plusieurs sources de parallélisme montrent que de meilleures performances peuvent être obtenues. Cependant, l'implantation efficace de tels systèmes sur des architectures existantes reste un grand challenge.

III.3.4 Exploitation des effets de bord

La sémantique séquentielle de Prolog définit un ordre dans lequel les interactions avec le monde extérieur sont effectuées (entrées/sorties), et indirectement elle introduit également un ordre entre les sous-buts qui modifient dynamiquement le programme. Cette sémantique détermine un ordre strict dans l'évaluation des branches de l'arbre d'exécution.

Si l'exploitation des opérateurs *méta-logiques* ne pose pas de problèmes de mise en œuvre d'un système OU-parallèle, il n'en est pas de même pour les opérateurs *extra-logiques (entrées/sorties, assert/retract)* et les opérateurs de *contrôle (la coupure)*.

Le problème majeur qui se pose dans l'exploitation parallèle des prédicats *extra-logiques* réside dans la nécessité d'introduire des mécanismes de synchro-

nisation sévère entre les branches parallèles, dès lors que l'on veut conserver la sémantique originelle de Prolog (l'ordre d'exécution séquentiel). Cela se traduit souvent par une suspension du processeur qui exécute ce prédicat jusqu'à ce que celui-ci soit reconnu comme développant la branche active (branche en cours de résolution) la plus à gauche de l'arbre de résolution.

Les opérateurs de contrôle (la coupure) posent un problème de sémantique relatif à la stratégie parallèle d'évaluation. Une manière de faire est d'exécuter séquentiellement les branches issues d'un nœud OU contrôlé par une coupure. Cette solution est dite **inhibitrice** du parallélisme. Une autre approche consiste à poursuivre l'exécution parallèle d'un nœud OU de façon **spéculative**, c'est-à-dire que si une coupure est franchie, les calculs issus des branches *à droite* seront supprimés, sinon ils seront validés. La figure III.9 montre un exemple de ce phénomène.

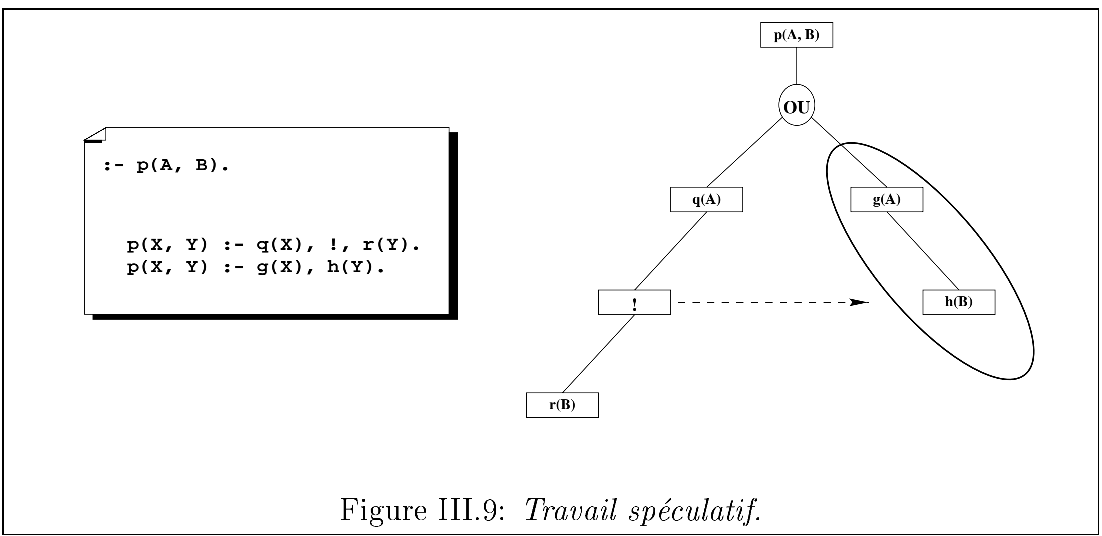


Figure III.9: *Travail spéculatif.*

Si l'on respecte la sémantique séquentielle de Prolog, l'exécution en parallèle des deux alternatives du prédicat $p/2$ engendre un travail spéculatif non productif si l'appel `q(X)` réussit.

D'autres approches ont introduit de nouvelles instructions au niveau de la WAM. Celles-ci permettent de diviser un groupe de clauses appartenant au même prédicat en sous-groupes, à l'intérieur desquels un processeur fonctionne en mode parallèle (sous-groupe ne contenant pas de coupure) ou en séquentiel (groupe de clause contenant au moins une coupure). Enfin, une autre alternative possible réside dans le fait de faire spécifier par le programmeur si l'on veut ou non exploiter le parallélisme OU (par annotations du programme), en fonction de l'absence ou non de prédicats à effets de bord dans certaines parties du programme.

Un autre problème majeur concerne l'interaction entre les différents effets de bord, avec notamment l'utilisation de la coupure, qui peut invalider l'existence

d'une branche de l'arbre. Cette branche peut être parcourue lors d'une exécution parallèle, mais elle n'aurait pas existé lors d'une exécution séquentielle. Aussi, les effets de bords doivent pouvoir être invalidés s'ils ont été effectués dans une telle branche. Le parcours d'une branche (alternative) dans la portée d'une coupure est dit spéculatif, car il peut être suspendu si la coupure est franchie. Si la coupure n'est pas franchie, cette branche devient valide et le travail correspondant n'est plus spéculatif. Pour gérer l'invalidation des effets de bord, il faut rendre ceux-ci réversibles, soit à l'aide d'un mécanisme de restauration de l'état antérieur, soit en utilisant un cache que l'on pourra détruire si l'effet de bord n'est plus utile. Le choix d'une solution d'implantation pour la prise en compte des prédicats à effets de bord dépend dans une grande mesure du type de l'architecture cible. Les solutions proposées dans les systèmes pour les architectures à mémoire partagée ne sont pas appropriées aux architectures sans mémoire commune [Ali90, AK90, Hau90, AKM91, Gup91, GSC92a, AK92b, Kar92]. En effet, celles-ci exigent le parcours de l'arbre de recherche afin de déterminer la branche la plus à gauche de l'arbre. Cette approche nécessite l'utilisation d'un grand nombre de messages sur une architecture sans mémoire commune. Les problèmes liés à l'exploitation efficace des effets de bord dans le cadre de la mise en œuvre d'un environnement de programmation Prolog parallèle sur des architectures sans mémoire commune font l'objet d'une présentation spécifique dans [Mor96].

III.4 Le problème d'efficacité

La première raison qui pousse les informaticiens vers le parallélisme est la puissance de calcul que recèle un ordinateur parallèle. L'idée de base est d'utiliser les multiples ressources matérielles (les processeurs de calcul) pour diviser d'autant le temps d'exécution. L'exploitation du parallélisme en programmation logique rejoint également cet objectif. Il s'agit alors d'apporter un gain de performance en terme de temps d'exécution des programmes, par rapport aux implantations séquentielles. La finalité est d'obtenir un facteur de gain linéaire (on va N fois plus vite on utilisant N processeurs). Atteindre ce but est un problème complexe qui dépend tout d'abord du programme à exécuter, mais aussi et surtout de l'architecture cible.

Le contrôle du parallélisme consiste principalement à gérer l'allocation des tâches aux processeurs. C'est un problème critique car le coût d'installation d'une tâche sur un processeur peut être tel qu'il annule le gain obtenu par parallélisation. Il est donc nécessaire de limiter, voire éliminer (quand c'est possible) ce cas défavorable, par une stratégie de contrôle appropriée de la répartition des tâches sur les processeurs.

III.4.1 Notion de tâche et de Granularité

On désigne par le terme **tâche** le traitement d'une séquence d'étapes de résolution (inférences logiques) effectuées par le moteur d'inférence (machine) Prolog. Il résulte de cette définition qu'une tâche n'est pas une unité de traitement statique pouvant être définie à la compilation. Une tâche est définie (créée) dynamiquement durant l'exécution. C'est une entité représentant un traitement dont le coût désigne sa **granularité**.

On définit la granularité, ou grain d'une tâche, par le temps nécessaire pour son exécution. La notion de tâche correspond à une séquence ininterrompue de calcul jusqu'à terminaison et/ou création éventuelle d'autres tâches.

Selon les formes de parallélisation exploitées, on a l'habitude de distinguer trois types de granularité en programmation logique parallèle :

- le gros grain correspond à l'exploitation du parallélisme OU de résolvante[2] (OU multi-séquentiel),
- le grain moyen correspondant à l'exploitation du parallélisme ET (multi-séquentiel) qui est un parallélisme de sous-arbre,
- le grain fin correspond à l'exploitation du parallélisme d'unification des termes d'un littéral.

Pour tous les modes de parallélisation, la construction récursive des prédicats fait qu'une exécution séquentielle d'un prédicat peut se découper en autant d'exécutions parallèles que le type de parallélisation le permet. Le degré de découpe, c'est à dire le choix d'un grain, est généralement conditionné par deux facteurs :

- le taux de déperdition *(overhead)* provoqué par la gestion des tâches,
- la flexibilité que l'on veut obtenir lors de l'équilibrage de charge.

Gestion des tâches

Le partage du travail entre processeurs induit inévitablement un coût de communication qui n'existe pas en séquentiel. La déperdition engendrée par cette gestion dépend des choix d'implantation et des caractéristiques de la machine cible. En effet cette déperdition, sur une architecture à mémoire partagée, dépend du coût nécessaire pour accéder aux données, alors que pour une architecture sans mémoire commune elle est généralement provoquée par les communications nécessaires pour le transfert des données entre processeurs (l'envoi et la réception de messages).

[2] *Le parallélisme* OU *du modèle théorique est un parallélisme à grain fin.*

Dans une architecture sans mémoire commune, un grain fin de parallélisme correspond à un degré ou **taux de parallélisme** élevé et induit généralement beaucoup de communications, alors qu'un gros grain de parallélisme exploite la **localité** des traitements pour réduire le taux de communication au détriment du degré de parallélisme. Un compromis reste donc à trouver entre l'exploitation d'un degré de parallélisme élevé d'une part, et la localité présente dans les traitements d'autre part, ceci afin de limiter la déperdition de gestion des tâches.

Flexibilité de l'équilibrage

Le découpage des tâches en grain de plus en plus petit a pour effet d'accroître leur nombre et d'homogénéiser leur taille. Ceci facilite leur répartition uniforme sur les processeurs, de manière à obtenir un bon équilibrage de la charge de travail (calcul). Par contre, si la granularité des tâches est trop petite par rapport au temps nécessaire pour leur installation, les performances peuvent chuter. Il est primordial d'adapter la granularité des tâches à l'architecture de la machine cible. Dans le cas de tâches non-homogènes, l'équilibrage peut être difficile à réaliser.

III.4.2 Régulation de charge

L'exploitation du parallélisme en programmation logique induit des surcoûts. Ces surcoûts, qui sont généralement absents en séquentiel, proviennent des coûts de gestion et des synchronisations correspondants :

- au partage du travail,
- à l'installation des tâches.

Ces surcoûts engendrent une augmentation des temps de calcul et/ou de l'espace mémoire utilisé. Il convient donc de bien les étudier et de les prendre en compte lors de l'implantation.

Le partage du travail est géré par une fonction de régulation de charge qui assigne des tâches aux processeurs. Ce processus consiste à :

- regrouper des étapes de résolution sous forme de tâches,
- rechercher les tâches et les processeurs disponibles,
- allouer des tâches (du travail) aux processeurs.

L'objectif de la fonction de régulation de charge est de répartir équitablement et le plus rapidement possible la charge de travail (tâches) sur les processeurs, de façon à optimiser le temps d'exécution du programme. Comme dans toute autre

implantation de systèmes parallèles, l'efficacité de cette fonction a un impact critique sur les performances globales du système.

Contrairement à certaines applications parallèles, l'élaboration d'une fonction de régulation de charge pour un système logique parallèle nécessite la prise en compte des caractéristiques particulières suivantes :

- Le caractère hautement dynamique de l'évaluation des programmes ne permet pas de partitionner *a priori* l'exécution pour un placement statique sur les processeurs ;
- La granularité des tâches peut être très petite ;
- La structure de l'arborescence développée peut être irrégulière et diffère d'un programme à un autre.

L'efficacité de la fonction de régulation de charge dépend du couple stratégie de régulation- efficacité d'installation des tâches. La stratégie de régulation définit le grain et le degré du parallélisme exploité.

Le coût C_{reg} induit par une fonction de régulation pour la gestion du partage de la charge de travail sur les processeurs dépend :

- du temps s_i nécessaire pour sélectionner (trouver) une tâche i,
- du temps d'installation T_i des données nécessaires à son exécution,
- du nombre de tâches installées M.

$$C_{reg} = \sum_{i=1}^{M}(s_i + T_i) \qquad \text{(III.1)}$$

Minimiser le surcoût C_{reg} revient à minimiser les s_i, T_i et M. Notons également que le nombre d'installations M est fortement lié à la granularité des tâches d'un programme donné. Augmenter la granularité des tâches a pour conséquence de réduire ce nombre.

Le temps (délai) d'acquisition T_{acquis} d'une tâche T_i peut être décomposé en deux parties : le temps d'accès aux données (gestion des structures de données, protocoles de communication) $T_{accès}$, et le temps d'installation des données T_{inst}, dépendant de l'architecture cible.

$$T_{acquis} = T_{accès} + T_{inst}$$

Pour une architecture sans mémoire commune, il est possible de diminuer le temps T_i à travers de celui de $T_{accès}$, en améliorant les structures de données et

les protocoles de communication. En revanche, le seul moyen de diminuer T_{inst} est d'améliorer le support de communication, le débit du réseau par exemple. De ce fait, les machines à mémoire physiquement partagée, avec leur temps de transmission faible, sont mieux adaptées à une granularité de travail relativement petite. Par contre, celles avec une mémoire physiquement distribuée doivent entraîner le choix d'une granularité plus importante.

Trois autres éléments sont déterminants dans l'élaboration d'une fonction de régulation de charge pour un système logique parallèle :

- l'implantation de la machine Prolog et plus particulièrement le modèle de gestion mémoire,
- la stratégie de contrôle de la granularité,
- la gestion des effets de bord.

a) Gestion mémoire

Le modèle de gestion mémoire utilisé par le moteur de résolution est caractérisé par les structures de données manipulées lors des liaisons de variables. La complexité d'implantation de ces structures de données et des liaisons détermine le coût de manipulation des variables lors de l'initialisation et de l'exécution de nouvelles tâches.

Dans les systèmes Prolog séquentiels, la mémoire est efficacement utilisée, du fait du parcours en profondeur d'abord de l'arbre de recherche. A un instant donné, une seule branche réside dans les piles de la WAM *(Warren Abstract Machine)*. Les deux règles suivantes sont toujours respectées dans les implantations séquentielles traditionnelles :

- si N_1 est un nœud appartenant à une branche située à droite d'une branche contenant le nœud N_2, les structures de données correspondantes au nœud N_2 sont désallouées avant celles du nœud N_1;
- si le nœud N_1 est l'ancêtre du nœud N_2 dans l'arbre d'exécution, les structures de données du nœud N_2 sont désallouées avant celles de N_1.

La conséquence de ces deux règles est que l'espace mémoire est désalloué à partir du sommet de pile, lors d'une opération de retour-arrière. Cependant l'introduction du parallélisme dans une implantation séquentielle introduit des modifications dans la gestion des piles de la WAM, rendant difficile la préservation de ces deux règles. L'exploitation du parallélisme OU rend la première règle difficile à assurer, alors que l'exploitation du parallélisme ET permet difficilement d'assurer la seconde. La rupture de la stratégie LIFO *(Last In First Out)* de la

gestion de pile a pour principale conséquence l'augmentation de l'espace mémoire utilisé.

Le modèle de gestion mémoire a un impact non-négligeable également sur les coûts de sélection et d'initialisation d'une nouvelle tâche. Gupta et Jarayaman [Gup91, GJ93a] définissent deux paramètres élémentaires pour l'analyse du coût d'initialisation d'une tâche dans un modèle de gestion mémoire :

- le coût de liaison/déliaison des variables β. Il dépend de la complexité des structures de données manipulées ($T_{accès}$),
- le coût de création T_{inst} (installation) de l'environnement d'exécution de la tâche. Celui-ci dépend du nombre v de variables à mettre à jour et du mécanisme mis en œuvre pour la reconstruction du contexte d'exécution. Il dépend de l'architecture cible (cf. section IV.4).

Le coût d'initialisation C_{init} d'une tâche i dépend de ces deux paramètres :

$$C_{init}(i) = T_{inst}(i) + \beta.v(i) \quad \text{(III.2)}$$

Le coût de reconstruction du contexte de la tâche n'étant pas constant mais fonction du nombre de variables v à mettre à jour, le problème de minimiser le surcoût d'initialisation d'une tâche vient s'ajouter à la fonction de régulation de charge. Parmi toutes les tâches disponibles, la fonction de régulation doit choisir celle qui minimise ce coût. Ce critère n'est pas compatible avec celui énoncé plus haut, qui stipule que le temps de sélection d'une tâche doit également être minimisé.

b) Contrôle du grain de parallélisme

Le contrôle du grain du parallélisme a pour but de garantir que l'accroissement du parallélisme ne conduise pas à une baisse d'efficacité. En effet :

- le partage de la charge de travail à un instant donné de l'exécution n'est bénéfique que si le coût de traitement de la tâche recouvre au moins son coût d'installation ;
- les ressources de calcul étant limitées et les coûts d'installation des tâches non-négligeables sur une architecture parallèle donnée, le nombre M de tâches parallèles doit être contrôlé pour ne pas provoquer une chute de performance.

Si l'on considère que l'on a, à un instant t donné, M instances de tâches de taille T_g et un temps résiduel d'exécution T_1, nous aurons alors un temps d'exécution séquentiel équivalent à :

$$T_{seq} = T_1 + M \times T_g \tag{III.3}$$

alors que pour une exécution parallèle optimale on aurait :

$$T_{par} = T_1 + T_g + M \times C \tag{III.4}$$

C correspond au coût d'installation d'une tâche sur un processeur. Garantir une exécution parallèle efficace signifie que l'exécution parallèle doit être plus rapide qu'une exécution séquentielle. Ce qui revient à avoir l'inégalité suivante : $T_{seq} \geq T_{par}$ (ou $\frac{T_{seq}}{T_{par}} \geq 1$). On obtient alors :

$$T_g > \frac{M}{M-1} \times C \tag{III.5}$$

T_g dans l'équation III.5 nous donne un ordre de grandeur de ce que doit être la granularité des tâches parallèles si l'on veut garantir un accroissement de performance par rapport à une exécution séquentielle[3].

Garantir que l'initialisation d'une tâche parallèle produise un accroissement de performance suppose que l'on puisse estimer d'une manière raisonnablement précise le temps de traitement de cette tâche. Or, du fait de la nature dynamique de l'exécution du programme Prolog, le temps d'évaluation d'un but est complexe, voire imprévisible dans le cas général. Le calculer exactement nécessite une exécution complète. Nous reviendrons plus en détail sur les différentes approches possibles pour l'estimation du poids d'une tâche dans les prochains chapitres.

c) Impact des effets de bord

La gestion des effets de bord, avec le respect de la sémantique séquentielle de Prolog, impose des contraintes supplémentaires au niveau de la fonction de régulation de charge. Les techniques proposées pour une implantation sur une architecture à mémoire partagée ne sont généralement pas exploitables (efficacement) sur des architectures sans mémoire commune, du fait que l'on ne dispose pas d'une représentation globale de l'arbre OU courant. Nous présentons brièvement les principales solutions pour des machines sans mémoire commune. Le lecteur intéressé par une description détaillée peut se reporter à la thèse [Hau90].

[3] *Si l'on considère que seules $(M-1)$ tâches sont lancées en parallèle, le temps d'exécution parallèle optimal serait : $T_{par} = T_1 + T_g + (M-1) \times C$. Le grain minimum exploitable T_g doit être, dans ce cas, supérieur au coût C d'installation d'une tâche ($T_g > C$).*

La coupure
il existe plusieurs possibilités pour la prise en compte du travail spéculatif lors de la régulation de charge [Hau90]. La première est de ne pas distinguer le caractère spéculatif. La gestion est simplifiée, mais le risque d'inefficacité est grand, car beaucoup de processeurs peuvent alors consommer du travail spéculatif qui sera ensuite détruit, et donc ne participeront pas à l'accroissement des performances. La deuxième possibilité est de différencier le poids du travail spéculatif 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 de favoriser, le reste du temps, le travail sûr. Enfin, une autre possibilité consiste à favoriser l'allocation de travail réduisant le caractère spéculatif d'un sous-arbre, ce qui conduit au choix de travail principalement en partie gauche de l'arbre d'exécution.

Les autres effets de bord
Les prédicats de modification de la base des prédicats ne peuvent être implantés de manière implicite et efficace sans suspension. 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. La modification de la base des prédicats impose des contraintes fortes de synchronisation, aussi faut-il limiter au maximum leur influence, en favorisant le travail en partie gauche de l'arbre.

Pour les prédicats d'*entrées/sorties*, leur gestion se traduit souvent par une suspension du processeur qui exécute ce prédicat, jusqu'à ce que celui-ci soit reconnu comme développant la branche la plus à gauche de l'arbre de résolution.

III.5 Conclusions

La sémantique des langages de programmation logique les rend particulièrement bien adaptés aux implantations parallèles. Les systèmes de programmation logiques parallèles se divisent en deux grandes familles : la première est orientée vers l'introduction de mécanismes pour exprimer le parallélisme : c'est l'approche des langages logiques gardés. La seconde caractérise les systèmes non-déterministes dont l'implantation vise une compatibilité avec les systèmes Prolog séquentiels. Dans cette thèse nous nous intéressons à cette dernière famille, et plus particulièrement aux systèmes multi-séquentiels, pour les raisons suivantes :

- l'approche multi-séquentielle vise à l'exploitation de la puissance de calcul des machines parallèles en masquant à l'utilisateur la complexité de gestion du parallélisme,
- les modèles multi-séquentiels sont plus réalistes que les modèles théoriques, principalement si l'on considère les caractéristiques des machines parallèles

actuelles,

- les modèles multi-séquentiels visent à tirer partie des techniques les plus efficaces d'implantation séquentielle,

- les premiers résultats obtenus par les prototypes d'implantation indiquent que ces systèmes permettent d'obtenir d'importants gains de performance par rapport aux systèmes Prolog séquentiels.

Plusieurs systèmes multi-séquentiels sont proposés dans la littérature. Ils diffèrent entre eux par la caractéristique du langage qu'ils supportent, la source de parallélisme exploitée, l'architecture cible et leurs mécanismes de mise en œuvre du parallélisme. Dans le cadre de la conception d'un système logique parallèle, nous avons choisi de ne prendre en compte que le parallélisme OU inhérent à la sémantique des programmes Prolog. Ce choix est motivé par les constatations suivantes :

- le parallélisme ET est plus difficile à mettre en œuvre que le parallélisme OU,

- le grain du parallélisme ET est plus fin que celui du parallélisme OU,

- l'exécution OU parallèle correspond à l'évaluation simultanée d'alternatives indépendantes,

- le parallélisme OU est exploitable pour de grandes classes de problèmes qui font appel au non-déterminisme. Pour ces problèmes, les méthodes de résolution passent par l'exploration d'un espace de recherche qui, même si elle ne produit que peu de solutions, offre un potentiel important d'exploitation du parallélisme OU.

Chapitre IV

Régulation de charge dans les systèmes OU multi-séquentiels

IV.1 Introduction

Un des points essentiels pour bien utiliser une architecture parallèle est de trouver un bon **placement** des composants de l'application sur les nœuds (processeurs) de la machine. Bien qu'il soit parfois possible de trouver une répartition de charge adéquate par l'utilisation de techniques de placement **statique**, celle-ci dépend beaucoup des caractéristiques de l'application et de l'architecture de la machine cible [Bok81, MT91, CT93].

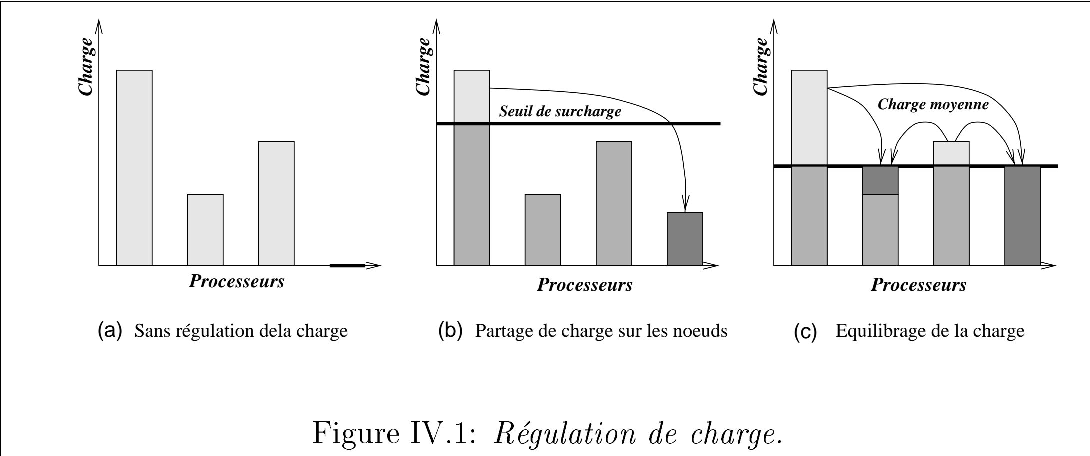
L'élaboration d'une fonction de régulation de charge pour un système logique parallèle nécessite de tenir compte du caractère hautement dynamique de l'évaluation des programmes, qui ne permet pas de partitionner a priori l'exécution pour un placement statique sur les processeurs. Le problème du placement devient alors plus difficile car il faut le traiter **dynamiquement** au moment même de l'exécution. Une stratégie de régulation pour une machine cible donnée doit tenir compte également du grain minimum de parallélisme exploitable en dessous duquel un écroulement de performance peut se produire. Une bonne répartition de charge dépend de la connaissance qu'a la fonction de régulation du grain des opportunités de parallélisation rencontrées. Or, du fait de la nature dynamique de l'exécution d'un programme Prolog, il est très difficile de déterminer le grain de ces opportunités à la compilation. Ceci impose donc l'utilisation de techniques dynamiques (heuristiques) de prévision de coût pour choisir, à l'exécution, celles des opportunités conduisant effectivement à un gain de performance.

Nous allons, dans un premier temps, présenter les fonctionnalités nécessaires à mettre en œuvre pour l'élaboration d'une fonction de régulation dans un système parallèle, ainsi que l'impact de l'architecture dans les choix d'implantation. La

deuxième partie de ce chapitre est consacrée à l'étude spécifique du problème de régulation de charge pour les systèmes logique OU-parallèles.

IV.2 Régulation de charge

Une bonne utilisation des machines parallèles nécessite un découpage de la charge que l'application va produire au cours de son exécution, de façon à pouvoir la distribuer sur chaque processeur de l'architecture. La charge se définit en fonction de la nature de l'application. Certaines applications s'organisent autour d'un traitement systématique à effectuer sur un volume de données important. Le découpage et le placement de ces données sont utilisés comme un moyen de distribution de la charge. Les exemples que l'on peut donner de ce type d'applications sont ceux du découpage de matrices dans le calcul numérique, ou encore de celui des données d'une scène en synthèse d'images [GO93, CT93]. D'autres applications manipulent des structures de données complexes et irrégulières qui demandent un traitement plus lourd. L'application est perçue comme composée d'entités (activités) élémentaires, que l'on désigne souvent sous le terme de **tâches** ou **processus** [Gos91], qui s'échangent les données dont elles ont besoin pour fonctionner. La stratégie utilisée pour le placement de ces entités a un impact direct sur les performances d'exécution de l'application. Il est de ce fait très important, pour des raisons d'**efficacité**, de pouvoir gérer l'allocation de la charge produite par l'exécution d'une application sur les différents processeurs d'une architecture : c'est le problème de la régulation de charge.



(a) Sans régulation dela charge (b) Partage de charge sur les noeuds (c) Equilibrage de la charge

Figure IV.1: *Régulation de charge.*

Ce problème a toujours suscité un très vif intérêt dans différents domaines de recherche. De nombreuses définitions et taxonomies de modèles de régulation de charge ont été proposées dans la littérature [Cas81, CK88, Gos91, SKS92, Her94].

La régulation de charge est également désignée sous les termes : l'équilibrage de charge, le partage de charge ou encore l'ordonnancement distribué [Gos91].

Nous adopterons la définition informelle suivante [Mil93] : *la régulation de charge désigne toutes les techniques utilisées pour le transfert d'une charge de travail (de calcul) entre les processeurs d'un système, afin de satisfaire un critère d'efficacité.*

Les algorithmes de régulation de charge se distinguent par leurs objectifs. Ils peuvent être utilisés dans un système pour répondre à différents critères d'efficacité. Leur utilisation la plus courante consiste à optimiser l'**utilisation des ressources** de calcul (partage de charge) [WM85] et/ou le **temps de réponse** (équilibrage de charge) [ZF88, SKS92] d'une application (cf. figure IV.1). Cet objectif peut être complété par des contraintes temporelles plus strictes dans le cadre d'applications temps réels [CL86, KC87, Sta89, Bla92] ou des contraintes de **tolérance aux pannes** [BT83, Abr83, GGLS91, KL94] qui modifient le comportement du régulateur de charge.

On distingue deux aspects dans l'élaboration d'une fonction de régulation de charge :

- les fonctionnalités à mettre en œuvre pour gérer le partage de la charge de calcul entre les processeurs,
- l'organisation (l'implantation) de cette fonction de régulation sur une architecture parallèle donnée.

IV.2.1 Composants d'une fonction de régulation de charge

De nombreux travaux ont été consacrés à l'étude et la réalisation de fonctions de régulation dynamique de charge appropriées aux systèmes parallèles. La plupart de ces travaux s'inspirent, du moins d'un point de vue conceptuel, de ceux réalisés dans le contexte des systèmes distribués, dits de large granularité, où un nombre de processeurs communiquent à travers un réseau local (Ethernet, Token Ring, FDDI) [WM85, ZF88, SKS92], ou dans des systèmes temps réels [CL86, KC87, Sta89].

L'élaboration d'une fonction de régulation dynamique de charge pour un système parallèle requiert, de manière générale, la mise en œuvre des fonctionnalités suivantes (cf. figure IV.2) :

- un mécanisme de transfert des calculs entre les processeurs de l'architecture,
- un gestionnaire de l'état de charge du système,
- un algorithme de régulation de charge.

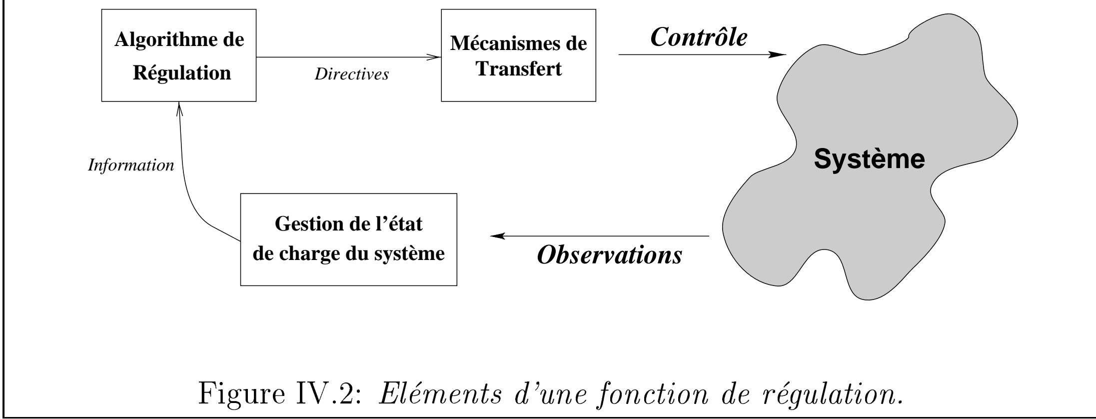


Figure IV.2: *Eléments d'une fonction de régulation.*

Mécanismes de transfert

Le transfert d'un calcul d'un processeur du système à un autre consiste à extraire son état *(contexte d'exécution)* sur son nœud d'origine, afin de le transférer et de l'installer sur le nœud cible du transfert.

Gestionnaire de l'état de charge

Il a pour objectif le maintien de l'information concernant l'état de charge du système, à partir duquel l'algorithme de régulation prendra des décisions d'équilibrage de charge. Il s'appuie pour cela sur :

- l'évaluation (instrumentation) d'un *indice* de charge : l'indice de charge est choisi de manière à ce que sa valeur soit corrélée avec le taux d'utilisation des ressources du processeur (CPU, mémoire) ;
- des mécanismes de *dissémination* de l'information de charge : ils sont utilisés pour la collecte et l'échange de l'information de charge entre les processeurs du système. La collecte de l'information de charge peut être effectuée selon différentes approches :
  - *échange périodique* : les processeurs échangent régulièrement les valeurs de leur indice de charge,
  - *échange sur un changement d'état* : chaque processeur ne transmet sa nouvelle charge que si une transition d'état est survenue.

Algorithme de régulation

C'est l'élément décisionnel d'une fonction de régulation. Il établit des directives de régulation à partir de l'état de charge des processeurs maintenu par le gestionnaire et de la stratégie de régulation qu'il met en œuvre. Le choix d'une stratégie est déterminé à partir des caractéristiques de l'architecture cible et du

critère d'efficacité poursuivi par le système. L'algorithme de régulation décide de l'instant où il est nécessaire d'effectuer une régulation de charge, de la quantité de charge (calcul) à transférer et de l'appariement des processeurs source et cible du transfert (cf. figure IV.3).

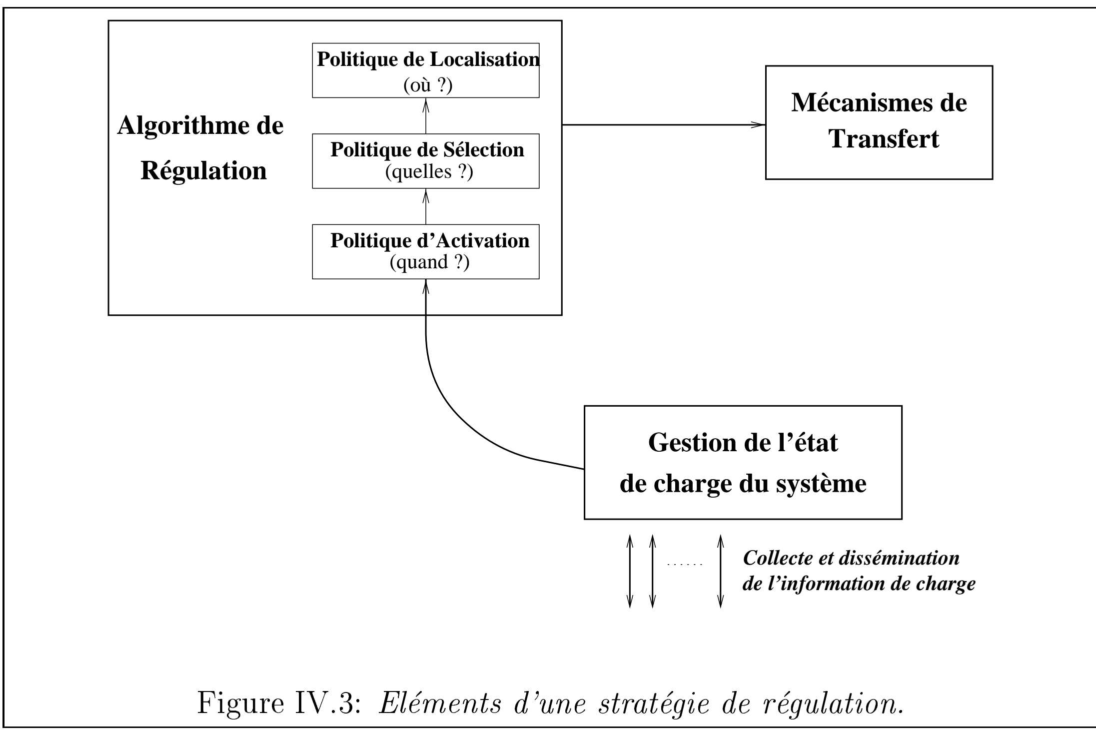


Figure IV.3: *Eléments d'une stratégie de régulation.*

Une stratégie de régulation est caractérisée par :

- une politique d'*activation* : elle exprime la règle de déclenchement d'une régulation. Différentes approches sont possibles :
  - à partir de transitions d'état de charges locales,
  - sur des transitions d'état de charge globale,
  - ou une activation de manière périodique.
- une politique de *sélection* : elle détermine la quantité de charge à transférer. Deux facteurs sont à considérer à ce niveau :
  - les caractéristiques de l'architecture cible (interconnexion des processeurs, protocole de communication, débit du réseau de communication) qui déterminent le coût induit par une opération de transfert de charge,
  - la granularité minimale exploitable en dessous de laquelle un écroulement de performance se produit.

- une politique de *localisation* : elle détermine le processeur cible du transfert. Cette localisation peut être :
  - aléatoire,
  - probabiliste,
  - ou sélective pour tenir compte, pour des raisons d'efficacité, des caractéristiques de l'architecture cible et des coût de transfert induits par la sélection d'un processeur.

IV.2.2 Architecture d'une fonction de régulation

Nous rappelons que le support de communication utilisé pour l'échange de données entre les processeurs permet de distinguer deux classes de machines MIMD : les machines parallèles à mémoire commune et celles à mémoire distribuée. L'espace mémoire adressable dans les machines à mémoire commune est le même pour chaque processeur. Dans les machines sans mémoire commune, l'espace mémoire adressable est local à chaque processeur. Les communications entre processeurs se font par échanges de messages.

La mise en œuvre d'un algorithme de régulation de charge pour un système parallèle doit tenir compte des caractéristiques essentielles de la machine cible pour être conçu et utilisé efficacement.

a) Gestion de l'état de charge du système

L'état de charge **global** $E = (e_1, e_2, ..., e_n)$ d'un système parallèle est constitué d'une collection d'états locaux e_i ($i \in [1...n]$) des n processeurs. Les décisions de régulation de charge supposent de disposer de cet état global lors de la prise de décision. Si les machines à mémoire commune facilitent le maintien d'un état global **exact** accessible par tous les processeurs, il n'en est pas de même pour les machines sans mémoire commune. Pour celles-ci, l'état global est distribué sur l'ensemble des processeurs. L'estimation de l'état du système ne peut être qu'une approximation, et ce, pour des raisons d'efficacité.

En effet, estimer précisément la charge des processeurs à un instant donné nécessiterait une synchronisation globale coûteuse qui ne pourrait que ralentir de façon importante l'exécution du programme parallèle considéré. Pour des machines sans mémoire commune, les décisions de régulation ne peuvent être effectuées que sur un état **approché** de l'état global du système. Cet état est obtenu par échantillonnage des états locaux sur chaque processeur. Ceci complique sérieusement les algorithmes de régulation de charge par rapport à ceux réalisés sur une machine parallèle comportant une mémoire partagée.

Plusieurs organisations de cette fonction de régulation sont possibles :

- centralisée,
- hiérarchisée,
- ou complètement distribuée sur les processeurs.

Dans cette dernière approche, chaque processeur maintient un état de charge approché du système ; de même que chacun exécute le même algorithme de régulation de charge. Les décisions de régulation prises par un processeur sont indépendantes des décisions prises par les autres processeurs. Par conséquent, le problème à résoudre consiste à garantir que ces décisions ne sont jamais contradictoires. Pour ne pas avoir de solutions contradictoires, il faudrait que les processeurs susceptibles d'entrer en conflit se synchronisent pour vérifier l'absence de contradiction. Cette solution n'est pas simple à mettre en œuvre et nécessite une synchronisation coûteuse entre les processeurs susceptibles d'entrer en conflit.

Dans une organisation centralisée, un seul processeur est dédié à la gestion et au contrôle du parallélisme. Chaque processeur transmet ses états à ce processeur qui maintient l'état de charge approché du système et effectue les opérations de régulation de charge. L'avantage de cette solution est la simplification de la gestion de l'état global de l'algorithme de décision. L'algorithme de régulation ne prend qu'une seule décision à la fois. L'inconvénient majeur de cette solution est le goulot d'étranglement potentiel que constitue l'unique processeur de contrôle : son temps de service augmente avec le nombre de processeurs à gérer.

Une solution intermédiaire consiste à définir un contrôle hiérarchique. Dans cette approche, un premier niveau de processeurs se partagent le contrôle des autres processeurs. Les processeurs d'un niveau donné pilotent des groupes (grappes) de processeurs du niveau inférieur. Le dernier niveau est le processeur maître. Les échanges entre grappes d'un même niveau sont gérés de manière centralisée par le supérieur hiérarchique des processeurs de contrôle des grappes de ce niveau [GG90].

b) Transfert de charge

La façon dont le transfert de charge entre processeurs est effectué dépend de l'architecture cible. Dans une machine à mémoire partagée, il suffit de transférer le descripteur du processus, d'une file d'un processeur à celle d'un autre. Dans le cas d'architectures sans mémoire commune, l'opération est beaucoup plus complexe et implique un échange important d'informations pour déplacer le contexte d'une tâche d'un processeur à un autre. Les méthodes de transfert sont variées [Roz88, DO91, Gos91, BSS91, Tan92, Mil93] :

- *Substitution* d'une création locale de processus par une création à distance : La charge qui aurait résultée de l'exécution de ces processus localement

est placée sur le nœud sélectionné. Le transfert de charge est assimilé à un appel de procédure à distance sur le nœud cible. Les paramètres de l'appel représentent le contexte d'exécution nécessaire à l'initialisation du traitement sur le processeur cible qui détient déjà le code du programme à exécuter.

- *Migration* de processus : Le transfert de charge consiste à extraire le contexte d'exécution *(état mémoire, état cpu)* de certains processus, à les déplacer et à les restaurer sur le nœud sélectionné.

Il est plus facile de déplacer une tâche avant son démarrage (création de processus à distance) qu'au cours de son exécution (migration) [Gos91, Tan92]. Le coût d'installation est fortement dépendant de l'architecture matérielle, en particulier du médium de communication employé et du volume de données à installer, ainsi que du protocole de communication utilisé pour l'acheminement des données.

c) Stratégie de régulation

La stratégie de régulation détermine la quantité de charge à transférer et le processeur cible du transfert. La quantité de charge doit être telle qu'elle compense le surcoût occasionné par son installation. Le coût d'installation dépend des caractéristiques de l'architecture cible [YS89] (interconnexion des processeurs, protocole de communication, débit du réseau de communication). Il est généralement plus important sur architecture sans mémoire commune que sur une machine à mémoire partagée.

Pour une architecture sans mémoire commune, le coût de transfert s'exprime généralement comme étant la somme d'une constante β qui représente le surcoût du processeur pour l'amorçage du transfert et d'une fonction croissante du volume (IV.1). Cette fonction est paramétrée par le débit des liens de communication τ_c. Ce paramètre est une caractéristique architecturale de la machine, son inverse $\frac{1}{\tau_c}$ représente la bande passante des liens de communication [YS89]. Cette estimation du coût de transfert est à raffiner en fonction des caractéristiques du réseau de communication et selon le mode de routage utilisé.

$$co\hat{u}t_transfert = \beta + L\tau_c \tag{IV.1}$$

Le coût induit par une opération de transfert de charge influe sur la stratégie de sélection du processeur cible du transfert. La topologie du réseau de communication constitue une donnée importante de l'architecture de la machine à prendre en compte. Elle est définie par :

- le nombre P de nœuds (processeurs) de la machine parallèle ;

- la distance d_{ij} entre chaque paire de processeurs $(i, j) \in P$: elle correspond au plus court chemin de liens (processeurs) physiques les reliant ;
- le diamètre D qui correspond à la distance maximale entre toute paire de processeurs, $D = max(d_{ij}) \ / \ \forall (i, j) \in P$;
- le degré du graphe d'interconnexion qui indique le nombre de voisins immédiats de chaque processeur.

De nombreuses taxonomies de stratégies de régulation de charge ont été proposées dans la littérature [Cas81, LK87, CK88, XH90, GG90, Sal90, Gos91, SKS92]. Une des classifications les plus utilisées est fondée sur le processeur instigateur d'une opération de régulation. On a alors trois grandes classes de stratégies :

- stratégies à l'**initiative de l'envoyeur** *(Sender Initiated Algorithm)* : ce sont les processeurs surchargés qui déclenchent l'opération de régulation de charge (le transfert) ;
- stratégies à l'**initiative du receveur** *(Sender Initiated Algorithm)* : ce sont les processeurs sous-chargés qui déclenchent l'opération de régulation de charge.
- stratégies **mixtes** *(Symmetrical Initiated Algorithm)* : le déclenchement de l'opération de régulation de charge peut être effectué par les processeurs surchargés comme par les processeurs sous-chargés.

IV.3 Les systèmes OU multi-séquentiels

Rappelons que le parallélisme OU consiste en une évaluation en *largeur* d'un programme Prolog. Le problème de l'évaluation en largeur est le risque d'explosion combinatoire du nombre de processus parallèles : pour un arbre OU *n-aire* équilibré de hauteur h, le nombre de processus parallèles est au plus n^h.

La stratégie multi-séquentielle [Ali86] a pour but de contrôler la création de processus en fonction des ressources (processeurs/mémoire) disponibles. Le principe est simple : le degré de parallélisme maximum, c'est à dire le nombre de processus actifs à un instant donné est borné par le nombre de processeurs disponibles (cf. figure IV.4).

Il s'ensuit que l'évaluation du programme correspond à une combinaison d'une stratégie parallèle et d'une stratégie séquentielle classique (en profondeur). Le principe est de poursuivre une résolution parallèle, tant que les deux règles suivantes sont vérifiées :

- il existe des processeurs inactifs,

- l'accroissement du parallélisme produit un gain d'efficacité.

Dans le cas où tous les processeurs sont actifs ou lorsque l'accroissement du parallélisme peut dégrader les performances d'exécution, la résolution est poursuivie selon une stratégie séquentielle.

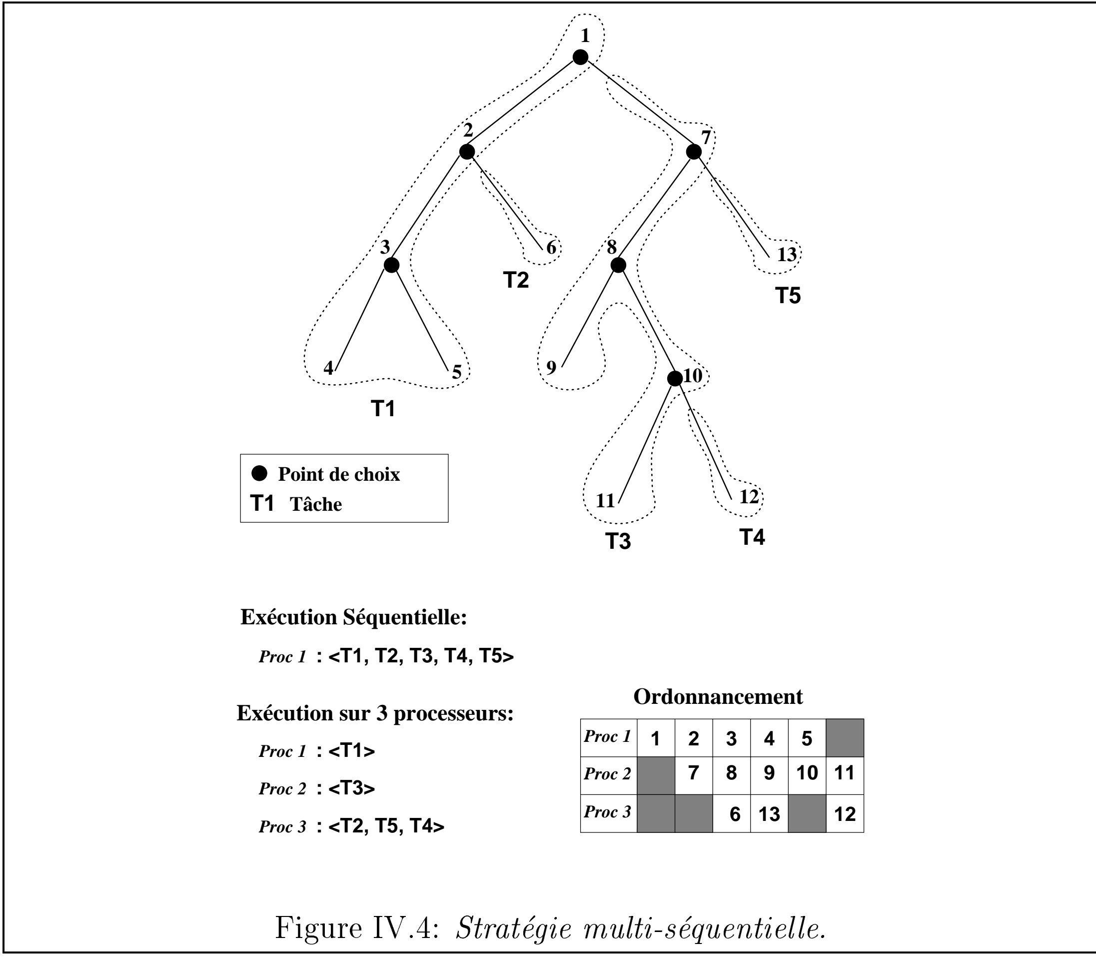


Figure IV.4: *Stratégie multi-séquentielle.*

Un système multi-séquentiel est alors vu comme un nombre fini de processeurs où :

1. chaque processeur poursuit une résolution séquentielle indépendante des autres processeurs actifs ;
2. chaque processeur produit des points de choix correspondants aux alternatives suspendues des différents nœuds OU rencontrés ;
3. un processeur inactif acquiert du travail auprès d'un processeur actif possédant un ou plusieurs points de choix en attente d'évaluation ;

4. l'acquisition d'un travail (alternatives) par un processeur inactif nécessite l'installation du contexte d'exécution (environnement) à l'instant de création du point de choix (nœud OU) considéré ;

5. les processeurs partagent nécessairement une partie de leurs résolvantes (partie commune dans l'arbre de recherche).

Une mise en œuvre des points (3, 4, 5) définit une stratégie de régulation de charge (une règle de partage du travail) pour un système OU multi-séquentiel. Nous nous intéressons dans la suite à la principale contrainte d'efficacité d'une stratégie de régulation donnée : garantir que la parallélisation de l'exécution d'un programme ne conduit pas à une efficacité inférieure à une exécution séquentielle.

IV.3.1 Formulation du problème de régulation

Le problème de la régulation de charge dans les systèmes logiques OU-parallèles peut s'énoncer de la manière suivante : un nombre P fixe de processeurs exécutent des tâches[1] de durées finies. L'exécution d'une tâche engendre un nombre indéterminé mais fini d'autres tâches. Ces tâches sont indépendantes les unes des autres.

La question qui se pose alors est la suivante : *comment garantir que la parallélisation d'une tâche produise un accroissement de performance par rapport à la solution laissant l'exécution de cette tâche sur le processeur qui l'a créée ?*

Intuitivement, on doit traduire le fait que l'exécution séquentielle d'un groupe de tâches est plus lente que l'exécution parallèle de ce groupe.

Si on considère qu'à un instant donné t, la durée résiduelle d'exécution $D_r(t)$ sur un processeur donné désigne la somme des durées d'exécution des tâches en attente sur le processeur plus le temps d'exécution restant pour la tâche en cours; la durée d'exécution d'une tâche D_T est sa durée propre plus la durée d'exécution de toutes les tâches qu'elle engendre. Le temps d'exportation E_T d'une tâche T est le temps nécessaire à l'amorçage du transfert et au ralentissement induit par le volume du transfert. Le temps d'importation I_T est le temps de transfert total plus le temps nécessaire au démarrage de la tâche T par l'importateur (cf. figure IV.5).

L'efficacité d'une parallélisation de la tâche T à l'instant t est conditionnée par la vérification de l'inégalité suivante :

$$D_r(t) \geq Max[(D_r(t) - D_T) + E_T, (D_T + I_T)]. \qquad \text{(IV.2)}$$

[1] *Nous désignons par le terme* ***tâche*** *une branche (alternative) de l'arbre de recherche. Un point de choix représente alors un ensemble (groupe) de tâches (alternatives en attente d'évaluation) partageant le même environnement (contexte) d'exécution initial.*

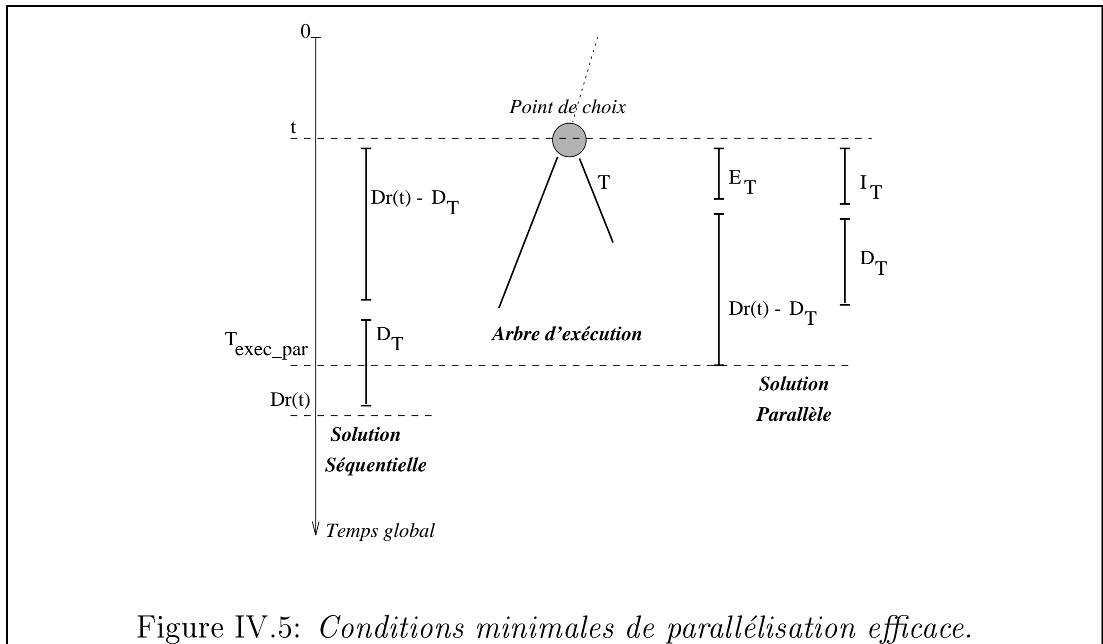


Figure IV.5: *Conditions minimales de parallélisation efficace.*

qui se traduit par la satisfaction des deux contraintes suivantes :

$$D_r(t) - D_T \geq I_T. \qquad \text{(IV.3)}$$

$$D_T \geq E_T. \qquad \text{(IV.4)}$$

La première contrainte IV.3 caractérise le grain de parallélisme minimum exploitable et la seconde IV.4 détermine l'opportunité de la mise en parallèle, compte tenu de l'état d'exécution du processeur exportateur.

Une fonction de régulation de charge vérifiant, simultanément à chaque invocation les deux contraintes (IV.3, IV.4) garantit que la parallélisation d'une tâche engendre une performance meilleure que celle obtenue par la solution qui laisse son exécution sur le processeur créateur. La question qui se pose alors est la suivante : *est-il possible de vérifier simplement et rapidement les deux contraintes à chaque invocation de la fonction de régulation de charge ?*

La difficulté majeure qui se pose dans la vérification des contraintes (IV.3, IV.4) provient du calcul des coûts d'exécution des tâches $(D_r(t), D_T)$. En effet, si les coûts d'importation et d'exportation (I_T, E_T) peuvent être estimés dynamiquement en fonction des caractéristiques de l'architecture cible et du volume de données à transférer, cela n'est souvent pas le cas pour les durées d'exécution des tâches. L'estimation de ces temps à la compilation est un domaine de recherche actif. Son but est de produire une fonction de calcul de ces temps qui serait évaluée à l'exécution (cf. Sections IV.5.1 et IV.5.2). La plupart des approches

actuelles se contentent d'estimations, sacrifiant l'exactitude à la simplicité (cf. Section IV.6.1 et IV.6.2).

IV.3.2 Conception et mise en œuvre

La difficulté de mise en œuvre d'une fonction de régulation de charge efficace repose sur :

- la définition de bons estimateurs de temps de calcul des tâches,
- la prise en compte des caractéristiques architecturales de la machine cible,
- la définition d'une bonne stratégie de répartition des tâches, applicable à tout programme Prolog.

La plupart des fonctions de régulation développées pour les systèmes OU multi-séquentiels sont fondées sur une représentation globale de l'arbre OU courant.

Si cette représentation est relativement simple à mettre en œuvre sur une architecture à mémoire partagée, de façon à ce que tous les processeurs aient un accès à l'ensemble de l'arbre d'évaluation, ce n'est souvent pas le cas pour une architecture sans mémoire commune. En effet, dans ce type de machines il n'existe pas de représentation complète de cet arbre mais seulement des représentations partielles. Ces représentations sont mises à jour par le régulateur, via l'échange de messages entre les processeurs.

Deux hypothèses de travail sont implicites dans l'élaboration d'une stratégie de régulation de charge sur une architecture à mémoire commune :

- On dispose d'un **état global exact** pour prendre une décision de régulation de charge et la structure de l'arbre OU (pile des points de choix) est accessible à tous les processeurs ;
- On connaît à tout moment la **distance**, en terme de nœuds OU, séparant deux processeurs.

Ces hypothèses de travail ne sont plus exactes dans une architecture à mémoire distribuée.

a) Stratégies de régulation

Plusieurs stratégies de régulation fondées sur la notion de partage de l'arbre d'exécution ont été proposées dans la littérature. La plupart des stratégies développées considèrent que chaque branche de l'arbre est partagée en deux sections : une section **privée** et une section **publique**. Tous les nœuds créés par

un processeur à partir d'une branche (alternative) qui lui a été attribuée lui sont *privés* et ne sont pas accessibles aux autres processeurs. La fonction de régulation peut décider de rendre *public* un nœud OU permettant ainsi aux processeurs oisifs d'y accéder. Les nœuds OU contenant des alternatives en suspens présents dans la section publique représentent l'ensemble des tâches disponibles.

L'idée de partager l'arbre d'exécution d'un programme en sections privées et publiques (cf. figure IV.6) a été introduite en programmation logique par D.H.D Warren [War87]. Des notions similaires ont également été introduites pour la régulation dynamique de charge de programmes de recherche arborescente, sur lesquels on ne dispose d'aucune information a priori (A*, Minmax, α-β) [KR87, KR90, YCD94, Cun94].

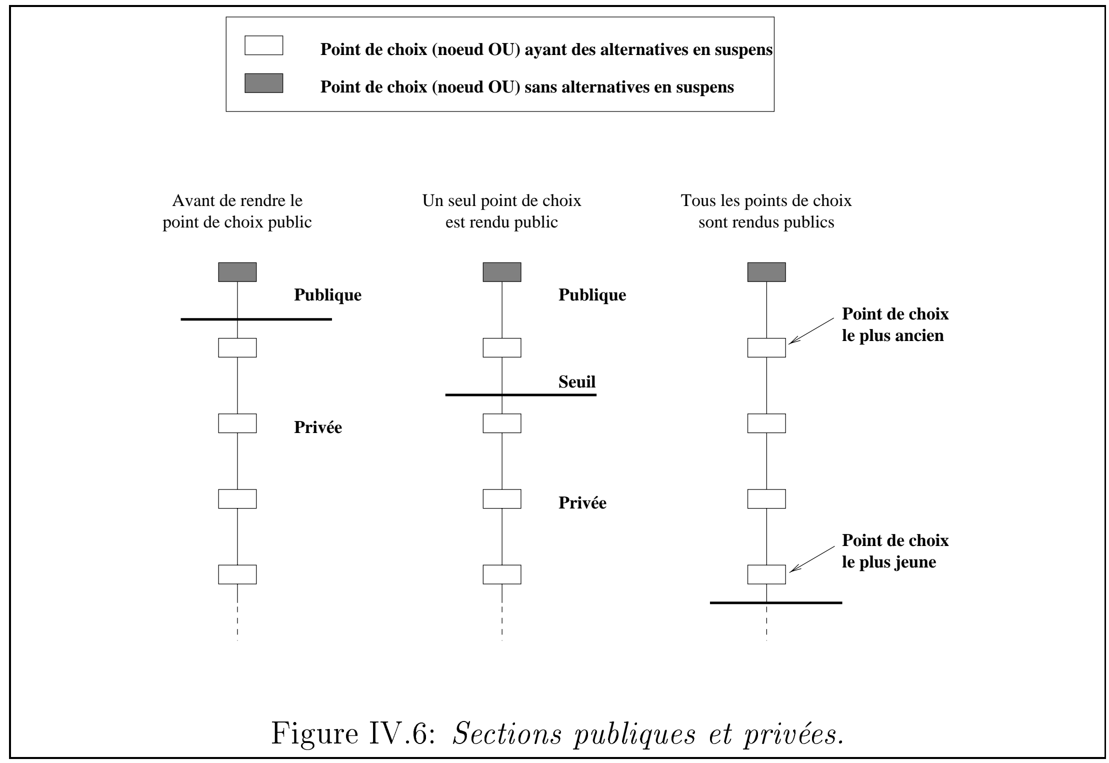


Figure IV.6: *Sections publiques et privées.*

Les différentes stratégies de régulation de charge proposées dans la littérature diffèrent par les réponses apportées aux points suivants :

1. à quel moment est-il nécessaire d'effectuer une opération de régulation de charge (transfert de tâches d'un processeur à un autre) ?

2. quel processeur a l'initiative du déclenchement de l'opération de régulation de charge ?

3. s'il n'existe plus de tâches disponibles dans les nœuds OU dans la partie publique de l'arbre, quel processeur doit rendre publique une partie de ses nœuds OU ?
4. de quelle manière l'arbre doit-il être parcouru pour la recherche des tâches disponibles ?
5. quel nombre de tâches doit-on transférer lors d'une opération de régulation de charge ?
6. de quelle manière les processeurs oisifs sont-ils sélectionnés ?
7. dans quel ordre les tâches doivent-elles être attribuées aux processeurs ?

Ce dernier point est généralement utilisé pour distinguer trois types de **stratégies de sélection** de tâches sur une branche donnée de l'arbre :

- sélection des tâches à partir du nœud OU le plus ancien (cf. figure IV.7(a)),
- sélection des tâches à partir du nœud OU le plus jeune (cf. figure IV.7(b)),
- sélection des tâches à partir de l'ensemble des nœuds OU disponibles (cf. figure IV.13).

b) Efficacité d'une fonction de régulation

Les deux rôles élémentaires d'une fonction de régulation de charge sont la détection des tâches disponibles et l'allocation de ces tâches aux processeurs. Une fonction de régulation de charge n'est pas uniquement supposée trouver la tâche appropriée au moyen d'un critère quelconque pour un processeur oisif ; elle doit aussi la trouver rapidement.

Il faut noter que la vitesse d'exécution de la machine Prolog a été grandement améliorée au cours des dernières années et que de nouvelles techniques ont été développées. Ces techniques devraient permettre d'augmenter encore la vitesse d'exécution de manière significative [Roy90].

La mise en œuvre d'une fonction de régulation de charge efficace est un défi d'autant plus important que cette mise en œuvre doit prendre en compte l'évolution constante des systèmes séquentiels. En effet, alors que les optimisations d'implantation des moteurs Prolog manipulent des ressources locales (mémoire, indexation de registres) une fonction de régulation de charge requiert, quant à elle, des informations globales pour prendre ces décisions.

En d'autres termes, on ne peut s'attendre à ce que les performances des méthodes de régulation de charge augmentent proportionnellement à l'évolution des

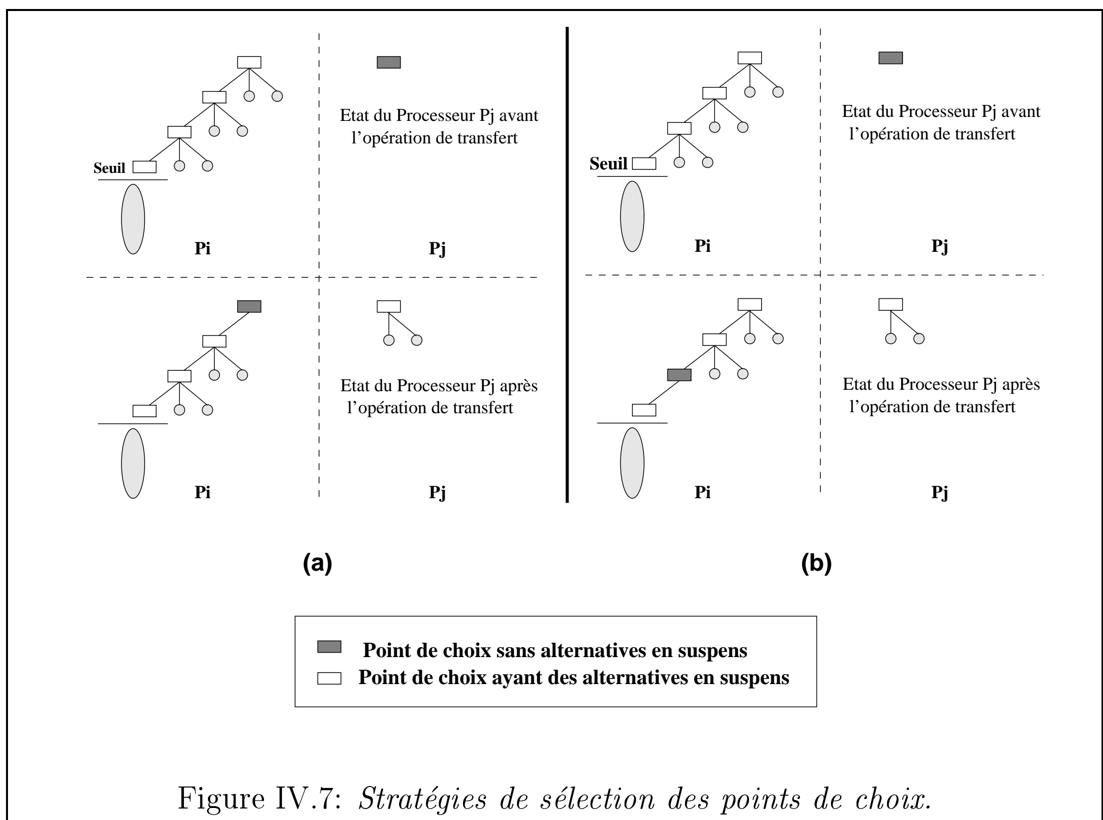


Figure IV.7: *Stratégies de sélection des points de choix.*

moteurs Prolog séquentiels, sans prendre en compte le problème potentiel de l'engorgement de l'accès à des ressources partagées (les caches dans les machines à mémoire partagée et les liens de communication dans les machines sans mémoire commune).

Répondre à cette contrainte supplémentaire qu'est l'accroissement de performance des systèmes séquentiels nécessite la mise en œuvre de mécanismes et stratégies appropriés, de façon à minimiser les coûts induits par la fonction de régulation de charge. Ces coûts dépendent de manière générale :

- du coût d'estimation de la charge du système,
- du coût de sélection de la tâche à transférer,
- du coût de transfert de la tâche sélectionnée,
- du nombre de transferts effectués.

Une fonction de répartition efficace tente de minimiser ces coûts par la réduction de la fréquence d'invocation de la fonction de répartition et l'optimisation de l'algorithme de régulation.

IV.4 Transfert de la charge de calcul

Dans un système multi-séquentiel, chaque processeur de l'architecture parallèle exécute une machine abstraite (WAM) classique. L'état d'exécution d'un processeur à un instant donné est représenté par une pile d'environnements chaînés, contenant des variables logiques liées entre elles lors d'unifications intervenues depuis le début d'exécution du programme. Le contexte d'exécution d'une alternative (tâche) en suspens à un point de choix (nœud OU) est représenté par l'état d'exécution du processeur à l'instant de création de ce point de choix. L'installation de cette alternative (tâche) sur un autre processeur nécessite de :

- construire le contexte de la tâche,
- transférer ce contexte (données),
- installer ce contexte.

IV.4.1 Extraction du contexte d'exécution

Il existe trois approches pour la construction du contexte d'exécution d'une tâche (alternative) :

- le partage d'une mémoire physique contenant ce contexte,
- la recopie (duplication) du contexte,
- la réplication du calcul.

Le choix d'une technique parmi les trois est guidé par le support de communication utilisé pour l'échange des données entre les processeurs d'une machine.

a) Machines à mémoire partagée

Dans ce type d'architectures, l'espace d'adressage est global à tous les processeurs *(architectures* UMA *et* NUMA*)*. Les modèles multi-séquentiels implantés sur ces machines exploitent donc la propriété qu'a un processeur de pouvoir accéder à la mémoire des autres pour partager des contextes d'alternatives évaluées en parallèle. Le problème qui se pose alors est de gérer l'environnement de données partagées par les branches parallèles (maintien de la cohérence des accès aux variables des environnements partagés).

Le principe retenu par les différents modèles proposés repose sur l'identification de l'environnement correspondant à chacune des branches OU parallèles. Il existe plusieurs possibilités que nous regroupons en trois classes :

- les modèles qui gèrent les variables conditionnelles dans un tableau *(hash code)* privé à chaque branche OU parallèle [Bor84, Hau90, War87],
- les modèles qui gèrent une copie explicite de l'environnement de chaque branche OU parallèle [Ali90, CH86],
- les modèles qui gèrent l'ensemble des liaisons conditionnelles dans une même structure commune, en attachant à chacune un identificateur désignant la branche OU parallèle correspondante [Tin88].

La plupart des modèles multi-séquentiels actuels sont implantés sur des machines à mémoire partagée.

b) Machines à mémoire distribuée

Dans ce type d'architectures, l'espace d'adressage est local à chaque processeur *(architectures* NORMA*)*. Il est nécessaire de créer une copie exclusive du contexte de la résolvante courante à chaque branche OU-parallèle, étant donné que le partage d'information est physiquement impossible. Le contexte de la résolvante est une image mémoire de l'exécution à l'instant de création du nœud OU (piles WAM).

Cette image mémoire peut être obtenue, soit par **recopie** du contexte (cf. figure IV.8), les piles sont alors transférées du processeur qui a créé le nœud OU (processeur père) au processeur destinataire (processeur fils) [Kum86, Con87a, Gey91, Fav92, BFGdK92, BT92, BGP95], soit par **réplication des calculs** [CA88, Kac90, KE92, Kac93, Kac94a] chaque processeur exécutant la même séquence de code jusqu'au point de choix divergent.

Dans l'approche par recopie de contexte, les variables conditionnelles à un nœud OU (point de choix) sont ré-initialisées (défaites). Dans l'approche de recalcul, le travail attribué à chaque processeur est déterminé statiquement ou dynamiquement. Une règle de codage statique des branches *(oracles)* permet d'identifier pour chaque processeur le chemin à reparcourir. Cette technique évite la communication de contexte (qui peut être prohibitive sur certaines architectures).

Dans une approche dynamique, chaque processeur reçoit la requête initiale. L'activation d'une branche parallèle consiste à envoyer au processeur destinataire la branche *(l'oracle)* à suivre dans l'arbre d'exécution. Chaque alternative d'un nœud OU de l'arbre d'exécution est identifiée par un code *(oracle)*. Cet oracle est calculé (attribué) dynamiquement en fonction de la profondeur du nœud OU dans l'arbre, et de l'ordre des alternatives pendantes à ce nœud. L'identité $I = e_0 e_1 ... e_h$ d'un nœud OU est définie récursivement de la manière suivante [LW90] : le nœud OU racine de l'arbre d'exécution (se trouvant au niveau 0) est identifié par 0. Le nœud OU P_{ij} se trouvant au niveau i et généré par la jme alternative d'un

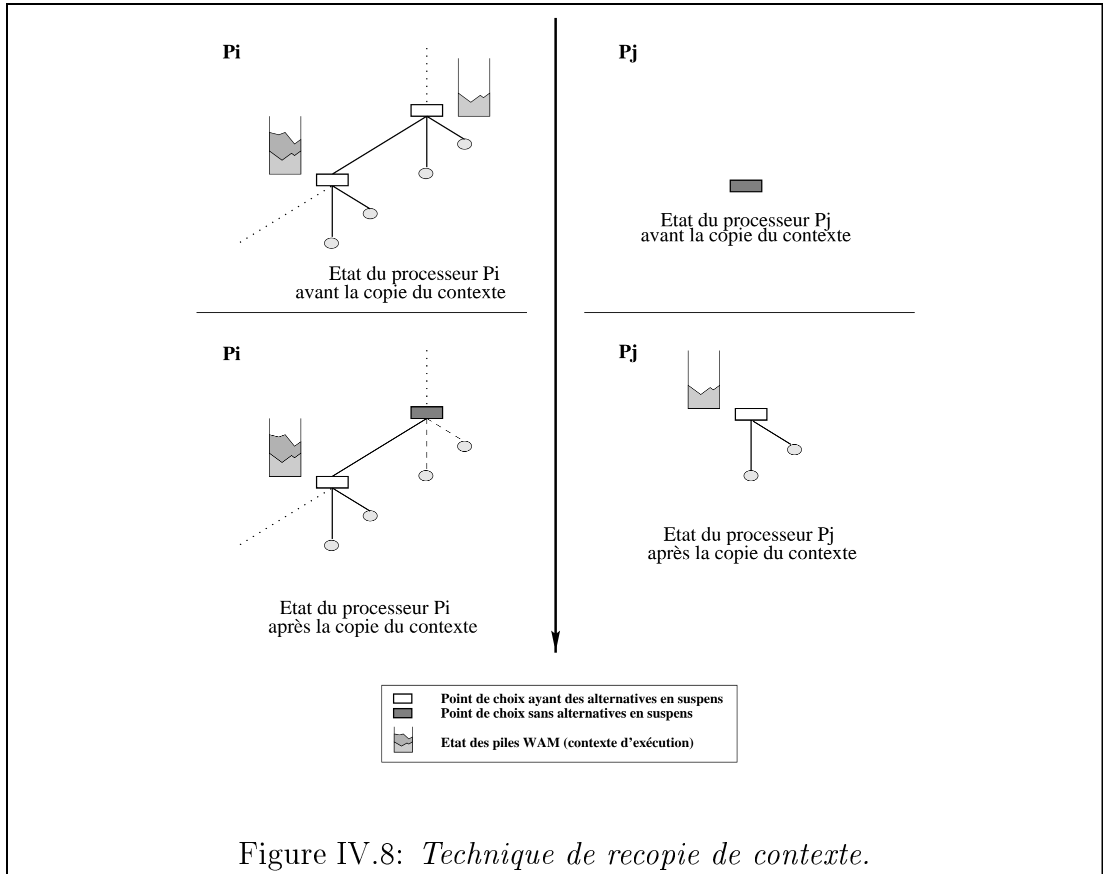


Figure IV.8: *Technique de recopie de contexte.*

nœud de niveau $(i-1)$ est identifié par la séquence $E_{ij} = e_0 e_1 ... e_{i-1}(j-1)$. Un exemple est illustré par la figure IV.9, pour la codification d'un arbre binaire avec un vecteur de bits. Les deux approches (statique et dynamique) introduisent une déperdition due au calcul redondant effectué.

IV.4.2 Coûts d'installation

Le coût engendré par ce partage sur les performances globales d'exécution dépend :

- du coût d'accès à une mémoire non-locale,
- du nombre d'accès aux contextes partagés.

L'impact de ces coûts sur les performances du système peut favoriser le choix d'une technique de copie, même si l'on dispose d'une machine à mémoire partagée ou d'un mécanisme de mémoire virtuelle partagée sur une machine sans mémoire commune.

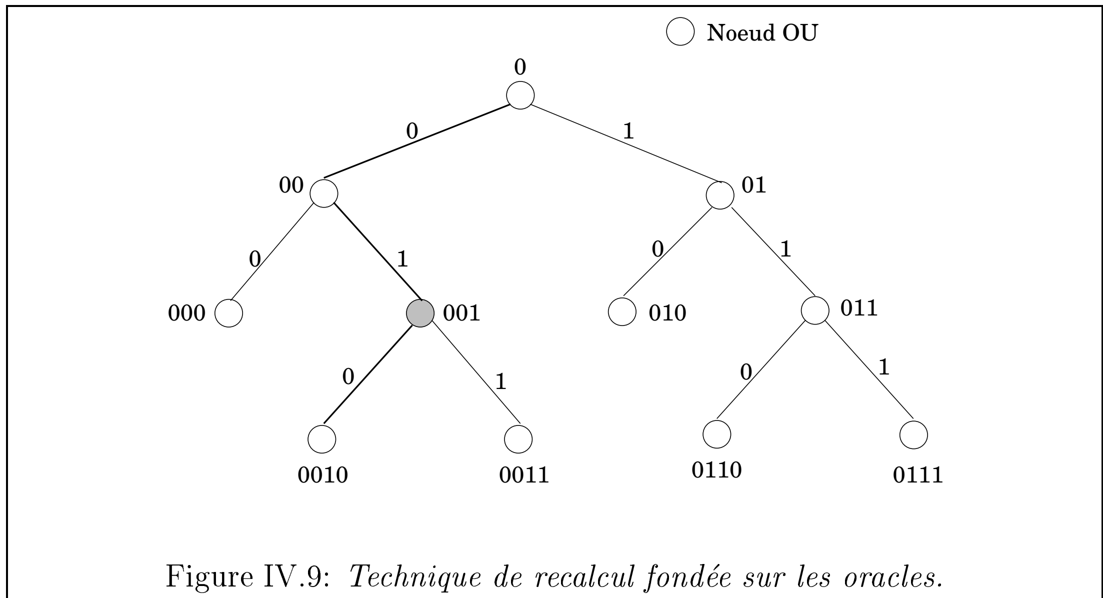


Figure IV.9: *Technique de recalcul fondée sur les oracles.*

Avec une technique de copie, le coût principal d'installation d'une tâche réside dans l'opération de transfert (envoi des données).

L'objectif principal, lors de la mise en œuvre d'un mécanisme de transfert fondé sur la technique de recopie, est de diminuer les coûts d'initialisation (installation) des tâches. Cet objectif se traduit par :

- la limitation du volume des données à transférer,
- la restriction des contraintes de synchronisation,
- la diminution du temps de calcul spécifique à l'installation.

La limitation de la taille des données à transférer nécessite l'utilisation de techniques d'optimisation de copie. Les techniques proposées exploitent le fait que deux parcours dans l'arbre OU ont nécessairement une portion de chemin commun. L'idée consiste alors à ne recopier, lors d'un transfert, que les parties non communes. Cette technique n'est applicable que si le processeur cible du transfert a déjà participé à la résolution d'une partie du programme.

Le transfert du contexte d'une tâche nécessite de synchroniser les processeurs concernés pour, d'une part, maintenir la cohérence du contexte transféré au processeur cible et pour, d'autre part, garantir la reprise du calcul à partir d'un état valide sur le processeur source du transfert. Restreindre cette synchronisation nécessite d'avoir une représentation adéquate des structures de données [Gey91, Fav92] manipulées pour permettre au processeur source de poursuivre le calcul et d'effectuer les opérations d'installation du contexte sur le processeur cible pendant le transfert.

IV.5 Evaluation de la charge du système

Les décisions de répartition dynamique de la charge globale sont fondées sur une estimation de l'état du système à partir duquel une décision de redistribution de la charge peut être prise. L'estimation de l'état du système est effectuée à partir des états de charge des processeurs. Il est donc nécessaire d'avoir un indice de charge caractéristique du taux d'utilisation des ressources du processeur (CPU, mémoire). Cette valeur doit également être facilement calculable afin de minimiser le surcoût d'estimation de la charge. La difficulté quant à l'identification de cet indice a conduit à deux approches différentes pour l'évaluation de la charge d'un processeur :

- l'analyse de complexité,
- l'utilisation d'heuristiques.

Dans la première approche, la charge du processeur est dérivée de l'analyse de complexité du programme considéré. Cette complexité peut être établie à partir du programme, par une annotation fournie par le programmeur ou par un compilateur procédant à une analyse de complexité du programme [DLH90, Giu90, HWD92].

La deuxième approche met en œuvre une heuristique d'estimation de la charge d'un processeur à partir d'éléments observables de l'exécution du programme [KR88, Bal91].

Le problème dans les deux cas est que la valeur fournie par une telle estimation se situe dans une métrique abstraite qui doit être ramenée à une échelle de temps physique.

IV.5.1 Estimation des coûts de calcul

Ces méthodes consistent à extraire, à la compilation, des paramètres caractéristiques du programme, afin d'évaluer sa complexité en terme de :

- granularité de calcul qu'il génère,
- quantité de ressources qu'il utilise.

De nombreux travaux sont effectués dans le but de déterminer la complexité des algorithmes [KL88, MG89]. En programmation logique, il s'agit d' établir à la compilation une expression permettant de calculer le poids du prédicat (une charge abstraite). Cette expression est alors évaluée à l'exécution, à partir des valeurs entrées en arguments du programme.

Pour un prédicat P/n d'arité n, on détermine à la compilation une fonction d'estimation de coût $\Phi_P(n)$. Cette expression est évaluée à l'exécution, une fois que les arguments d'appel du prédicat $(Arg_1, Arg_2, ..., Arg_n)$ sont connus.

$$\Phi_P \ : \ (Arg_1, Arg_2, ..., Arg_n) \longrightarrow co\hat{u}t_P(n)$$

$\Phi_P(n)$ est une approximation de la complexité de l'appel au prédicat P/n. Cette approximation peut être de deux natures:

- *approche optimiste* : on choisit Φ_P de sorte que le coût estimé du prédicat soit une borne supérieure majorant tous les coûts réels [DLH90].

  $$\forall n, \quad co\hat{u}t_P(n) \leq \Phi_P(n)$$

  Cette approche favorise le parallélisme, mais si l'écart est trop important, le coût estimé n'est plus représentatif d'une charge de calcul.

- *approche pessimiste* : on choisit Φ_P de sorte que le coût estimé du prédicat soit une borne inférieure minorant tous les coûts réels [DGHL94].

  $$\forall n, \quad co\hat{u}t_P(n) \geq \Phi_P(n)$$

  Cette approche ne favorise pas le parallélisme, mais rend le coût estimé plus représentatif (robuste) d'une charge de travail.

L'estimation du coût d'un appel de prédicat dépend en particulier :

- du nombre d'arguments manipulés,
- de la complexité structurelle de ses arguments,
- du nombre de solutions possibles.

L'élaboration d'une *bonne* fonction de coût dépend donc de la connaissance que l'on a de ces éléments pour chaque prédicat du programme [DLH90]. Pour cela, des annotations sont rajoutées au programme permettant d'analyser le flot des données manipulées lors de son exécution. L'annotation du programme consiste à caractériser chaque prédicat du programme, non seulement par ses termes fonctionnels et son arité mais encore par son **mode** d'appel et le **type** des arguments qu'il manipule. L'analyse du programme permet en particulier de construire :

- un graphe des appels de prédicats,

- un graphe de dépendance des littéraux dans chaque clause,
- un graphe de dépendance des arguments dans chaque clause.

Considérons l'exemple de prédicat `q/2` de la figure IV.10 dont le premier argument est lié à l'appel de `q/2` (variable entrante notée +).

```
q([], []).
q([H|T], [X|Y]) :- X is H+1,
                   q(T, Y).
```

Figure IV.10: *Analyse de complexité.*

Considérons également que l'unité de coût est le nombre d'étapes de résolution. Pour des raisons de simplicité, nous prenons comme approximation du coût d'une étape de résolution (appel de prédicat) le même coût que le prédicat prédéfini `is/2`. Nous pouvons alors définir la fonction de coût du prédicat `q/2` comme :

$$co\hat{u}t_q(n) = 2.n + 1$$

où n représente la taille de la liste passée par le premier argument d'appel du prédicat `q/2`.

IV.5.2 Estimation de la charge

Garantir que le parallélisme produise un accroissement de performance nécessite de vérifier simultanément les contraintes (IV.3 et IV.4). Cette vérification implique que l'on puisse estimer d'une manière précise la charge de chaque processeur du système.

Les méthodes d'estimation des coûts de calcul à la compilation ont pour but de pondérer (donner une charge approximative) les prédicats et éventuellement chaque alternative d'un point de choix (nœud OU). Le principe (cf. section IV.5.1) consiste à inférer à la compilation une fonction de coût paramétrée pour chaque prédicat (alternative) du programme. On peut ainsi estimer la charge d'un processeur à un instant donné par un critère combinant le calcul des fonctions de coût affectées à chaque point de choix. L'idéal serait de pouvoir estimer le coût de chaque alternative en suspens ; on aurait ainsi une meilleure connaissance des durées résiduelles d'exécution sur chaque processeur, ce qui permettrait de prendre une décision de régulation de charge appropriée.

Bien qu'attrayante, cette approche présente néanmoins certains inconvénients :

- les techniques de compilation permettant de pondérer les prédicats (alternatives) par une fonction de coût sont lourdes et complexes à mettre en œuvre. Leur efficacité dépend du bon modage et typage du programme nécessaires pour la pertinence des informations extraites ;
- Le calcul (l'évaluation) des fonctions de coût à l'exécution génère des surcoûts qui viennent s'ajouter aux coûts induits par la fonction de régulation. Cela est en contradiction avec la contrainte d'efficacité énoncée en section IV.3.2. Par conséquent, un compromis reste à trouver entre la pertinence de l'information de charge et les coûts induits pour l'obtenir.

Une alternative à ces problèmes consiste à utiliser des méthodes heuristiques pour l'estimation de la charge d'un processeur. Ces méthodes considèrent des indices de charge simples observables à tout instant de l'exécution du programme, sacrifiant ainsi l'exactitude (pertinence) à la simplicité d'évaluation. Comme indices de charge le plus souvent retenus, on peut citer par exemple :

- le nombre de points de choix cumulés par le processeur,
- le nombre d'alternatives en suspens sur le processeur,
- la profondeur des points de choix dans l'arbre ET/OU.

Il est possible également de concevoir un indice de charge fondé sur une combinaison linéaire des indices précédents, en leur attribuant des coefficients de pondération dans l'estimation de la charge du processeur.

$$Charge = \sum_i \beta_i.N_i \tag{IV.5}$$

La charge du processeur peut également tenir compte de la taille mémoire disponible. L'équation IV.6 illustre un exemple de calcul de la charge en fonction du taux d'utilisation mémoire. L'idée est alors de minimiser la contribution du taux d'utilisation mémoire dans l'estimation de la charge du processeur, jusqu'à ce qu'elle devienne saturée (où λ est une constante : $0 < \lambda < 1$ et M la fraction de la mémoire utilisée).

$$Charge = \frac{\lambda}{1-M} + \sum_i \beta_i.N_i \quad avec \tag{IV.6}$$

L'inconvénient majeur de cette approche réside dans le fait que la relation d'ordre que l'on établit à partir d'une heuristique donnée entre les processeurs suppose intuitivement une corrélation entre l'indice de charge mesuré et la charge réelle du processeur. Or, cela n'est pas toujours vrai, car la charge de calcul effective dépend souvent des données manipulées :

- la taille, le type et le contenu des arguments d'appel d'un prédicat,
- la complexité (en terme temps de calcul) de chaque alternative.

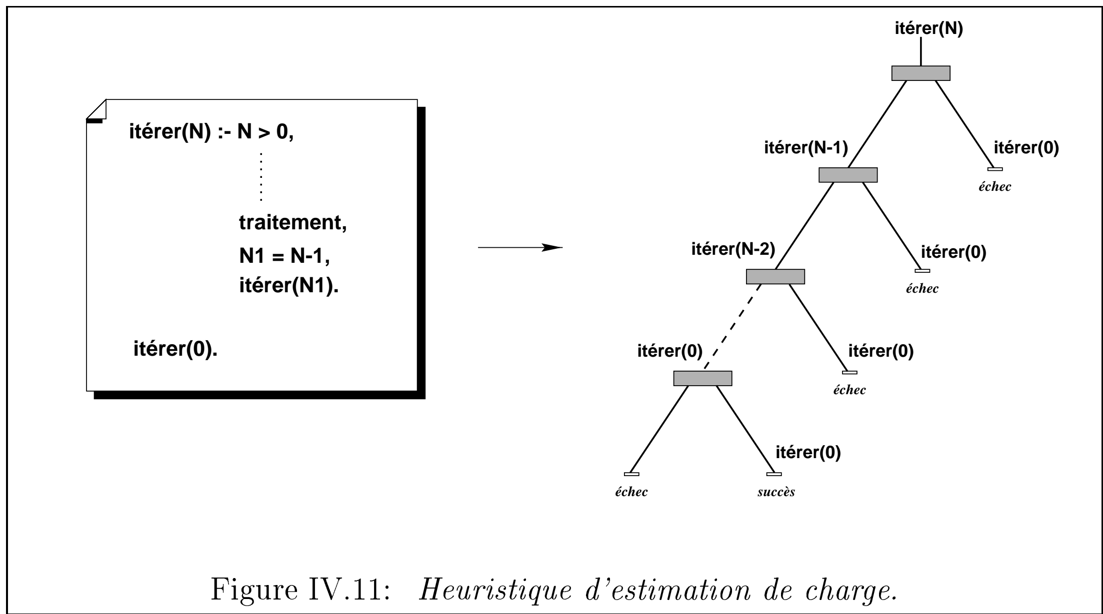


Figure IV.11: *Heuristique d'estimation de charge.*

Si l'on considère par exemple le cas de programme de la figure IV.11 et que la charge d'un processeur, à un instant donné, est proportionnelle au nombre de points de choix qu'il a en suspens, on s'aperçoit que l'on risque de concevoir le processeur comme étant chargé, alors qu'en réalité les alternatives en suspens sont toutes rapidement vouées à un échec (sauf la dernière) lors du retour-arrière.

IV.6 Contrôle du grain

Le contrôle du parallélisme dans les systèmes logiques parallèles doit principalement assurer la fonction de répartition homogène de la charge de travail entre les processeurs. Cette fonction doit également assurer le maintien d'une structure permettant de reconstruire l'ordre séquentiel pour la gestion des prédicats à effets de bord (travail spéculatif).

L'algorithme de régulation de charge doit être conçu de telle manière que les processeurs (unités de travail) soient actifs la plus grande partie du temps, tout en limitant le surcoût dû au parallélisme. La première contrainte nécessite de générer suffisamment de parallélisme, tandis que la seconde nécessite que la granularité de chaque nouvelle tâche parallèle reste supérieure au surcoût inhérent à sa création (installation). La difficulté de mise en œuvre d'une fonction de régulation garantissant cet objectif réside principalement dans la satisfaction de cette

dernière contrainte. Ce problème est particulièrement vrai pour une implantation sur une machine parallèle sans mémoire commune.

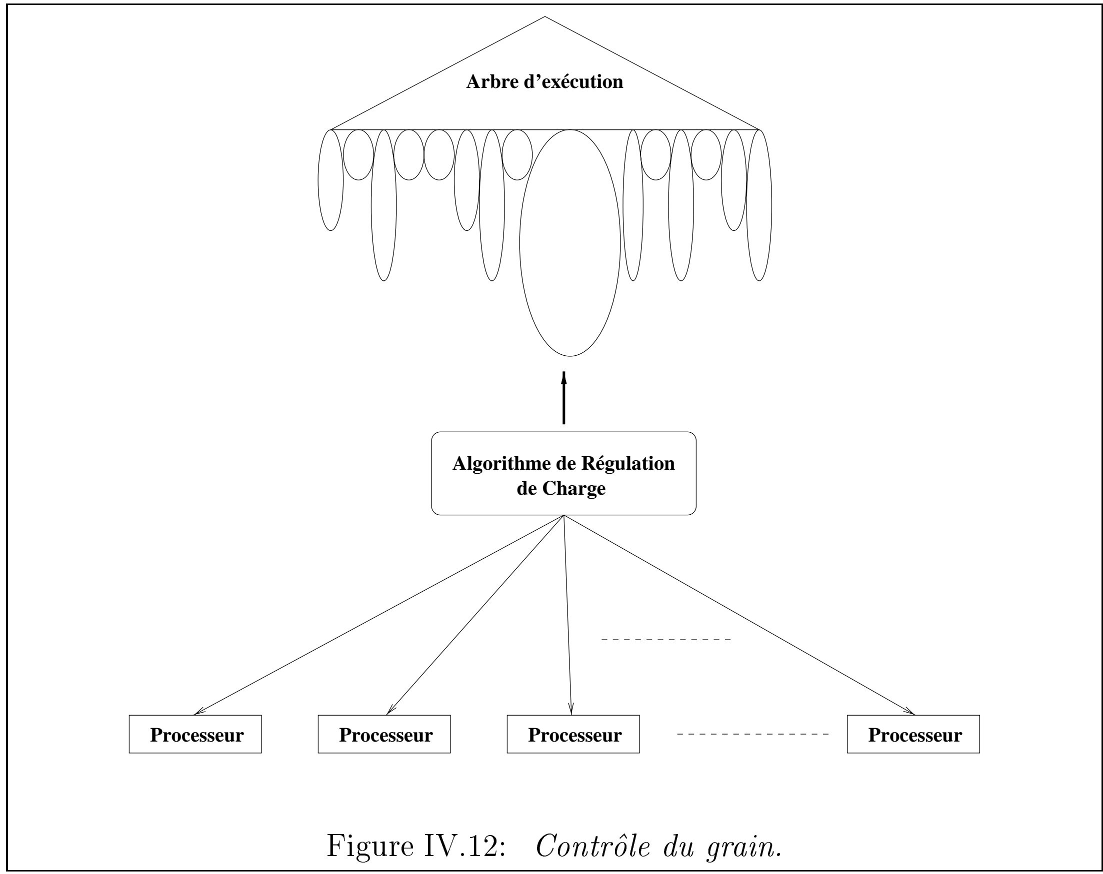


Figure IV.12: *Contrôle du grain.*

La régulation de charge est plus difficile à effectuer sur des architectures sans mémoire partagée que sur des machines à mémoire partagée, ceci pour deux raisons :

- le coût plus élevé pour la création des tâches,
- la communication de données se fait par échanges de messages dont le coût est très supérieur à celui d'un accès à la mémoire.

Le coût de création d'une tâche exige un grain important de la tâche, sans lequel les performances de l'exécution peuvent chuter. Cependant, la prédiction de la granularité d'une tâche est dans le cas général impossible à calculer précisément. Deux approches différentes sont développées pour tenter de pallier à ce problème de contrôle de la granularité des tâches parallèles :

- la transformation du programme,

- l'utilisation d'heuristiques.

La première approche repose sur l'analyse de complexité du programme effectuée à la compilation. La pondération des prédicats (clauses) du programme par des poids sur leur complexité (établi par des fonctions de coût) suggère la réécriture du programme en tenant compte de la taille de chaque tâche. Dans cette même approche, une autre solution serait de confier à l'utilisateur le choix d'un système de pondération de chaque branche d'évaluation. Cette façon de définir la charge est très lourde pour le programmeur. Dans les deux cas, cela suppose une modification du programme source.

La seconde approche consiste à prédire dynamiquement et de manière heuristique la taille des tâches en fonction d'éléments observables. Cette approche peut également être combinée à une analyse statique simple du programme, afin de produire des poids sur la complexité relative des clauses ou de permettre d'extraire certaines informations de complexité des prédicats (nombre de points de choix, nombre d'alternatives par prédicat).

IV.6.1 Transformation du programme

Considérons la clause suivante :

$$h \ :- \ ..., L, L_1, ..., L_n. \qquad (1)$$

où le littéral L représente le prédicat p dont la définition contient a clauses éligibles $\{Cl_1, ..., Cl_a\}$. Chaque clause Cl_i est définie par une clause de la forme $h_i \ : \ b_i$. L'analyse de granularité (complexité) permet de pondérer chacune des a clauses éligibles par une fonction de coût qui est évaluée à l'exécution. Dans une exécution OU-parallèle du littéral L, les a clauses et leurs continuations (le reste des littéraux L_i représentant chacune des a résolvantes) sont évaluées en parallèle. Considérons que $Coût_{Cl_i}(x)$ et $Coût_{L_i}(x)$ représentent respectivement le coût de la clause Cl_i et du littéral L_i. Le coût du choix de la clause Cl_i est noté $Coût_{ch_i}(x)$.

Si l'évaluation de la clause Cl_i et des m premiers littéraux ne conduit pas à un *échec* (le littéral L_m est le premier littéral pour lequel on ne garantit pas un succès), le coût $Coût_{ch_i}(x)$ peut alors être estimé par :

$$Coût_{ch_i}(x) = Coût_{Cl_i}(x) + \sum_{j=1}^{m} Coût_{L_j}(x)$$

Si par contre l'évaluation de la clause Cl_i peut engendrer un *échec*, ce coût est alors estimé par :

$$Coût_{ch_i}(x) = Coût_{Cl_i}(x)$$

L'idée du contrôle automatique de granularité consiste alors à conditionner la parallélisation d'une ou plusieurs clauses, en fonction de leurs coûts estimés [HG91, HWD92, ZT92, MdlBH94, GHD94]. Il s'agit de transformer la clause (1) en une clause relisant ce contrôle :

$$h \ :- \ ..., (condition \rightarrow L' \ ; \ L), L_1, ..., L_n. \qquad (2)$$

où L' est la version parallèle de L. Il est créé à partir de L en remplaçant le prédicat p par p' dans chacune des clauses. Si la condition de contrôle de granularité *condition* est vérifiée, la version parallèle p' du prédicat est exécutée, sinon c'est la version initiale (p) qui est exécutée en séquence.

Le problème de cette approche réside dans le fait que, ou bien toutes les clauses sont traitées en parallèle, ou leur traitement est complètement séquentiel. Etant donné que chacune des clauses peut avoir un coût de traitement différent, cela peut engendrer une mauvaise régulation de charge. Une approche plus intéressante consiste à spécifier des paquets (*cluster*) de clauses. Les paquets sont alors évalués en parallèle, tandis que les clauses appartenant à un même paquet sont traitées de manière séquentielle.

Bien qu'attrayante, cette approche de contrôle du parallélisme a deux inconvénients majeurs ; très lourde à mettre en œuvre, elle induit des surcoûts importants dus à l'estimation de la complexité des différentes clauses d'un prédicat. Ces surcoûts viennent s'ajouter également aux coûts de sélection et de transfert des tâches éligibles. Même si des optimisations sont proposées par différents travaux pour réduire le coût inhérent au contrôle de la granularité des opportunités parallèles, l'utilisation effective de cette approche se limite pour l'instant à ranger les prédicats suivant deux classes : parallélisable, non parallélisable.

IV.6.2 Approches heuristiques

La plupart des méthodes heuristiques utilisées pour le contrôle du parallélisme dans les systèmes OU multi-séquentiels sont fondées sur une représentation globale de l'arbre OU courant. On distingue principalement deux approches souvent retenues :

- le regroupement des points de choix (alternatives) (cf. figure IV.13),
- la distance d'un point de choix par rapport à la racine de l'arbre OU (cf. figure IV.7(a)).

Ces deux approches ont pour objectif commun d'augmenter la granularité de la tâche à transférer sur un processeur oisif, de manière à vérifier la contrainte (IV.3).

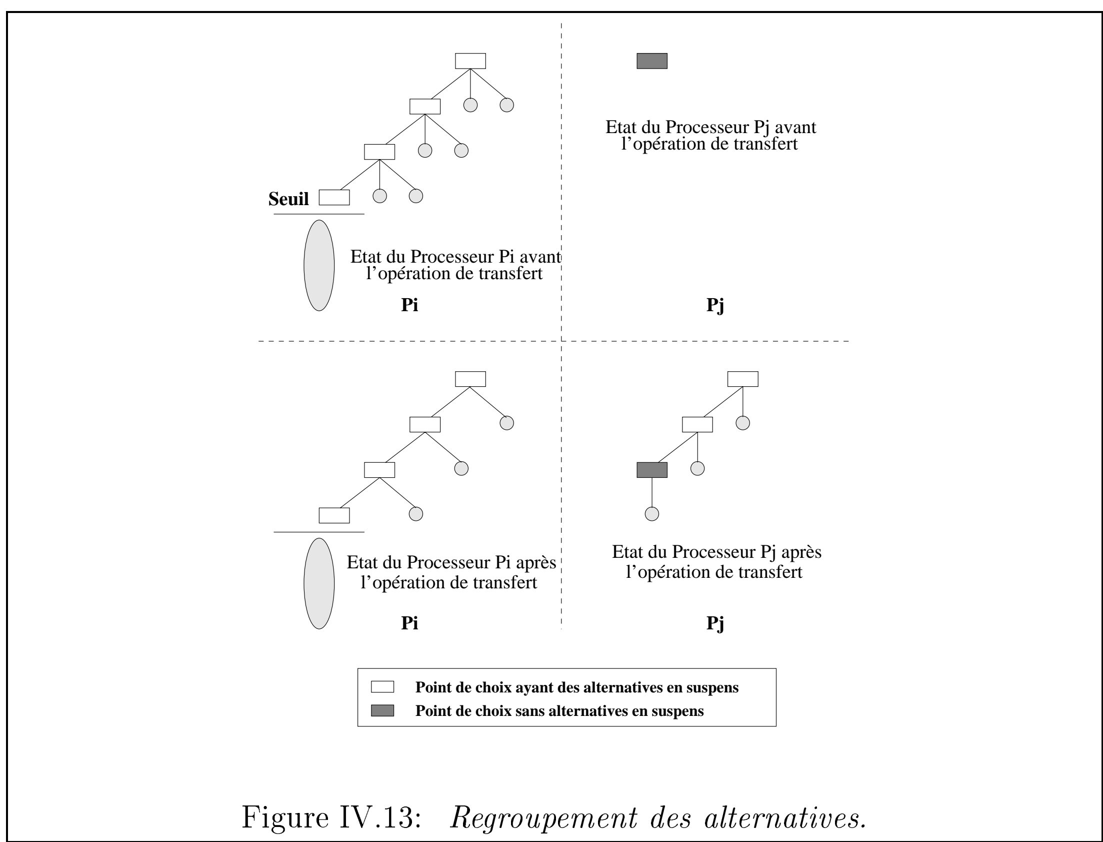


Figure IV.13: *Regroupement des alternatives.*

La première approche consiste à partager l'ensemble des alternatives en suspens sur un processeur en deux groupes. L'un est transféré au processeur oisif alors que le second est traité localement. L'heuristique (règle) de partage doit tenir compte des caractéristiques de la machine cible et maintenir suffisamment de travail sur le processeur exportateur (contrainte IV.4).

La deuxième approche repose sur l'heuristique suivante : *plus un point de choix est situé haut dans l'arbre* OU*, plus grande est la probabilité d'avoir une granularité importante*. Une propriété intéressante de cette heuristique est que, sur une même branche, les coûts d'installation (volume de données à transférer) croissent avec la profondeur. En choisissant le point de choix le plus près de la racine, on peut espérer minimiser les coûts de transfert et d'installation (contraintes IV.3 et IV.4).

Le problème majeur de ces deux approches rejoint ceux cités en Section IV.5.2 concernant l'estimation de la charge d'un processeur. Si l'on considère l'exemple de programme de la figure IV.11, on constate que l'utilisation de l'une des deux heuristiques peut dégrader les performances, du fait que les alternatives en suspens sont vouées à un échec rapide, violant ainsi les contraintes (IV.3 et IV.4). En effet, aucune de ces deux approches n'établit une réelle corrélation entre la pro-

fondeur d'un point de choix (ou un groupe alternatives)et une durée d'exécution. Etablir cette corrélation nécessite la prise en compte de la complexité (coût de calcul) de la continuation d'une résolvante (alternative).

Une approche intéressante serait de combiner une technique fondée sur une analyse de complexité permettant d'annoter les points de choix (alternatives) par une fonction de coût simple (une valeur probabilistique représentative du travail exportable qui pourrait être généré) avec une heuristique combinant les deux approches précédentes.

D'autres méthodes heuristiques sont également proposées de manière à contrôler le parcours parallèle de l'arbre de recherche. Ces techniques s'inspirent essentiellement des algorithmes d'**évaluation, séparation et élagage** *(Branch and bound and pruning)* du type A*, Minmax et $\alpha - \beta$ [Bal91, Sze91b, Sze91a] utilisés en intelligence artificielle et en optimisation combinatoire.

IV.7 Quelques cas de fonctions de régulation

La régulation de charge est un domaine où les recherches sont tout particulièrement actives. Nous avons choisi de présenter dans cette section les stratégies développées pour les deux principaux systèmes Prolog parallèle *Aurora* [LWH90] et *Muse* [Ali90, AKM91] et la stratégie retenue pour le système OPERA [Gey91, Fav92].

Sélection des tâches à partir:	Muse [Ali90]	Aurora [LWH90]	Opera [Fav92]	Delphi [AM88]	Orbit [YN84]
du nœud OU le plus ancien		•	•	•	•
du nœud OU le plus jeune	•	•		•	
stratégie mixte					•

Table IV.1: *Stratégies de régulation de charge dans les systèmes* OU *parallèles.*

IV.7.1 Régulateurs de charge du système *Aurora*

Aurora est un système OU multi-séquentiel développé initialement pour des architectures parallèles à mémoire partagée de type UMA (Sequent Symmetry). Le système a par la suite été adapté pour des architectures NUMA (BBN TC-2000) [LWH90].

Le modèle de calcul adopté par le système *Aurora* repose sur le modèle SRI [War87]. Dans ce modèle, un groupe d'*agents* appelés *workers* coopèrent à

l'exploration de l'arbre développé par l'évaluation du programme. Chaque *worker* est constitué de deux composants : le *travailleur* (*engine*) qui est une instance de la machine Prolog, et le *régulateur* (*scheduler*) qui alloue des traitements (parties de l'arbre) au *travailleur*. Ces deux éléments sont indépendants. L'interaction est assurée par une *interface* [SCY91]. Cette interface a été conçue dans le but d'évaluer différentes fonctions de régulation de charge sur différents modèles de gestion mémoire. Cinq fonctions de régulation ont été développées pour le système *Aurora*. Toutes utilisent la représentation de l'arbre OU et l'idée de partage des branches en sections publiques et privées. L'installation d'une tâche utilise la technique de partage des sections communes.

Le système utilise des annotations fournies par le programmeur sur les prédicats (parallélisable, non-parallélisable) pour éviter le transfert de tâches que l'on sait de faible granularité.

L'ordonnanceur d'*Argone* La stratégie adoptée dans ce régulateur est de minimiser le nombre de structures globales et de répartir les décisions de régulation. Chaque nœud public contient une indication de l'existence de tâches en suspens. Le *worker* oisif cherche périodiquement un nœud public ayant du travail [BDL$^+$88].

L'ordonnanceur de *Manchester* Le principe de ce régulateur est de donner du travail aux *workers* le plus tôt possible. Il a été proposé comme une optimisation de l'ordonnanceur d'*Argone*. La recherche de travail et l'installation de la tâche sont effectuées par le *worker* inactif. Lors de la recherche de travail, il sélectionne le point de choix le plus proche de la position où le processeur est devenu oisif. Cette heuristique vise à diminuer le temps de recherche. Le régulateur maintient des structures de données globales pour la gestion des *workers* oisifs et le nœud public de chaque *worker* actif [CS89].

Ordonnanceurs	1	4	8	16
Manchester [LWH90] (s)	29.18	7.31	3.69	1.95
Argone [LWH90] (s)	29.11	7.37	3.74	1.96
Wavefront [LWH90] (s)	29.12	7.32	3.78	2.08

Table IV.2: *Performances d'Aurora (8-reines) sur Sequent Symmetry.*

Le *Wavefront* Le *"front de la vague"* a pour principe de faciliter la recherche de travail en maintenant un chaînage entre les nœuds des sections publiques.

Lors de la recherche de travail, le point de choix le plus ancien (proche de la racine) est sélectionné [Bra88a]. Cette approche a été reprise dans [Kac94b].

L'ordonnanceur *Bristol* Le choix dans ce régulateur est de minimiser les surcoûts de régulation. L'idée consiste à rendre publique le plus tôt possible une séquence de nœud au lieu d'un seul. Lors de la recherche de travail, le processeur sélectionne une tâche à partir du point de choix le plus jeune. Une adaptation récente de ce régulateur a été développée pour la gestion du travail spéculatif [BRPS91].

L'ordonnanceur*Dharma* L'objectif de ce régulateur est la gestion efficace du travail spéculatif. Cette gestion équivaut à trouver rapidement la branche la plus à gauche n'ayant pas de travail spéculatif [Sin92].

Les mesures de performances *Aurora* sur un Sequent Symmetry à 20 processeurs [Car90] montrent que les différentes stratégies de régulation obtiennent des résultats sensiblement équivalents. Ceci remet en question l'emploi de techniques sophistiquées et complexes à mettre en œuvre.

IV.7.2 Régulateur de charge du système *Muse*

Muse est un système logique OU multi-séquentiel conçu pour des machines à mémoire partagée de type UMA (Sequent Symmetry, Sun Galaxy, BBN Butterfly II) [Ali88, Ali90, AKM91, AKM92a, AKM92b].

Le modèle de calcul considère un groupe de *travailleurs* (*workers*) coopérant à l'exécution du programme. Chaque *worker* est composé de deux éléments : une machine abstraite Prolog (extension d'une WAM séquentielle) et le régulateur (*scheduler*). La modularité du système est également assurée par une interface entre la partie opérative (machine Prolog) et la partie contrôle (régulation de charge) permettant ainsi la modification des stratégies de régulation.

Dans *Muse*, l'installation d'une tâche sur un processeur oisif utilise la technique de copie de contexte. Deux stratégies de régulation ont été développées [AK90, AK91]. La première est fondée sur la partition des nœuds de l'arbre en deux sections : une section contenant les nœuds OU publics et une section contenant les nœuds de travail (privés). Lorsque tous les nœuds publics sont épuisés, certains nœuds privés sont alors rendus publics. Le critère de proximité (en nombre de nœuds OU) est également celui retenu dans le modèle, dans le but de diminuer le coût d'installation des tâches. Cependant, à l'opposé du système *Aurora* où une seule alternative est transférée, le modèle permet d'accéder à un certain nombre de nœuds OU.

Le principe de la seconde stratégie consiste à sélectionner les alternatives à transférer, à partir du point de choix le plus jeune du processeur ayant cumulé le plus grand nombre de points de choix.

Le contrôle de granularité dans *Muse* est effectué de manière heuristique. Le programmeur n' a pas à rajouter des annotations. Le principe de l'heuristique de contrôle est le suivant : lorsque le processeur n'a plus qu'un seul nœud privé, ce nœud ne sera visible aux autres processeurs que s'il survit à une certaine durée (cette durée est modulable).

Les performances de ce système montrent des résultats très prometteurs en comparaison d'autres systèmes. *Muse* est approximativement 30% plus rapide que le système *Aurora* en utilisant l'ordonnanceur *Bristol* (celui-ci s'inspire de celui développé pour *Muse*).

IV.7.3 Régulateur de charge du système *Opera*

Opera est un système logique OU multi-séquentiel conçu pour des machines sans mémoire commune de type NORMA. Le système à été implanté sur une machine parallèle composée de 16 *Transputers* (le *Tnode*) [Gey91, Fav92].

Systèmes	1	2	4	8	12	16	Machines
Muse [Ali90] (ms)	17540	•	4419	2240	1510	•	S. Symmetry
Opera [Fav92] (ms)	8571	4397	2380	1319	1085	933	Tnode

Table IV.3: *Performances de Muse et Opera pour le problème des 8-reines.*

Comme dans *Muse*, le modèle de calcul d'*Opera* considère un groupe de *travailleurs* (*workers*) coopérant à l'exécution du programme. Chaque *worker* est composé de deux éléments : une machine abstraite Prolog TWAM (extension de la WAM séquentielle pour le transputer) et le régulateur (*scheduler*). Une stratégie fondée sur un seuillage du nombre de points de choix cumulés permet de répartir les processeurs en trois états : *oisif, surchargé et isolé*. L'état de charge du système est maintenu de manière centralisée sur un processeur dédié au contrôle de lancement des tâches parallèles. La stratégie de transfert de tâches repose sur la sélection des points de choix les plus anciens. Le système a été développé en tenant compte de certains aspects caractéristiques de la machine hôte (reconfiguration dynamique du réseau d'interconnexion), afin de minimiser les coûts induits par le transfert des tâches.

Les performances du système montrent des résultats très prometteurs pour des programmes Prolog présentant un comportement équilibré et comportant du parallélisme OU.

IV.8 Conclusion

Un des points clefs pour garantir que la parallélisation d'un programme Prolog ne conduise pas à une efficacité inférieure à une exécution séquentielle réside dans la mise en œuvre d'une fonction de régulation de charge appropriée.

La difficulté principale d'élaboration d'une telle fonction provient du caractère hautement dynamique de l'exécution des programmes. Cette propriété rend difficile la vérification des contraintes de parallélisation qui garantissent une augmentation de performances.

La mise en œuvre d'une fonction de régulation de charge efficace repose sur :

- la définition de bons estimateurs de temps de calcul des tâches,
- la prise en compte des caractéristiques architecturales de la machine cible,
- la définition d'une bonne stratégie de répartition des tâches applicable à tout programme Prolog.

Deux principales approches ont été menées conjointement pour la mise en œuvre de fonctions de régulation de charge efficaces :

- analyse de complexité des programmes,
- utilisation d'heuristiques fondées sur une représentation de l'arbre OU.

Nous avons choisi cette deuxième approche pour les raisons suivantes :

- les techniques de compilation permettant de pondérer les prédicats (alternatives) par une fonction de coût sont lourdes et complexes à mettre en œuvre. Leur efficacité dépend du bon modage et typage du programme nécessaires pour la pertinence des informations extraites ;
- Le calcul (l'évaluation) des fonctions de coût à l'exécution génère des surcoûts qui viennent s'ajouter aux coûts induits par la fonction de régulation. Cela est en contradiction avec la contrainte d'efficacité énoncée en section IV.3.2. Par conséquent, un compromis reste à trouver entre la pertinence de l'information de charge et les coûts induits pour l'obtenir.

Bien que les stratégies heuristiques actuelles ne permettent pas de garantir une augmentation de performance pour tous les programmes Prolog, elles affichent, pour des programmes comportant du parallélisme OU, des performances intéressantes et prometteuses. La suite du document est consacrée à la présentation et l'évaluation de notre fonction de régulation, dans le cadre de la conception et de la réalisation du système logique parallèle PLoSys.

Chapitre V

PLoSys : Modélisation et Plate-forme d'évaluation

V.1 Introduction

Alors que la plupart des travaux sur la programmation logique parallèle choisissent de travailler sur des machines à mémoire commune, nous nous sommes intéressés à la viabilité d'une implémentation sur une architecture à mémoire distribuée. Si l'accroissement des performances s'avère vérifiée, l'implantation de Prolog sur les différentes sortes d'architectures parallèles permettra d'exploiter une large gamme de machines de manière uniforme. Le but est de réussir à faire fonctionner le même programme (sans aucune modification) sur un seul ou sur un nombre donné de processeurs, selon l'accroissement de performance souhaité et/ou l'accroissement de la complexité des programmes traités : C'est l'objectif du projet PLoSys.

La mise en œuvre d'un système qui permette d'exploiter de façon automatique le parallélisme inhérent aux programmes Prolog et de garantir un accroissement de performance impose, dans le cadre de l'implémentation actuelle sur les machines à mémoire distribuée, des défis non-négligeables. Elle fait d'ailleurs toujours l'objet d'une recherche active. L'un de ces enjeux et certainement l'un des plus importants est le développement des techniques de régulation de charge, aux fins de guider l'exécution parallèle des programmes Prolog.

La première partie de ce chapitre est consacrée à la présentation des choix retenus pour la mise en œuvre du système PLoSys. Dans la deuxième partie nous proposons une fonction de régulation de charge et nous étudions son comportement sur deux classes de comportement de programmes Prolog. La dernière partie de ce chapitre est entièrement consacrée à la présentation du modèle et de l'environnement d'évaluation que nous avons développés pour l'étude expérimentale de notre fonction de régulation.

V.2 Régulation de charge dans *PLoSys*

Un système logique parallèle est défini à partir des caractéristiques du langage qu'il supporte, de son modèle d'exécution, du type d'architecture parallèle ciblée et de sa stratégie de contrôle de l'exécution parallèle.

PLoSys se place dans la catégorie des systèmes adoptant un modèle d'exécution multi-séquentiel sur des machines parallèles sans mémoire commune (architecture de type NORMA), exploitant automatiquement le parallélisme OU inhérent à la sémantique des programmes Prolog *'pur'*.

V.2.1 Le modèle *PLoSys*

Le modèle OU multi-séquentiel de PLoSys est inspiré de celui d'OPERA, sans toutefois reprendre les optimisations d'implantation liées aux caractéristiques de l'architecture cible initiale (le TNODE). Le modèle de calcul correspond à un schéma type d'algorithme parallèle dit *ferme de processus* (*process farm*). Le principe de ce schéma est d'avoir plusieurs unités de travail (processus) partageant le même code et coopérant entre elles à l'exécution du programme. Le système est alors perçu comme constitué de deux parties :

- une partie opérative,
- une partie contrôle.

La partie opérative est constituée d'un ensemble d'unités de travail (processeur). Chaque unité de travail se décompose en une unité de traitement du programme Prolog (le *Travailleur*) et une partie contrôle qui gère la coopération des unités de travail dans le but d'optimiser le temps d'exécution du programme. Le rôle de la partie contrôle est d'assurer la fonction de régulation de charge entre les différentes unités de travail (le *Régulateur*).

a) Le travailleur

Les processeurs (unités de travail) sont des machines abstraites Prolog séquentielles identiques. L'évaluation du programme au niveau de chaque Travailleur (processeur) s'effectue selon la stratégie classique de Prolog (*en profondeur d'abord puis de la gauche vers la droite*). Chaque unité de travail est une adaptation de la machine abstraite de *Warren* [AK92a] (la WAM), décrite en section II.5, pour faciliter l'exploitation du parallélisme. Les piles **locale** et **globale** ont été réorganisées en quatre piles. Les extensions apportées au modèle dans PLoSys concernent principalement les structures de données manipulées dans la WAM afin de :

- permettre de profiter de l'évolution des techniques de compilation séquentielles de Prolog ;
- faciliter l'implantation du modèle sur différentes plates-formes ;
- faciliter le transfert des portions de calcul ;
- permettre une meilleure gestion de la mémoire ;
- permettre l'implantation des effets de bord.

Le lecteur intéressé par une description détaillée concernant ces travaux peut se reporter aux documents [Gey91, Fav92, BFGdK92, Mor96].

b) Le régulateur

La partie contrôle dans le modèle de calcul PLoSys gère la coopération des unités de travail pour l'exécution d'un programme Prolog, dans le but d'optimiser son temps d'exécution. Ce contrôle est mis en œuvre par une fonction de régulation de charge qui a pour objectifs :

- l'extraction automatique du parallélisme OU ;
- le transfert et l'initialisation de portions de calcul sur les unités de travail ;
- la garantie qu'un accroissement du parallélisme ne conduise pas à une baisse d'efficacité.

L'extraction du parallélisme met en évidence les portions de calcul pouvant être traitées en parallèle. Notre choix a été d'adopter le parallélisme implicite afin d'une part, de dégager l'utilisateur du problème de gestion du parallélisme et, d'autre part, d'exploiter les programmes existants. Les portions de calcul échangées entre ces unités représentent le traitement des branches disjointes de l'arbre correspondant au programme exécuté (parallélisme OU).

L'élaboration d'une fonction de régulation dynamique de charge pour le système PLoSys requier la mise en œuvre des fonctionnalités suivantes (cf. section IV.2) :

- Un gestionnaire de l'état de charge du système à partir duquel on établira des décisions de régulation (cf. section IV.5).
- Un mécanisme de transfert des calculs entre les processeurs de l'architecture (cf. section IV.4).
- Un algorithme de régulation de charge (cf. sections IV.2.2 et IV.6).

La suite de notre étude est consacrée à la présentation et à l'évaluation des choix que nous avons retenus pour la mise en œuvre de ces fonctionnalités dans le cadre de la réalisation du système PLoSys.

c) Le schéma de coopération

Le principe de coopération est le transfert de la portion de calcul d'une unité surchargée vers une unité sous-chargée. La gestion de cette coopération est assurée par une fonction de régulation de charge qui contrôle le parallélisme. Un processeur physique peut supporter plusieurs unités de travail (multiprogrammation). Cependant, pour des raisons de simplicité et d'efficacité [Fav92], nous avons choisi d'assimiler la notion d'unité de travail (logique) à la notion de processeur (physique). Chaque unité de travail (processeur) représente une machine Prolog séquentielle.

L'exécution du programme commence sur un seul processeur par l'évaluation du but initial, tandis que les autres processeurs sont oisifs et tentent d'acquérir du travail. L'espace de travail sur chacun des processeurs P_i est matérialisé par une pile de *points de choix*. Un point de choix représente les alternatives pendantes (en attente d'évaluation) d'un nœud OU de l'arbre d'exécution. La profondeur de la pile correspond au nombre de points de choix cumulés. Lorsqu'un processeur termine l'exploration de son espace de travail, il tente d'en acquérir un autre à partir d'autres processeurs via la partie contrôle.

V.2.2 Evaluation de la charge du système

Compte-tenu des limitations dues à la complexité intrinsèque du problème de l'évaluation de la charge exacte d'une unité de travail, nous avons choisi de définir la charge à partir de paramètres évaluables dynamiquement.

a) Evaluation de la charge d'un *travailleur*

Vis à vis d'une unité de travail, chaque processeur se comporte comme un producteur et un consommateur de points de choix. Pour un programme Prolog ρ, le temps total nécessaire à son évaluation séquentielle est une fonction croissante des points de choix créés lors de son exécution. C'est ce qui apparaît clairement (cf. figures V.1 et V.2) lors des mesures[1] que nous avons effectuées sur des petits programmes Prolog (cf. section VI.3).

L'état de charge visible d'un processeur (unité de travail) à un instant donné t peut donc être représenté par le nombre de points de choix cumulés à cet instant.

[1] *Ces mesures ont été effectuées sur un* SUN4 *en utilisant le système* WAMCC *[Dia94].*

Programmes	Nb points de choix	temps d'exécution (sec)
4-reines	69	< 0.05
6-reines	531	0.05
8-reines	7589	0.1
10-reines	145569	1.21
12-reines	3771957	33.35

Table V.1: *X-reines : Nombre de points de choix Vs. Temps d'exécution.*

Programmes	Nb points de choix	temps d'exécution (sec)
4-gasp	2099	0.05
6-gasp	22341	0.1
8-gasp	179653	0.3
10-gasp	1210309	4.46

Table V.2: *X-gasp : Nombre de points de choix Vs. Temps d'exécution.*

b) Maintien de l'état de charge du système

Le système est constitué de N processeurs qui participent à une résolution d'un même problème. Chaque processeur est caractérisé par une charge locale. L'ensemble des N charges constitue la charge globale du système.

Nous avons choisi dans un premier temps d'organiser notre fonction de régulation de charge de manière centralisée pour les raisons suivantes :

- la facilité de conception et de mise au point ;
- la bonne adaptation de la plate-forme d'évaluation aux caractéristiques architecturales de la machine cible *(le Méganode)* ;
- la conception d'un contrôle distribué sur une architecture sans mémoire commune est complexe à mettre en œuvre.

Bien que cette solution (processeur de contrôle dédié) puisse provoquer un goulot d'étranglement, on considère que, pour des petites configurations de machines (de l'ordre d'une dizaine de processeurs), ce risque est négligeable (tant que l'on n'a pas noté une chute des performances).

L'état de charge du système est maintenu de manière centralisée sur un processeur dédié que l'on appelle *contrôleur*. Chaque processeur évalue sa charge à

partir d'un échantillonnage périodique du nombre de points de choix cumulés. Le *régulateur* est informé d'un changement d'état lorsque la variation de charge sur un processeur dépasse une certaine valeur.

Du fait des fluctuations intermittentes et rapides du nombre de points de choix sur chaque processeur et des délais de communication, l'état de charge observable à un instant donné du système est donc un état approché. Maintenir un état exact exigerait une synchronisation à chaque modification partielle survenue sur chaque processeur. Cette approche synchrone n'a pas été retenue pour des raisons d'efficacité et d'inadéquation au type d'architectures ciblées. Il faut donc prévoir que les décisions de régulation peuvent échouer ou être erronées au moment de la prise de décision.

V.2.3 Mécanisme de transfert de la charge

La décision adoptée concernant le mécanisme de transfert du travail entre unités de travail a été le choix d'une technique de copie au détriment du partage des sections communes des piles et d'une technique de duplication des calculs. Ce choix est justifié par :

- l'absence d'une mémoire commune ;
- les machines à mémoire distribuée ont un temps d'amorçage de communication important pénalisant la mise en œuvre de mécanismes d'accès non-locaux ;
- la gestion d'une mémoire virtuelle partagée est un problème complexe ;
- la technique de duplication du calcul est complexe à mettre en œuvre.

Bien que restrictive, cette solution permet une plus grande portabilité du système sur des architectures différentes. En effet, il est plus facile de ne pas utiliser d'échange par mémoire sur une architecture à mémoire partagée que de simuler une mémoire commune sur une machine n'en ayant pas.

Le coût de transfert d'une tâche dépend du volume des données à copier (cf. section IV.4). Les performances globales du système sont très dépendantes des performances :

- du médium de communication ;
- des mécanismes (techniques) de transfert (routage) ;
- de la localité de données ;
- de la fréquence des échanges entre unités de travail ;

Une stratégie de régulation doit tenir compte de ces paramètres.

V.2.4 Stratégies de régulation de charge

La régulation de charge du système consiste donc à répartir dynamiquement les points de choix entre les processeurs, de façon à maintenir équilibrée la charge des processeurs.

Un critère de sélection qui repose sur un double seuillage (k_{min}, k_{max}) du nombre de points de choix cumulés permet de répartir les processeurs en trois catégories :

- ***oisif*** : le nombre de points de choix cumulés est inférieur au seuil k_{min},
- ***surchargé*** : le nombre de points de choix cumulé est supérieur au seuil k_{max},
- ***isolé*** : le nombre de points de choix est compris entre k_{min} et k_{max}.

Le volume de données à transférer est une fonction strictement croissante de la profondeur des points de choix sélectionnés pour une exportation. La minimisation du coût de transfert dépend donc du critère de sélection de la tâche à transférer. On rencontre souvent dans la littérature deux approches de sélection. (cf. Section IV.3.2) :

- Sélection d'une tâche à partir du point de choix le plus ancien *(top-most scheduler cf. figure IV.7.a)* ;
- Sélection d'une tâche à partir du point de choix le plus jeune *(bottom-most scheduler cf. figure IV.7.b).*

La taille mémoire nécessaire à l'évaluation du point de choix le plus ancien est plus petite que celle nécessaire à l'évaluation de tout autre point de choix. On choisit donc, pour minimiser le coût de transfert, de sélectionner la tâche à exporter à partir du fond de la pile des points de choix. Pour cela, les points de choix sont pondérés par une date de création qui correspond à leur profondeur dans l'arbre de recherche. Cela permet de répartir les processeurs surchargés en des classes correspondant à la profondeur de leur point de choix le plus ancien.

L'heuristique que nous adoptons, pour tenter de satisfaire la contrainte minimale de parallélisation (IV.3 et IV.4), consiste à sélectionner un sous-ensemble non vide de points de choix sur le processeur le plus surchargé (à partir du point de choix le plus ancien) et de le transférer sur le processeur oisif. On peut transférer toutes les alternatives des points de choix sélectionnées ou seulement une partie. Cette heuristique repose sur le fait que plus on est haut dans l'arbre plus la granularité de la tâche exportée est susceptible d'être importante.

V.3 Incidence des paramètres de régulation

Trois paramètres principaux régissent le fonctionnement de notre stratégie de régulation dynamique de charge :

- La fréquence d'échantillonnage pour l'évaluation de l'état de charge d'un processeur ;
- Le nombre de points de choix exportés à chaque opération de transfert ;
- Les valeurs de seuillage de l'indice de charge.

Les valeurs de ces paramètres déterminent l'efficacité de notre stratégie. Cette instanciation dépend des caractéristiques d'exécution du programme Prolog, à savoir :

- la vitesse de production et de consommation des points de choix ;
- le nombre de points de choix générés par l'exécution du programme ;
- La structure de l'arbre d'exécution.

Examinons l'influence de chacun des paramètres sur l'efficacité de la fonction de répartition dynamique de charge.

V.3.1 La fréquence d'échantillonnage

L'évaluation de la charge sur chaque processeur est invoquée de manière périodique. L'augmentation de la fréquence d'échantillonnage permet d'éviter une trop grande dérive de l'état global vu par le régulateur par rapport à l'état global réel. Cela permet aussi de limiter le nombre des décisions de régulation inappropriées du fait de l'état approché sur lequel est fondé le régulateur. Cependant, cette augmentation de fréquence a pour inconvénient de surcharger le processeur.

D'un autre coté, une fréquence d'échantillonnage faible peut engendrer une transition d'état directe du processeur, de l'état surchargé à l'état oisif. Ceci n'est pas désirable du fait que l'indice de charge ne reflète plus vraiment l'état de charge courant d'un processeur. Le problème consiste alors à déterminer une période d'échantillonnage appropriée.

Le surcoût d'estimation périodique de la charge est augmenté du coût de calcul de la variation de charge à chaque échantillonnage. Ce dernier a pour objectif d'éliminer les micro-variations de la charge instantanée au moyen d'une fonction de lissage, laquelle est une moyenne pondérée de l'ancienne valeur de charge et de la charge courante. L'invocation du régulateur pour la mise à jour

de l'état de charge n'est réalisée que si la variation de la charge dépasse un seuil donné. Si ce seuil est petit, cela implique une mise à jour fréquente. Cela permet aussi de prendre de meilleurs décisions de régulation. En revanche, un petit seuil augmente le coût de communication dans le système. Un compromis doit être établi alors entre la fréquence de mise à jour de l'état de charge d'un processeur et le degré de tolérance d'une information de charge périmée.

V.3.2 Contrôle de la granularité

La stratégie de choix d'un sous-ensemble d'alternatives à partir du fond de la pile de points de choix du processeur surchargé tend à vérifier la contrainte (IV.3). Intuitivement, pour vérifier la contrainte (IV.4), l'idéal serait de partager équitablement la charge de travail disponible entre le processeur surchargé et le processeur oisif.

En effet, si la portion de travail transférée est trop petite, le processeur redeviendra oisif rapidement, et si celle-ci est trop importante le processeur qui était surchargé risque de manquer très vite de travail.

Si l'on considère que $I_T = \alpha.E_T$, alors les contraintes (IV.3 et IV.4) sont augmentées de la contrainte suivante exprimant le grain de parallélisme maximum exploitable (cf. section IV.3.1) :

$$D_T = \frac{D_r(t) - (\alpha - 1)E_T}{2}. \quad \text{(V.1)}$$

N'ayant pas un critère d'évaluation efficace de la quantité de travail $D_r(t)$ disponible sur un processeur surchargé à l'instant t, notre approche consiste à adopter une méthode heuristique fondée sur la sélection d'un nombre de points de choix à exporter à partir du nombre de points de choix cumulés sur le processeur surchargé à cet instant. On peut donc transférer un ou plusieurs points de choix au cours d'un même appariement[2]. Cette factorisation de transfert a pour avantage d'augmenter le granularité de la tâche exportée. Par contre, la taille des données transférées est plus importante, car il faut transférer toutes les portions de piles (contextes d'exécution) correspondant aux points de choix exportés. Un compromis doit donc être établi entre l'augmentation de la tâche à exporter et le coût nécessaire à son transfert.

V.3.3 Seuillage de l'indice de charge

Dans notre approche, nous avons choisi comme indice de mesure de la charge le nombre de points de choix en attente d'évaluation. L'avantage de ce choix est que cette valeur est rapidement évaluable et se corrèle souvent avec le taux

[2] *Appariement d'un processeur surchargé et d'un processeur oisif.*

d'utilisation du processeur (cf. section V.2.2). Deux seuils k_{min} et k_{max} permettent de déterminer le régime de fonctionnement de chaque processeur à partir du nombre de points de choix cumulés sur chacun d'eux.

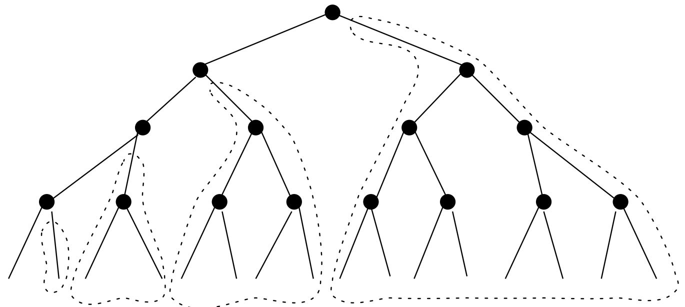


Figure V.1: *Arbre équilibré complet.*

L'utilisation de ces deux seuils permet une grande souplesse pour la configuration du système.

Considérons un programme Prolog ρ. Son exécution consiste alors à parcourir un arbre de recherche A. Chaque nœud interne de A est caractérisé par un facteur de branchement moyen b (nombre d'alternatives moyen d'un point de choix).

Supposons que le transfert d'une alternative d'un processeur à un autre nécessite en moyenne M unités de temps . Ce coût est contracté par les deux processeurs (importateur et exportateur).

Supposons de plus que la profondeur maximale d'un nœud (point de choix) est h et que l'on dispose d'autant de processeurs P que l'on veut.

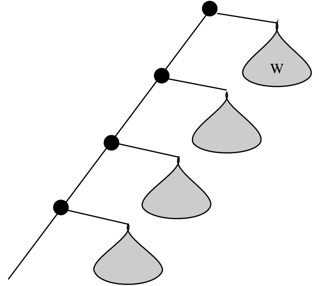


Figure V.2: *Arbre dégénéré.*

Examinons maintenant l'influence des valeurs attribuées au seuillage de l'indice de charge sur la fonction de répartition. Pour cela, nous considérons deux types d'arbres : le premier (cf. figure V.1) est un arbre équilibré complet alors que le second (cf. figure V.2) est un arbre dégénéré.

Cas 1 : $k_{min} = k_{max} = 0$. Le processeur fonctionne comme un distributeur d'alternatives. A chaque création d'un point de choix, les alternatives en suspens sont transférées sur un processeur oisif. Etant donné que le nombre maximum de tâches pouvant s'exécuter en parallèle correspond au nombre de feuilles de l'arbre ($Nb_{tâches} = b^{h+1}$), on a au maximum ($b^{h+1} - 1$) transferts pour un arbre équilibré de la figure V.1.

Le temps d'exécution parallèle peut donc être borné par :

$$T_{exéc_par_max} = T_{séq} + (b^{h+1} - 1)M. \quad (V.2)$$

qui correspond au temps de parcours séquentiel de l'arbre, augmenté du coût de transfert des ($b^{h+1} - 1$) tâches.

Si l'on considère que T est le temps de calcul maximum de la branche la plus longue de l'arbre et que la profondeur maximale d'un point de choix de cette branche est h, alors le temps d'exécution parallèle optimal du programme est borné par :

$$T_{exéc_par_opt} = T + (h+1)(b-1)M. \quad (V.3)$$

qui correspond au temps de calcul de la branche la plus longue augmenté du coût de transfert des alternatives pendantes à chaque nœud.

Si l'on considère maintenant le cas d'un arbre de la figure V.2, où chaque branche droite d'un nœud correspond à un coût de traitement w. Si $(w < M)$, alors à chaque transfert correspondra une perte de performance $(M - w)$ par rapport à une exécution séquentielle. Si la profondeur maximale d'un point de choix est h, alors le temps d'exécution parallèle sera au moins :

$$T_{exéc_par} = T_{séq} + (h+1)(M - w). \quad (V.4)$$

Cas 2 : $k_{min} = 0$; $k_{max} = k$ *avec* $0 < k < h$. Un processeur surchargé n'amorce une répartition de sa charge que s'il a cumulé au moins k points de choix. Nous supposons que le processeur surchargé transfère toutes les alternatives pendantes d'un seul point de choix lorsque ce seuil est atteint.

Considérons d'abord le cas d'un arbre complet (cf. figure V.1). Le nombre de points de choix créés est borné par :

$$Nb_{Pchoix} = \sum_{i=0}^{h} b^i = \frac{b^{h+1} - 1}{b - 1}. \quad (V.5)$$

Si pour chaque k point de choix créé, un transfert est effectué, alors le temps d'exécution parallèle peut être borné par :

$$T_{exéc_par_opt} = T_{séq} + M\frac{b^{h+1}-1}{k(b-1)}. \quad \text{(V.6)}$$

En tenant le même raisonnement que dans le cas précédent, le temps d'exécution parallèle optimum avec un seuillage k serait :

$$T_{exéc_par_opt} = T + M\frac{h+1}{k}. \quad \text{(V.7)}$$

Si l'on considère le cas de l'arbre de la figure V.2, à chaque création de k points de choix, un transfert engendre une perte de performance $(M - w)$ par rapport à une exécution séquentielle. Donc le temps d'exécution parallèle sera au moins :

$$T_{exéc_par} = T_{séq} + (M - w)\frac{h+1}{k}. \quad \text{(V.8)}$$

Cas 3 : $k_{min} = 0$; $k_{max} = h$**.** Dans ce cas, le processeur ne participe pas à la répartition de charge. Cela correspond à une exécution séquentielle du programme (ou de la partie de l'arbre de recherche traitée sur ce processeur).

V.3.4 Bilan

L'analyse des effets du seuillage de l'indice de charge montre que l'algorithme de régulation dans le cas 1 appliqué à un arbre équilibré effectue une répartition rapide de la charge de travail sur l'ensemble des processeurs. Si l'on suppose un nombre fini de processeurs, cela signifie que tous seront très vite actifs et que le transfert de tâches sera souvent efficace. Dans le cas 2, l'algorithme impose un délai avant d'entreprendre la répartition de la charge de travail. Ce délai est le temps nécessaire à l'accumulation de k points de choix. Cependant, il a un meilleur comportement que le premier lorsque l'on s'approche des feuilles de l'arbre.

Appliqué à un arbre dégénéré, l'algorithme du cas 2 a aussi un meilleur comportement que le cas 1. En effet, si l'on suppose que k est fixé de sorte à effectuer au moins M unités de traitement avant de transférer une tâche, alors l'algorithme du cas 1 est de l'ordre de M fois plus inefficace que celui du cas 2.

La perte de performance dans ces deux cas est due au fait que le coût de transfert d'une tâche est supérieur à son coût de traitement ou, réciproquement, que le coût de transfert et de traitement de la tâche est plus important que le coût de traitement restant sur le processeur source du transfert. Dans un cas, la contrainte (IV.4) n'est pas satisfaite et dans l'autre, c'est la contrainte (IV.3) qui ne l'est pas.

N'ayant pas de moyen d'évaluer le coût potentiel d'une tâche, le seul critère qui nous permette de gérer la répartition reste le contrôle des paramètres de seuillage de l'indice de charge. Nous avons montré comment la valeur de ces seuils dépend fortement à la fois de la structure de l'arbre d'exécution et du coût de transfert de tâches. Il est donc nécessaire de pouvoir adapter dynamiquement ces paramètres en fonction de l'évolution de l'arbre d'exécution. Nous avons développé pour cela une plate-forme qui permet d'une part, de raffiner l'étude d'une stratégie OU-parallèle et d'autre part, d'investiguer cet aspect afin de déterminer dans quelle mesure il est possible d'adapter (de configurer) les paramètres de notre fonction de régulation.

V.4 Modéliser pour évaluer

Trois approches peuvent être utilisées pour l'évaluation des systèmes parallèles : par une modélisation analytique, par la simulation, ou à partir d'une implantation réelle du système sur une architecture donnée.

Dans une ***modélisation*** analytique, le langage d'abstraction est le langage mathématique. Pour les modèles analytiques en général, différentes théories mathématiques sont employées : la théorie de probabilités et des processus stochastiques, la théorie des files d'attente [GST94], les réseaux de Pétri et les réseaux d'automates. On obtient les indices de performance en résolvant les modèles, soit par des techniques algébriques, soit par des techniques numériques. La description du modèle en soit n'est pas coûteuse, mais sa résolution peut l'être. Cependant, la modélisation devient plus complexe si l'on veut se rapprocher de la réalité pour obtenir des indices de performance plus précis.

La ***simulation*** consiste à reproduire le comportement du système parallèle, à l'aide d'événements provenant de mesures sur une charge réelle (de vrais programmes exécutés sur une machine séquentielle) ou modélisées (un modèle de programme). Les contraintes imposées sur cette *re-exécution* définissent la précision de la simulation. En générale, la simulation apporte des résultats plus complets que la modélisation analytique. Cependant, la simulation est faite par logiciel et peut être très coûteuse en temps et en utilisation de la mémoire.

La troisième approche considère que la machine parallèle est disponible et que l'on peut effectuer des mesures. La charge (l'application à traiter) peut être réelle ou synthétique. A partir d'une application parallèle (réelle ou synthétique), on fait des mesures du système en utilisant des moniteurs (outils de prise de traces). Les moniteurs fournissent des traces qui sont traitées afin de donner des indices de performance. Les indices de calcul sont plus proches de la réalité que ceux obtenus par les modèles analytiques et par des outils de simulation. L'atout principal d'une exécution sur une vraie machine parallèle est que les traces

obtenues tiennent compte de la surcharge *(overhead)* imposée par la machine et le système d'exploitation. Cette surcharge est difficile à modéliser de façon analytique ou dans une simulation. Néanmoins, cette technique engendre une surcharge supplémentaire due à la génération des traces. Il est nécessaire dans ce cas de limiter ces perturbations ou de pouvoir les décompter lors de l'analyse de traces. Nous avons opté pour cette troisième approche pour l'étude, la conception et l'évaluation d'une fonction de régulation dynamique de charge adaptée à notre système logique parallèle *PLoSys*.

V.5 Technique de modélisation

Rappelons que l'évaluation d'un programme Prolog consiste en un parcours exhaustif d'un arbre *ET/OU*. Chaque nœud ET correspond à une sélection de sous-buts et contient la résolvante courante. Il représente une conjonction des sous-buts. La racine de l'arbre est un nœud ET dit but initial. Chaque nœud OU regroupe un ensemble d'alternatives. L'objectif de notre modélisation est d'**émuler** *(simuler)* ce comportement sur une architecture parallèle réelle afin de permettre l'étude et l'évaluation (mise au point) des caractéristiques de notre fonction de régulation.

V.5.1 Principe de modélisation

Notre approche consiste à modéliser l'exécution d'un programme Prolog "pur" ρ par un graphe de tâches acyclique *(PLoSys_DAG)* $G(T, A)$ défini par l'ensemble T des n nœuds du graphe ($T = \{t_0, ..., t_{n-1}\}$) et l'ensemble A des arcs dirigés du graphe. Chaque nœud t_i ($i \in [0...n-1]$) du graphe G modélise une tâche synthétique [KKMB94, KMB95, Kan95].

Définition V.1 *Une **tâche** synthétique représente un traitement d'une séquence de nœuds* ET *entre deux nœuds* OU*, ou entre un nœud* OU *et une feuille de l'arbre. La tâche racine représente le traitement effectué entre le but initial et la création du premier point de choix. Chaque tâche synthétique a trois paramètres : un coût de traitement synthétique δ_i, un coût mémoire μ_i, le nombre de ses tâches filles ω_i.*

Le coût de traitement synthétique δ_i d'une tâche t_i représente la granularité de la tâche. Il modélise un quantum d'exécution (nombre d'unités de temps d'occupation cpu). Ce coût est une fonction de la puissance de calcul du processeur émulé et du nombre d'opérations effectuées par la tâche.

Le second paramètre μ_i modélise la taille des données à transférer si l'exécution de la tâche t_i n'est pas effectuée par le processeur créateur. Elle représente le

contexte d'exécution de la tâche. Ce coût correspond à la taille des piles à l'instant de la création du point de choix.

Enfin, le dernier paramètre ω_i représente le nombre d'alternatives issues de la création du point de choix. Il est défini par un ensemble d'arcs (orientés) sortants de la tâche t_i. Chaque tâche fille t_j est connectée à t_i par un arc orienté a_{ij}. Chaque arc a_{ij} définit une relation de précédence temporelle entre la tâche t_i et t_j.

V.5.2 Extraction du graphe de tâches

Afin de modéliser et d'analyser différents comportements de programmes, nous avons élaboré trois approches de construction de graphe de tâches [Kan94] :

- à partir d'une analyse de trace post-mortem d'une exécution réelle d'un programme Prolog ρ (cf. figure V.3),
- à partir d'un programme Prolog *synthétique*,
- à partir d'un générateur automatique.

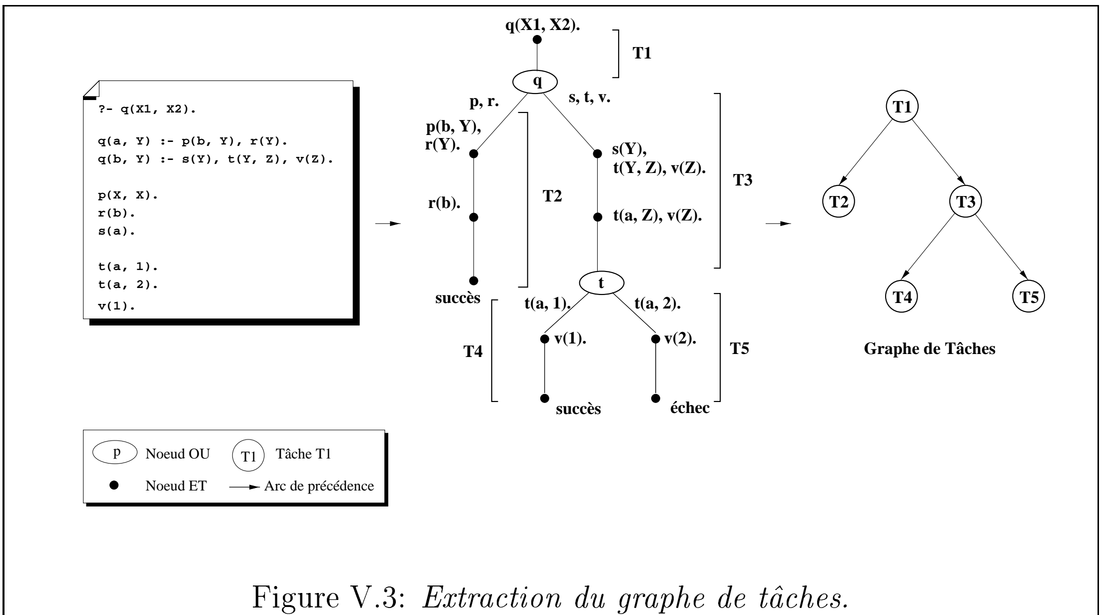


Figure V.3: *Extraction du graphe de tâches.*

L'analyse d'une trace d'exécution permet la construction du graphe de tâche correspondant à une exécution réelle d'un programme Prolog ρ sur une instance donnée π.

Un programme Prolog *synthétique* est obtenu à partir d'une transformation d'un programme Prolog réel. La transformation porte sur les termes Prolog qui,

dans les programmes synthétiques, sont dénués de paramètres. Cela préserve le mécanisme de résolution, seules les phases d'unification sont écartées. On obtient alors un *squelette* d'exécution du programme réel.

La dernière approche permet la construction automatique de graphes de tâches à partir de paramètres caractéristiques donnés en entrée. Quatre paramètres régissent cette construction : la profondeur maximale du graphe h, le facteur de branchement moyen de chaque tâche b, le coût de traitement synthétique δ_i et le coût mémoire μ_i de chaque tâche t_i. Le facteur b détermine le nombre de tâches filles moyen ω_i pour chaque tâche t_i. Les coûts (δ_i, μ_i) peuvent être [Kan94] :

- constants pour toutes les tâches ;
- représentés par une distribution aléatoire ;
- représentés par une fonction dont les paramètres sont des attributs du graphe (la profondeur de la tâche par exemple).

Chaque tâche $t_i(\delta_i, \mu_i, \omega_i)$ du graphe obtenu G a un identificateur unique (un entier $i \in [0...n-1]$). Il existe un ordre partiel (noté $\prec$) entre les tâches de l'ensemble T défini par les dépendances présentes dans l'exécution du programme Prolog.

$$\forall a_{ij} \in A, \quad \exists\, t_i, t_j \in T : \; t_i \prec t_j \tag{V.9}$$

Cette relation correspond à une dépendance temporelle. Elle modélise également une dépendance de données entre chaque tâche. Un arc a_{ij} de l'ensemble A indique que l'exécution de la tâche t_j débute uniquement après la fin d'exécution de la tâche t_i.

V.5.3 Avantages et limites du modèle

L'avantage principal de notre approche, par l'utilisation des tâches synthétiques, est d'offrir une grande flexibilité dans la manipulation des paramètres modélisant le graphe de tâches, ce qui permet ainsi la simulation de différents comportements de programmes. Cette caractéristique peut s'avérer utile, au moment d'analyser l'incidence de la granularité des tâches et des coûts de communication sur la performance d'une fonction de régulation de charge :

- en modifiant le coût de traitement δ_i de chaque tâche t_i, on joue sur la granularité des parties de l'arbre d'exécution ;
- il est également utile de modifier le coût de communication μ_i en vue de tester l'influence de la communication sur une topologie de réseau donnée ;

- il est vrai aussi que le fait de changer, dans le générateur automatique de graphe de tâches, la profondeur du graphe de tâches maximum h, ainsi que le facteur de branchement moyen b, permet de construire des structures d'arbres (équilibrés ou dégénérés) variées modélisant ainsi différents comportements de programmes.

Ce modèle peut également être utilisé pour émuler les caractéristiques de différentes machines. Ces changements sont guidés par le *calibrage* des valeurs de paramètres de tâches, selon les caractéristiques de la machine cible émulée. Par exemple, changer le coût de traitement δ_i, d'une part, revient à modéliser différents types de processeurs. Changer le coût de mémoire μ_i, d'autre part, équivaut à modéliser la relation entre le coût de communication et la bande passante du réseau.

L'inconvénient majeur de cette approche réside dans le fait que le graphe de tâches correspondant à l'exécution d'un programme Prolog donné est très large, entraînant ainsi une forte consommation de mémoire. La prise en compte de la gestion des effets de bord nécessite une extension du modèle afin de modéliser les possibilités d'élagage de l'arbre. Cet aspect n'a pas été traité dans notre étude.

V.6 Environnement d'évaluation

Notre environnement d'évaluation s'inspire des méthodes utilisées dans la plate-forme ALPES *(ALgorithmes Parallèles et Evaluation de Systèmes)* [KP94] initialement développée pour l'évaluation de stratégies de placement statique. Les extensions que nous avons apportées ont pour objectif de supporter l'évaluation quantitative de fonctions de régulation dynamique de charge pour le modèle de calcul PLoSys.

La figure V.4 présente les principaux éléments que nous avons développés pour la mise en œunvre de notre plate-forme d'évaluation.

V.6.1 Description de l'architecture matérielle

Notre plate-forme est actuellement implantée sur un *Méganode*[3] dont les 128 processeurs de calcul sont des *Transputers* T800. Le *Méganode* est une machine parallèle à mémoire distribuée, développée dans le cadre du projet Esprit 1085 [HJM86]. Cette machine est de la gamme des *Tnode* (module de base de la famille des machines *Supernode*, cf. figure V.5). Ses principales caractéristiques sont :

[3] *L'exemplaire utilisé dans le cadre de cette thèse est le Méganode du Laboratoire de Modélisation et de Calcul à l'Institut de Mathématiques Appliquées de Grenoble (LMC/IMAG).*

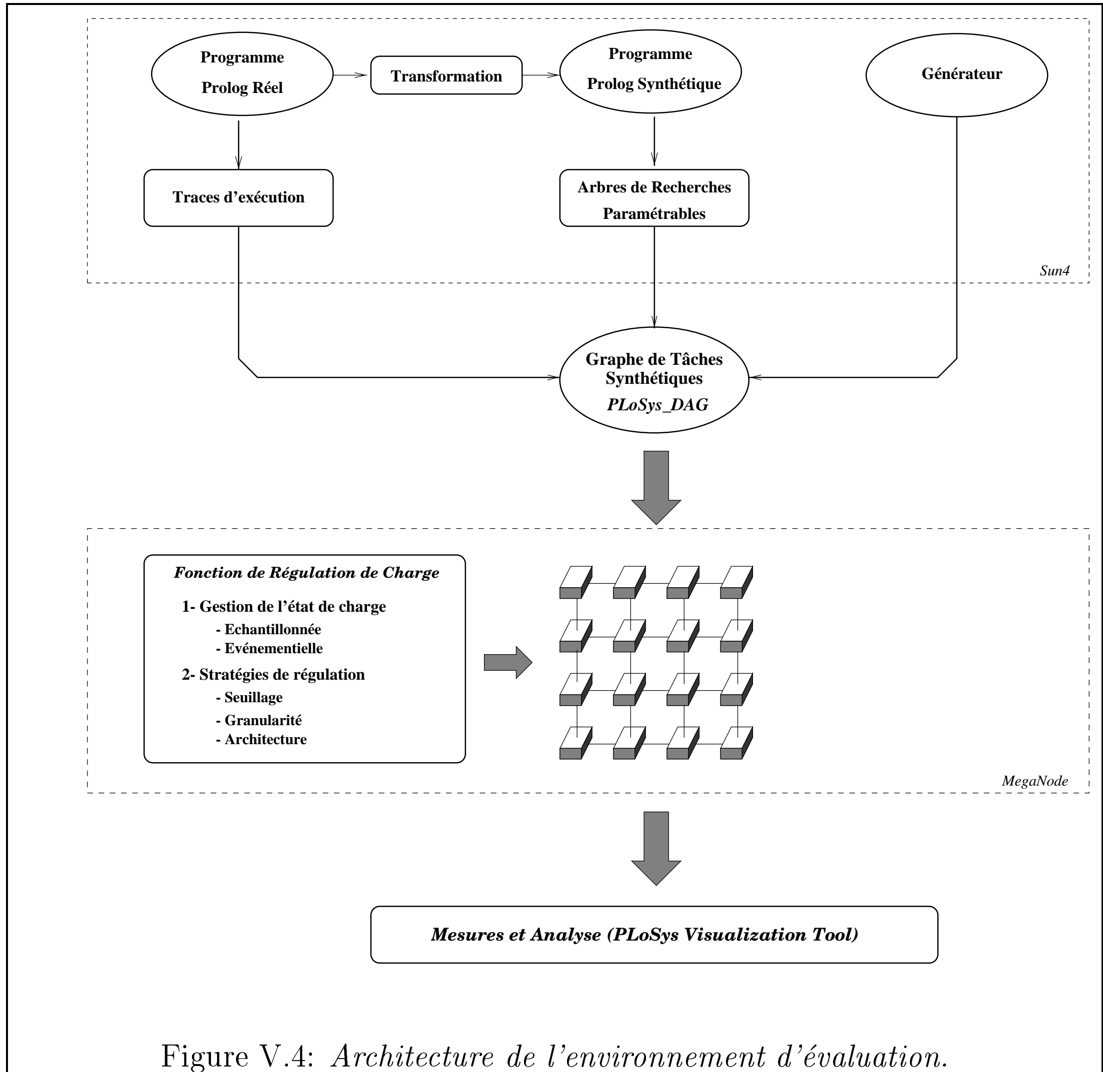


Figure V.4: *Architecture de l'environnement d'évaluation.*

- la communication inter-processeurs est inspirée du modèle des processus communicants *(CSP)* [Dij68] ;
- le réseau d'interconnexions (graphe de degré 4) est dynamiquement configurable ;
- une voie de contrôle offre une communication additionnelle entre les processeurs de travail et un processeur de contrôle ;
- l'architecture est modulaire et hiérarchisée ;
- on peut réaliser toute topologie de degré inférieur ou égal à 4.

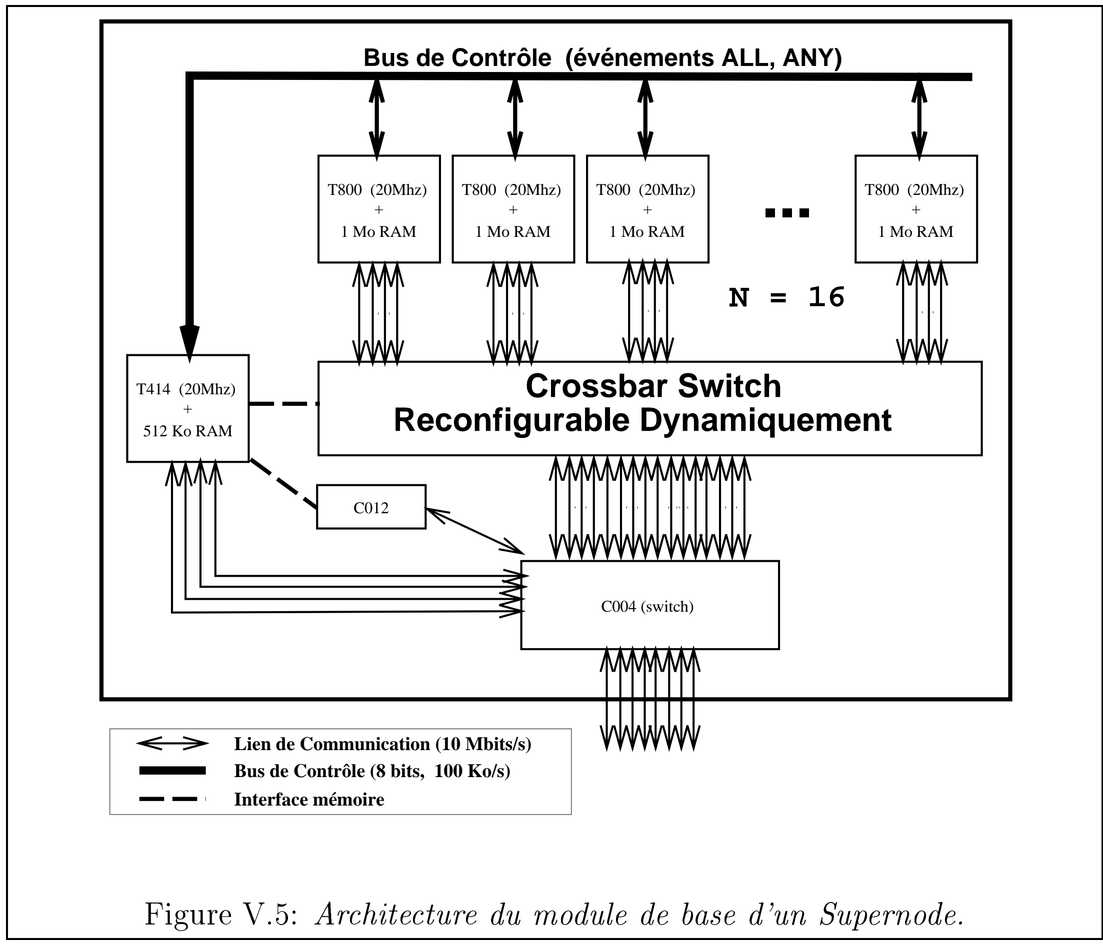


Figure V.5: *Architecture du module de base d'un Supernode.*

Aucun système d'exploitation n'est implanté sur cette machine. Les langages de programmation disponibles sont *Occam* [Lim88] et le *C* d'Inmos [Inm90] comportant des primitives spécifiques au parallélisme.

a) Les ressources de calcul : *le Transputer*

Chaque processeur du *Méganode* fonctionne avec une horloge à 20 Méga-Hertz. La communication entre processeurs est rendue possible par la présence de 4 liens permettant une communication bidirectionnelle. Chaque lien a une bande passante (théorique) de 10 Mégabits par seconde et par lien. Enfin, il faut remarquer que chaque processeur de la machine dispose de 1 Mégaoctet de mémoire seulement.

Du point de vue programmation, le Transputer est tout à fait original. Son jeu d'instructions a été spécialement conçu pour faciliter l'implantation du langage *Occam* [Lim88]. La multiprogrammation y est aisée car le processeur comporte un ordonnanceur de processus câblé, une gestion du temps (échéancier) ainsi que des

primitives de communication par *rendez-vous*. Ses capacités de multiprogrammation sont utilisées pour l'implantation des composants de notre environnement.

b) Réseau d'interconnexion

Pour toutes nos expériences, nous avons choisi de configurer le *Méganode* en Tore (4×4) de 16 processeurs et de deux processeurs auxiliaires (un processeur interface avec la machine hôte et un processeur utilisé pour la fermeture de tore - cf. figure V.6). La raison de ce choix est liée à la structure du *Méganode*[4]. Une telle topologie est construite en utilisant seulement un module du *Méganode*. Le tore a été choisi pour son faible diamètre.

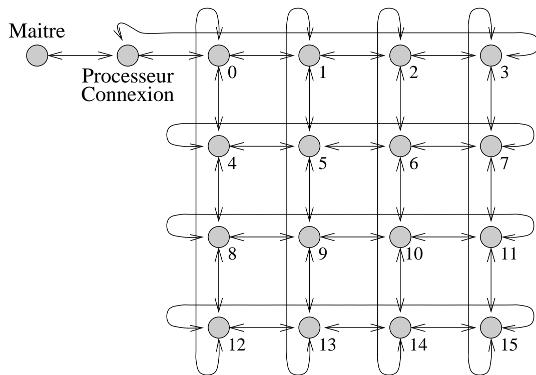


Figure V.6: *Topologie du réseau de processeurs.*

La machine ne disposant pas d'un routeur matériel, nous avons utilisé le routeur logiciel développé à l'université de Southampton, VCR *(Virtual Channel Router)* [DHN91]. VCR effectue un routage en mode commutation de paquets. La configuration que nous avons utilisée considère une taille de paquets de 160 octets envoyés en pipeline. Les communications ne sont pas totalement synchrones (le rendez-vous n'est pas parfait) car le récepteur d'un message envoie un accusé de réception *(acknowledgement)* lors de la réception du premier paquet et non pas lors de la réception de tout le message. De cette façon, l'émetteur peut se débloquer avant la réception complète du message par le destinataire. Le routage est statique, dans la mesure où les messages prennent toujours le même chemin entre deux processeurs pendant une exécution.

[4] *Cette restriction est due à la présence de problèmes physiques sur certains modules de la machine (liens de communication).*

V.6.2 Organisation de la plate-forme

La plate-forme est implantée par trois types de serveurs : le *travailleur*, le *gestionnaire de communication* et le *régulateur* (cf. figure V.7). Tous les serveurs sont virtuellement complètement connectés : pour chaque couple de serveurs (s_i, s_j), il existe un canal logique de communication de s_i à s_j (même pour $i = j$). Le routage des messages entre processeurs est assuré par VCR *(Virtual Channel Router)* [DHN91].

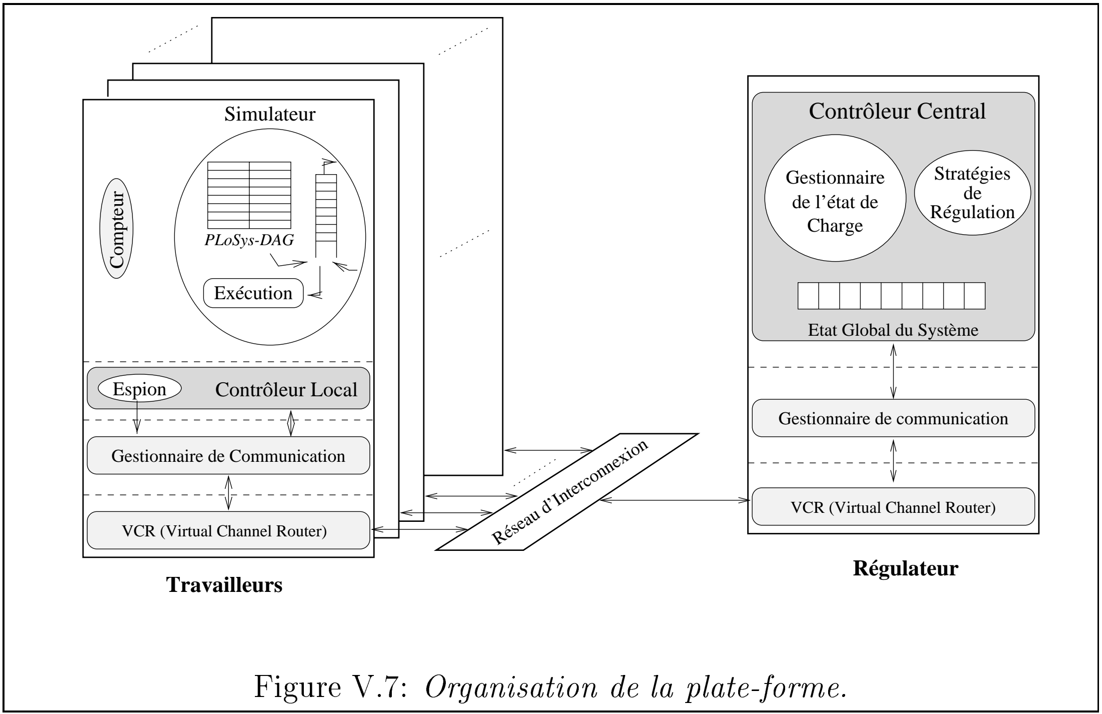


Figure V.7: *Organisation de la plate-forme.*

a) Le travailleur

Ce serveur s'exécute sur chaque processeur du *Méganode*. Il a deux rôles :

- recevoir la description du graphe de tâches *(PLoSys_DAG)* du processeur de contrôle et exécuter le traitement synthétique attribué à chaque nœud du graphe ;
- décompter le temps d'inactivité du processeur.

Ce serveur est implanté par deux processus :

Le simulateur : il modélise le moteur d'inférence de Prolog. Si la liste des tâches prêtes à être exécutées sur le processeur n'est pas vide, ce processus

en sélectionne une et l'exécute. Cette exécution consiste à itérer sur une boucle vide. Le nombre d'itérations est proportionnel au coût de traitement attribué à la tâche considérée. Ce traitement effectué, le processus insère dans la pile les tâches filles engendrées par cette tâche. La pile des tâches sur chaque processeur est gérée comme une liste à double accès. Les tâches générées localement sont insérées au sommet de la pile. Si le processeur doit exporter des tâches, celles-ci sont sélectionnées à partir du fond de la pile. Pour une exécution locale, les tâches sont extraites à partir du sommet de la pile.

Le compteur : ce processus n'est déclenché que lorsque tous les autres processus sont inactifs. Il incrémente un compteur dont la valeur finale est utilisée pour estimer le temps d'oisivité du processeur.

b) Le gestionnaire de communication

Ce serveur réalisé au dessus de VCR [DHN91] doit faciliter la gestion des messages de communication entre les processeurs. Il est lancé sur tous les processeurs à l'initialisation du système. Un message est caractérisé par sa taille, son type et les données à transmettre. Deux types de messages sont distingués :

- les messages de contrôle (initialisation, terminaison, acquittement) ;
- les messages de régulation (état de charge, commandes de régulation).

c) Le régulateur

Ce serveur matérialise notre fonction de régulation. Il est implanté par trois processus : *le contrôleur central* qui s'exécute sur un processeur dédié, *le processeur de contrôle*, *le contrôleur local* et *l'espion* qui s'exécutent sur chaque processeur du *Méganode*. Ces processus coopèrent pour le contrôle du degré de parallélisme et la répartition dynamique de charge.

Le contrôleur central : Il envoie à chaque tâche *'travailleur'* T_i la description du graphe de tâches *(PLoSys_DAG)*. Après cette phase de chargement, ce processus envoie un message d'initialisation à tous les processeurs. L'initialisation peut être paramétrée de telle sorte qu'elle exécute le graphe de tâches sur un seul processeur (ce qui correspond à une exécution séquentielle), ou de manière à avoir une exécution parallèle sur un groupe de processeurs. Pour une exécution parallèle, il sélectionne un processeur et lui envoie un message de démarrage de l'exécution synthétique du graphe de tâches. Les processeurs restants reçoivent un message d'initialisation.

Une horloge est alors initialisée. L'exécution du graphe de tâches terminée, tous les processeurs envoient un signal de fin au contrôleur central qui arrête l'horloge et calcule le temps d'exécution total. Le contrôleur reçoit par la suite une trace de l'exécution effectuée sur chacun des processeurs. De l'initialisation à la fin du traitement du graphe de tâches, ce processus maintient l'état de charge du système et prend des décisions de régulation. Ces décisions sont établies par un algorithme de régulation, à partir de l'état de charge du système connu à cet instant *(état approché)*. L'algorithme consiste à apparier un processeur surchargé et un processeur oisif. Les commandes de régulation sont envoyées aux contrôleurs locaux des deux processeurs concernés.

Le contrôleur local : Ce processus gère les commandes de régulation émanant du processeur de contrôle. Il implante une stratégie de sélection qui détermine les tâches qui doivent être transférées vers le processeur oisif. L'opération de régulation consiste à :

- ôter les tâches choisies de la pile des tâches du processeur surchargé ;
- envoyer un message au processeur oisif. La taille du message correspond au coût mémoire attribué à la tâche exportée[5] ;
- insérer les tâches exportées dans la pile du processeur oisif.

Lorsque le transfert est effectué, les deux processeurs (source et cible du transfert) émettent un message d'acquittement vers le processeur de contrôle, et le processeur précédemment oisif reprend son exécution.

l'espion : Ce processus estime la charge de travail locale au processeur à partir de la taille de la pile des tâches (nombre de tâches en attente d'exécution). Il informe périodiquement le processeur de contrôle des changements d'état donnés[6].

V.6.3 Schéma d'une exécution parallèle

Le schéma de coopération parallèle entre les unités de travail modélise le principe de fonctionnement du modèle de calcul du système PLoSys (cf. figure V.9).

Initialement, l'exécution du graphe de tâches commence sur un seul processeur. Le régulateur (contrôleur central) reçoit successivement les informations

[5] *Simule l'envoi du contexte d'exécution de la tâche exportée.*

[6] *Ce processus est initialisé seulement si la stratégie d'évaluation de la charge des processeurs est initialement fixée à l'approche fondée sur l'échantillonnage périodique de l'état de charge (cf. section VI.4.1)*

sur l'état de charge (nombre de tâches disponibles) en provenance de chaque processeur (contrôleurs locaux). Il classe chaque processeur, à partir d'une stratégie de sélection à seuil sur le nombre de tâches cumulées, en trois catégories : *Oisif, Surchargé ou Isolé* (cf. figure V.8).

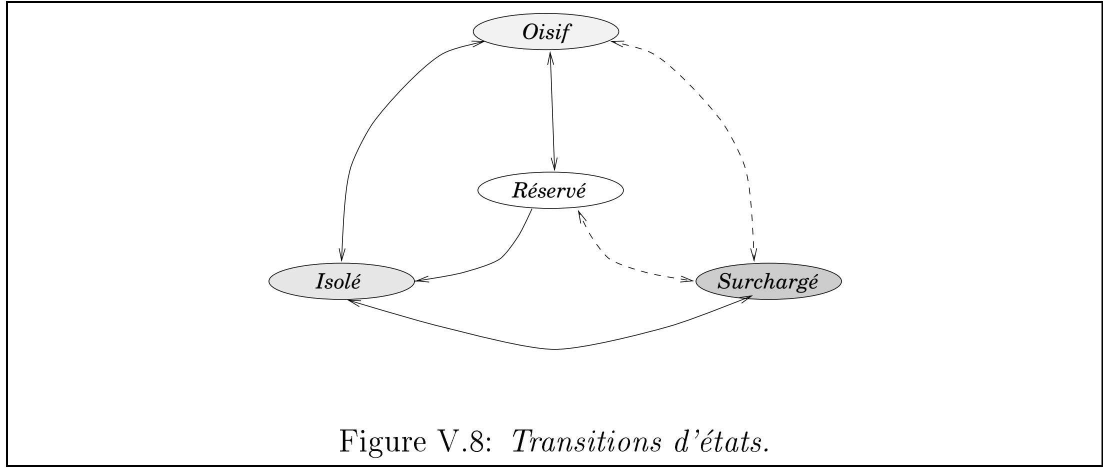


Figure V.8: *Transitions d'états.*

Un état supplémentaire, *Réservé* est utilisé afin de distinguer un processeur *oisif* sélectionné pour une opération de régulation (transfert de charge). La transition d'*isolé* à *surchargé* correspond à un franchissement d'un seuil de charge qui est un paramètre de la stratégie de régulation. Les transitions d'états entraînent les traitements suivants :

Lorsqu'un processeur devient *inactif* Le processeur n'a plus de tâches disponibles dans sa pile. Il signale une charge négative ($Charge = -1$) au régulateur. Le régulateur met à jour l'état de charge approché du système et effectue un test de détection de fin d'exécution du graphe de tâches[7].

Lorsqu'un processeur devient *isolé* Le régulateur est simplement informé de cet état et met à jour l'état global approché du système.

Lorsqu'un processeur devient *surchargé* Le processeur signale sa charge au régulateur. Ce dernier met à jour l'état du système et essaie alors d'apparier ce processeur avec un processeur inactif de la manière suivante :

- S'il existe un processeur inactif, le régulateur signale aux deux processeurs (surchargé et inactif) que l'opération de transfert peut être effectuée. Les deux processeurs sont répertoriés alors comme *réservés*, pour éviter qu'ils ne soient resélectionnées aussitôt.

[7] *Si le nombre de processeurs inactifs est égal au nombre de processeurs utilisés, alors le régulateur détecte la fin d'exécution du graphe de tâches.*

- L'état du système est mis à jour dès que les processeurs appariés ont acquitté le transfert (échec ou succès de la régulation de charge). Ces acquittements sont *tracés* afin de mesurer le taux de décisions de régulation *inappropriées* (cf. section VI.4).

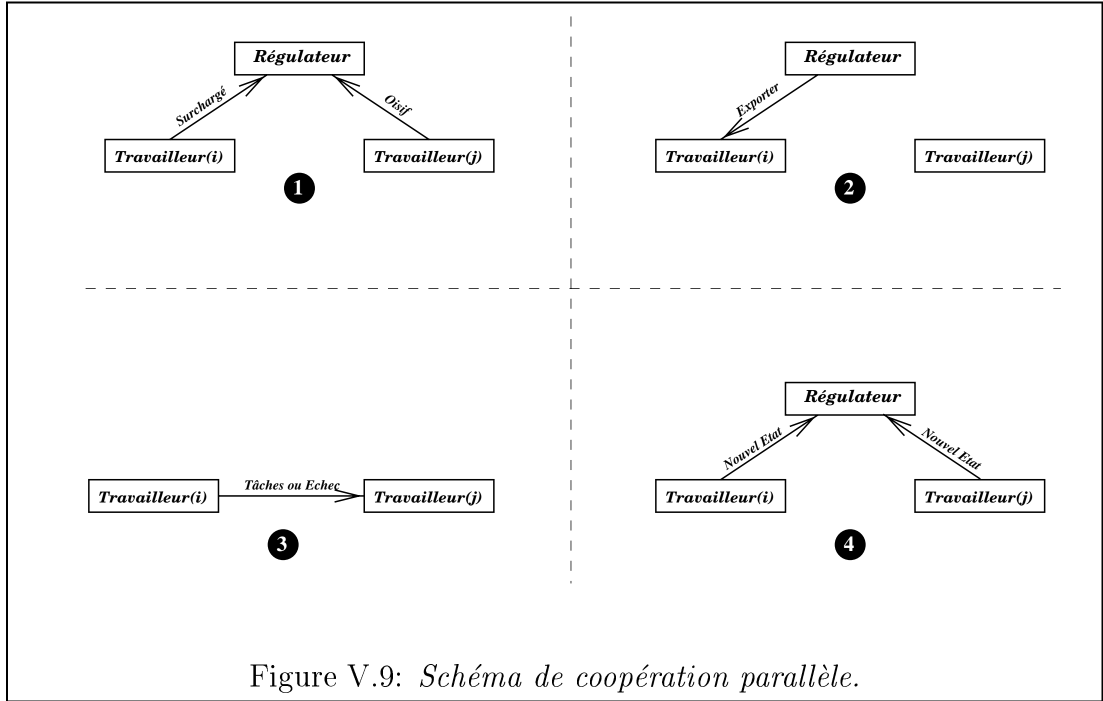


Figure V.9: *Schéma de coopération parallèle.*

V.7 Collecte d'informations de mesures

Trois types de mesure sont obtenues par le moniteur de collecte d'informations lors d'une exécution d'un graphe de tâches : (a) le temps d'exécution des tâches synthétiques, (b) la charge des processeurs au cours de l'exécution et (c) le nombre d'invocations de la fonction de régulation et le taux d'échec (décisions de régulation infructueuses) enregistré.

La charge globale d'un processeur est composée de trois éléments (cf. figure V.10) :

1. *temps utile :* c'est la fraction de temps total correspondant à l'exécution des boucles vides. Cette fraction peut être estimée avant l'exécution et après l'utilisation des stratégies implantées, car nous connaissons la puissance du processeur *'émulé'* et le nombre total de boucles vides exécutées sur chaque processeur.

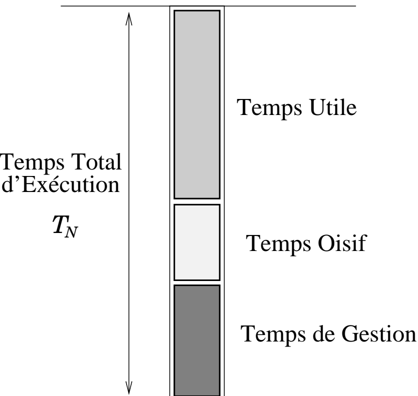


Figure V.10: *Caractéristiques d'une exécution*

2. *temps oisif :* c'est la fraction du temps d'exécution où le processeur est inactif (n'exécute pas de tâches synthétiques),

3. *temps de gestion :* il correspond au surcoût dû aux communications et à la gestion de la pile des tâches[8].

V.8 Interface utilisateur

La visualisation des caractéristiques d'exécution des programmes, la mise au point ainsi que le réglage précis d'un système parallèle se sont révélés être d'un grand secours dans l'analyse [Tic92, CGH93].

Nous avons développé un outil graphique *PVT* (*PLoSys Visualization Tool*), en vue de faciliter les interactions avec la machine cible (le *Méganode*) permettant de visualiser les caractéristiques d'exécution post-mortem d'un programme Prolog (cf. figure V.11).

Cette visualisation est réalisée à partir de l'analyse des informations de trace d'exécution d'un graphe de tâches donné (collectées dans un fichier *Log_File*).

L'outil a pour objectif de montrer le comportement de l'exécution du graphe de tâches, au moment de son exécution, en fonction de la stratégie de régulation adoptée. Le réglage des paramètres de régulation s'effectue à l'initialisation (panneau de contrôle) de l'exécution d'un graphe de tâches.

Les résultats présentés dans le chapitre suivant ont été collectés au moyen de cet outil. Cet outil est brièvement présenté en annexe C.

[8] *Ce temps est estimé à partir du temps total d'exécution T_N, du temps utile et du temps oisif (cf. section VI.3.2).*

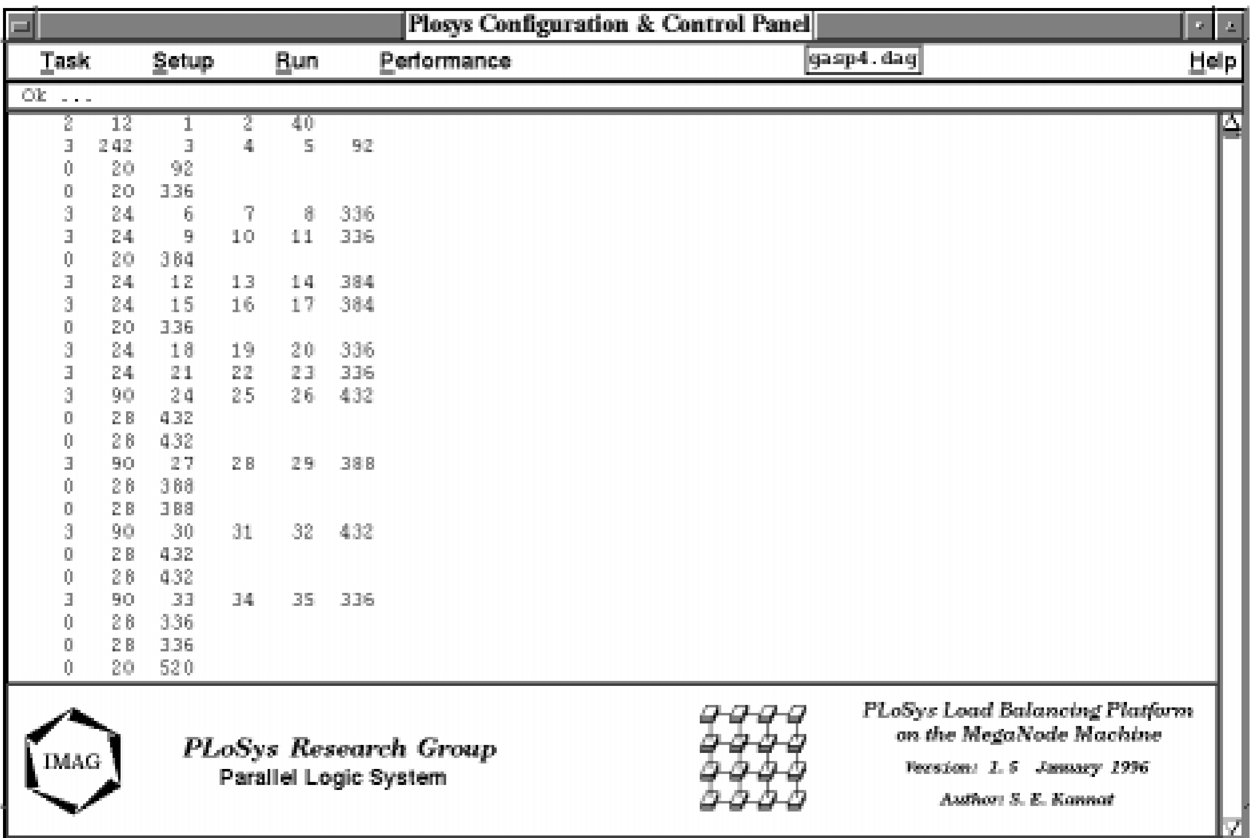


Figure V.11: *Interface utilisateur*

V.9 Conclusion

PLoSys se place dans la catégorie de systèmes qui adoptent un modèle d'exécution multi-séquentiel sur des machines parallèles sans mémoire commune (architecture de type NORMA), exploitant automatiquement le parallélisme OU inhérent à la sémantique des programmes Prolog *'pur'*. Dans cette classe de systèmes, l'exécution d'un programme Prolog correspond à une recherche exhaustive en parallèle de toutes les solutions possibles. La parallélisation est guidée dynamiquement par la stratégie de régulation, adoptée en fonction des ressources de calcul disponibles.

Nous avons proposé dans ce chapitre une fonction de régulation de charge pour le système PLosys afin de contrôler l'accroissement du parallélisme, lequel a pour objectif de *garantir* un gain de performance par rapport à une exécution séquentielle. Cette fonction repose sur deux heuristiques simples : la première concerne l'évaluation de la charge d'un processeur. Cette charge est estimée, à un instant donné, à partir du nombre de points de choix cumulés sur chaque

processeur. La deuxième heuristique concerne la vérification des contraintes de granularité et d'efficacité. Elle considère pour cela la profondeur des points de choix dans l'arbre OU développés par l'exécution du programme.

Une première analyse, sur deux classes de comportements de programmes Prolog, montre que le gain de performance dépend fortement des valeurs attribuées aux paramètres de notre fonction de régulation.

Nous avons ensuite présenté une approche originale, en vue d'évaluer plus finement, dans le chapitre suivant, différentes stratégies de régulation de charge pour notre système PLoSys . Il est important de remarquer que le modèle s'efforce de refléter un système Prolog pur réel sur une machine sans mémoire commune. Notre objectif vise à représenter des événements significatifs au moyen d'une technique de modélisation et en effectuant des mesures directes. Nous pensons que des informations plus réalistes sont obtenues, étant donné que les coûts de communication et de gestion sont difficiles à représenter par un modèle analytique ou un modèle de simulation.

La flexibilité de notre modèle par l'utilisation de tâches synthétiques accompagnées de paramètres adaptables permet d'analyser différents comportements. Ces paramètres permettent de modéliser différentes échelles de problème et de simuler les caractéristiques spécifiques d'une machine.

Chapitre VI

Contexte expérimental : Premiers résultats

VI.1 Introduction

L'évaluation de la fonction de régulation de charge élaborée pour le système PLoSys a deux objectifs principaux :

- la validation des idées mises en œuvre dans la phase de conception ;
- l'évaluation critique de leur mise en œuvre.

L'évaluation d'une fonction de régulation de charge pose un certain nombre de problèmes. Il s'agit en général de :

- définir les paramètres significatifs à évaluer,
- réduire l'incidence de la prise de mesures sur le comportement du système ;
- définir des programmes de tests significatifs.

Nous présentons, dans la première partie de ce chapitre, les indices que nous avons retenus pour évaluer les performances de notre fonction de régulation de charge. La deuxième partie décrit le jeu de test considéré. La dernière partie de ce chapitre est consacrée à la présentation et l'analyse des premiers résultats de mesure obtenus.

VI.2 Indices de performances

L'objectif principal du système PLoSys est l'amélioration des performances d'exécution de programmes Prolog par l'utilisation du parallélisme. L'accélération (*speedup*) et l'efficacité (*efficiency*) sont les deux indices de mesure classiques des performances d'un système parallèle.

Définition VI.1 *L'* ***accélération*** $S_N(S, \mathcal{F}_{reg}, \rho, \pi)$ *est définie comme le rapport du temps* $T_S(S, \rho, \pi)$ *obtenu avec la meilleure exécution séquentielle* S *sur celui obtenu avec* N *processeurs* $T_N(\mathcal{F}_{reg}, \rho, \pi)$. *Elle est mesurée pour une fonction de régulation* $\mathcal{F}_{reg}$ *sur une instance de donnée* π *d'un programme Prolog* ρ.

$$S_N(S, \mathcal{F}_{reg}, \rho, \pi) = \frac{T_S(S, \rho, \pi)}{T_N(\mathcal{F}_{reg}, \rho, \pi)} \tag{VI.1}$$

On notera par simplicité : $S_N = \frac{T_S}{T_N}$. Une accélération S_N est qualifiée de :

$$\begin{cases} \textbf{sur-linéaire}, & \text{si } S_N > N. \\ \textbf{sous-linéaire}, & \text{si } 1 < S_N \ll N. \\ \textbf{préjudiciable}, & \text{si } S_N < 1 \textit{ on perd du temps.} \end{cases}$$

Quand il n'est pas toujours possible de comparer avec la meilleure exécution séquentielle, on prend pour temps séquentiel de référence celui donné par l'exécution parallèle du programme sur un seul processeur, c'est à dire $T_1(\mathcal{F}_{reg}, \rho, \pi)$ (**l'accélération relative** $S_N = \frac{T_1}{T_N}$).

Cette mesure traduit le gain en vitesse de résolution en utilisant P processeurs avec une fonction de régulation $\mathcal{F}_{reg}$ sur une instance de donnée π pour un programme Prolog ρ.

Définition VI.2 *L'* ***efficacité*** $E_N(S, \mathcal{F}_{reg}, \rho, \pi)$ *est définie comme étant le rapport de l'accélération* $S_N(M, \mathcal{F}_{reg}, \rho, \pi)$ *sur le nombre* N *de processeurs utilisés.*

$$E_N(M, \mathcal{F}_{reg}, \rho, \pi) = \frac{S_N(S, \mathcal{F}_{reg}, \rho, \pi)}{N} \tag{VI.2}$$

On notera par simplicité : $E_N = \frac{S_N}{N}$. L'efficacité exprime l'utilisation effective des ressources de calcul offertes par les P processeurs.

Les mesures d'accélération S_N et d'efficacité E_N, quoique habituelles, sont souvent contestées. D'une part, à cause du choix de T_1 qui dépend fortement de l'implémentation séquentielle, et d'autre part, du fait qu'elles ne traduisent

pas l'influence des paramètres comme la complexité du programme à traiter, l'architecture des machines et l'augmentation du nombre de processeurs.

En effet, l'efficacité (la vitesse) d'un système logique parallèle est déterminée par l'architecture de la machine cible A, la fonction de régulation de charge $\mathcal{F}_{reg}$, le nombre de processeurs N et la complexité W du programme. Considérons par exemple que la complexité W d'un programme Prolog ρ représente le nombre de points de choix (nœud OU) engendré par son exécution. Pour une complexité donnée W, l'augmentation du nombre de processeurs N s'accompagne d'une augmentation des coûts de communication (réseau plus grand, messages plus longs et plus nombreux). Pour maintenir une efficacité E constante, Il faut compenser la perte d'efficacité liée aux communications en augmentant la complexité W du programme ρ.

Définition VI.3 *Un système logique parallèle est qualifié d'**extensible** (scalable), si l'accroissement de la complexité W d'un programme ρ en fonction de celui du nombre de processeurs utilisés N lui permet de garder la même efficacité.*

Le facteur d'accroissement de la complexité W au regard de celui du nombre de processeurs dépend de l'architecture machine A et de la fonction de régulation de charge $\mathcal{F}_{reg}$. Kumar, Ananth et Rao [KG92] ont ainsi défini une mesure complémentaire, appelée **isoefficacité** (*isoefficiency*). Cette mesure permet de connaître le comportement d'un système parallèle face à l'extensibilité des machines.

Définition VI.4 *Si, pour garder une efficacité E du système logique parallèle constante, la complexité d'un programme ρ doit augmenter suivant une fonction $I(N)$, alors $I(N)$ est appelée **fonction d'isoefficacité**.*

Si pour une efficacité E donnée, l'accroissement de la complexité des programmes est linéaire par rapport à la fonction d'isoefficacité $I(N)$, le système (ou la fonction de régulation de charge $\mathcal{F}_{reg}$) est dit à **extensibilité optimale**.

VI.2.1 Accélération et efficacité

Ne disposant pas du système Prolog séquentiel *le plus efficace* sur l'architecture cible (le *Méganode*), nous avons retenu comme mesure de facteur d'accélération (**relative**) le rapport entre le temps d'exécution par le système parallèle sur un seul processeur (mode séquentiel) et le temps d'exécution parallèle du graphe de tâches sur les 16 processeurs.

Par ailleurs, nous intéressant plus particulièrement au comportement de la fonction de régulation sur différents programmes, notre choix, dans un premier temps, a été d'analyser le facteur d'accélération en fonction des stratégies de régulation adoptées, plutôt que d'analyser le facteur d'extensibilité de la fonction

de régulation. Cette décision est également due en partie à la complexité de manipulation des différentes configurations de la machine cible.

Notons T_N le temps d'exécution total du graphe de tâches sur N processeurs. Nous avons :

$$\forall i \in [0, ..., N-1], \quad T_N = T_{utile}(i) + T_{oisif}(i) + T_{gestion}(i) \tag{VI.3}$$

T_1 désigne alors le temps d'exécution séquentiel du graphe de tâches sur le *Méganode*. Etant donné que l'exécution parallèle comme l'exécution séquentielle exécutent le même graphe de tâches, et compte tenu du surcoût occasionné par la gestion des tâches, nous avons l'inégalité suivante :

$$T_1 \geq \sum_{i=0}^{(N-1)} T_{utile}(i) \tag{VI.4}$$

L'accélération relative $S_N(G, \mathcal{F}_{reg})$ obtenue par l'application d'une fonction de régulation de charge $\mathcal{F}_{reg}$ dans l'exécution parallèle d'un graphe de tâches G sur N processeurs est donnée par :

$$S_N(G, \mathcal{F}_{reg}) = \frac{T_1(G)}{T_N(G, \mathcal{F}_{reg})} \tag{VI.5}$$

L' efficacité $E_N(G, \mathcal{F}_{reg})$ est donnée par le rapport de l'accélération $S_N(G, \mathcal{F}_{reg})$ sur le nombre de processeurs N utilisés.

$$E_N(G, \mathcal{F}_{reg}) = \frac{S_N(G, \mathcal{F}_{reg})}{N} \tag{VI.6}$$

VI.2.2 Facteur de distribution

Puisque nous cherchons à savoir dans quelle mesure les stratégies de régulation de charge peuvent équilibrer la distribution des tâches sur les processeurs, nous définissons le facteur de distribution des tâches $\mathcal{D}_f(G, \mathcal{F}_{reg}, N)$ de la manière suivante :

$$\mathcal{D}_f(G, \mathcal{F}_{reg}, N) = \frac{\frac{1}{N}\sum_{i=0}^{(N-1)} Nb_{tâches}(i)}{Max(Nb_{tâches}(i))} \tag{VI.7}$$

Dans ce rapport, $Nb_{tâches}(i)$ indique le nombre total de tâches attribuées au processeur i. Une meilleure distribution de charge se reflète par les plus grandes valeurs de $\mathcal{D}_f$.

$$\frac{1}{N} \leq \mathcal{D}_f(G, \mathcal{F}_{reg}, N) \leq 1 \tag{VI.8}$$

Cette notion tend à refléter une distribution de charge *idéale*, telle qu'il est possible de la rencontrer dans une stratégie d'équilibrage qui exclut les surcoûts dus aux communications et au surcoût de gestion des tâches.

VI.3 Programmes de test

Les contraintes matérielles ne nous ont pas parmi d'utiliser un large éventail de programmes Prolog. En effet, du fait de la limitation en taille mémoire sur chaque processeur du *Méganode* (1 Mégaoctets), nous nous sommes limités à deux programmes de test : le programme de résolution du problème des 8-reines et de celui des 4-gasp. Les programmes Prolog correspondant à la résolution de ces deux problèmes sont présentés en annexe A.

Le problème des **8-reines** consiste à rechercher toutes les configurations possibles lorsque l'on place huit reines sur un échiquier (8×8) de telle manière qu'aucune reine ne puisse en attaquer une autre (cf. figure VI.1).

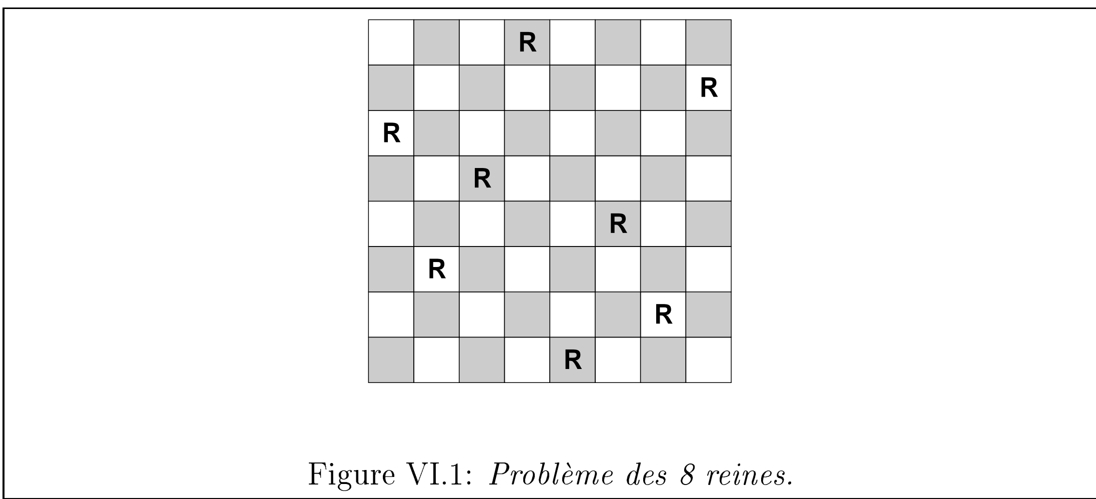


Figure VI.1: *Problème des 8 reines.*

Considérons un échiquier (4×4) sur lequel on place 16 jetons réversibles de deux couleurs (une couleur différente pour chaque face). Partant d'une configuration initiale où toutes les faces noires sont apparentes, l'objectif est d'obtenir une configuration finale de l'échiquier où toutes les faces sont blanches, en appliquant la règle de sélection suivante : chaque jeton sélectionné doit modifier la couleur des jetons de ses cases voisines sans que sa couleur soit modifiée. Le programme **4-gasp** est le programme Prolog qui recherche toutes les solutions possibles à ce problème (cf. figure VI.2).

L'observation de ces deux programmes de test nous ont permis d'aboutir à des conclusions intéressantes concernant le comportement de notre fonction de régulation.

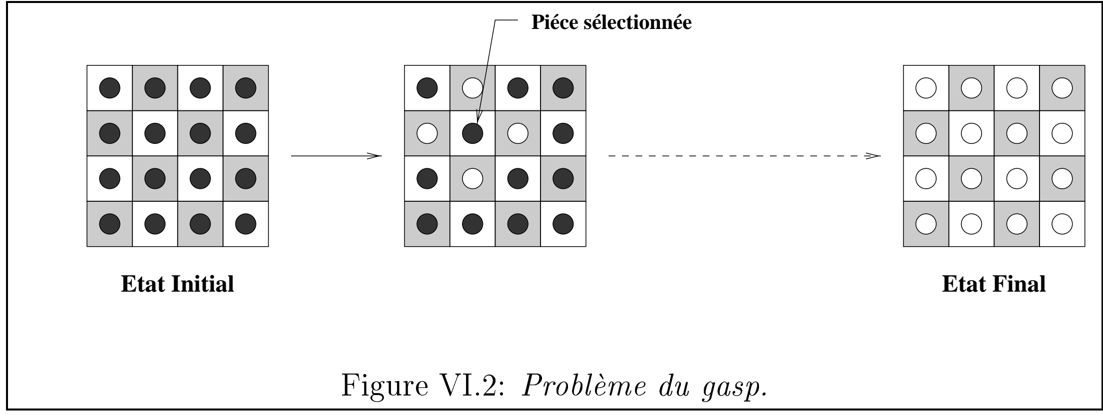


Figure VI.2: *Problème du gasp.*

VI.3.1 Extraction du graphe de tâches

L'extraction du graphe de tâches correspondant aux deux programmes Prolog du jeu de test est réalisée à partir d'une analyse post-mortem d'une trace d'exécution. Pour obtenir cette trace d'exécution, nous avons utilisé le moteur d'inférence WAMCC [Dia94] (cf. annexe B).

Programmes	Taille du graphe	profondeur h	δ_m	ω_m	μ_m
4-gasp	6252	85	25	3	1945
8-reines	15148	47	71	2	795

Table VI.1: *Caractéristiques des programmes d'essai.*

L'analyse post-mortem de cette trace d'exécution permet de construire le graphe de tâches correspondant à l'exécution du programme (cf. section V.5 et l'annexe B).

La table VI.1 présente les principales caractéristiques des programmes utilisés dans le jeu de test, à savoir : la taille du graphe (nombre de tâches), la profondeur maximale du graphe (h), le coût de traitement moyen d'une tâche (δ_m) exprimé en nombre d'itérations élémentaires, le facteur de branchement moyen d'une tâche (le nombre de tâches filles ω_m) et la taille moyenne des données à transférer lors d'une opération de régulation de charge (μ_m).

VI.3.2 Calibrage des programmes de test

A partir de mesures faites sur un transputer réel, le modèle [temps de calcul (T_{calcul}) versus nombre d'opérations (N_{ops})] employé est le suivant [KP94] :

$$T_{calcul} = 1,76 * N_{ops} \tag{VI.9}$$

Ce modèle linéaire est utilisé pour adapter (calibrer) le nombre d'itérations attribué à chaque tâche du graphe, en fonction de leur coût (temps) de calcul. Il est également utilisé pour déterminer les temps T_{utile} et T_{oisif} à partir du nombre d'itérations effectuées par les boucles vides (valeur des compteurs) pour chaque exécution du graphe de tâches. Le temps de gestion ($T_{gestion}$) correspondant au surcoût *(overhead)* dû aux communications et à la gestion des tâches. Ce surcoût ($T_{gestion}$) est estimé alors de la manière suivante (cf. section V.7) :

$$\forall i \in [0, ..., N-1], \quad T_{gestion}(i) = T_N - (T_{utile}(i) + T_{oisif}(i)) \tag{VI.10}$$

VI.4 Evaluation de stratégies de régulation

Nous présentons et analysons dans cette section une première campagne de mesures effectuées sur les programmes du jeu de test (cf. table VI.1). Nous nous sommes intéressés particulièrement à l'étude des points suivants :

- l'impact d'une stratégie d'évaluation de charge sur les performances d'une fonction de régulation de charge,
- l'incidence des paramètres de l'algorithme de régulation sur les performances d'exécution.

VI.4.1 Gestion de l'état de charge du système

La gestion de l'état de charge du système consiste à collecter (disséminer) les informations de charge des unités de travail. Le processeur de contrôle a pour objectif le maintien de l'état de charge global du système. La charge d'un processeur est évaluée à partir d'un indice de charge. Cet indice, dans notre plate-forme d'évaluation, correspond à la taille de la pile des tâches en attente d'évaluation. L'heuristique utilisée pour l'évaluation de la charge d'un processeur à un instant donné repose sur un double seuillage du nombre de tâches cumulées à cet instant. Cela permet de distinguer trois régimes (états) de fonctionnement (cf. figure VI.3). Le processeur de contrôle maintient un tableau donnant l'état courant de chaque unité de travail.

L'analyse effectuée à ce niveau concerne les points essentiels suivants :

1. à quel moment l'évaluation de charge d'un processeur est-elle effectuée ?
2. quel est l'impact des délais de communication sur la gestion de l'état de charge du système ?

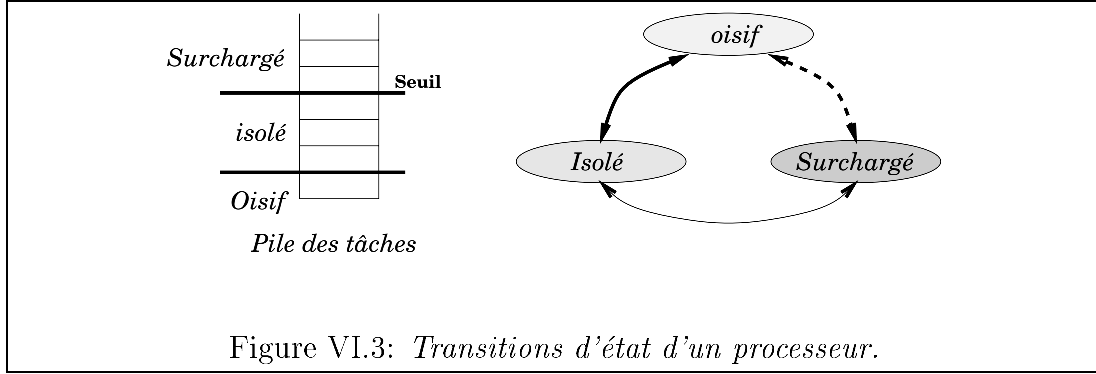


Figure VI.3: *Transitions d'état d'un processeur.*

Nous avons développé à cet effet deux stratégies d'évaluation de charge. La première est fondée sur un échantillonnage périodique de l'état de charge d'un processeur. Elle est désignée, dans la suite du document, par le terme *approche échantillonnée.* La seconde stratégie s'appuie sur les événements de création (terminaison) d'une tâche (*event-driven*).

Il est à noter la différence essentielle qui distingue ces deux approches : dans le premier cas, l'évaluation de la charge du processeur s'effectue par un processus *espion* avec une fréquence d'évaluation f_m ; dans le cas second, cette évaluation est réalisée par le *simulateur* (moteur d'inférence).

a) Evaluation de la charge d'un processeur

Un processeur, dans les deux approches d'évaluation, envoie sa charge au processeur de contrôle à chaque transition d'état ou lors d'une variation de l'état de charge d'un taux Δ. En raison des délais de communication, le processeur de contrôle à un instant donné n'a qu'une image approchée de l'état de charge exacte du système. Ceci est valable dans les deux approches d'évaluation.

Déterminer une bonne fréquence d'instrumentation (de mesure) de la charge de travail des processeurs reste un problème important à résoudre pour chaque comportement de programme, en vue de minimiser les surcoûts d'évaluation de charge. Les paramètres d'évaluation de la charge (Δ, f_m) sont fixés statiquement, à l'initialisation de chaque exécution.

b) Maintien de l'état de charge du système

Du fait des délais de communication induits par la transmission de l'état de charge, l'état global maintenu par le processeur de contrôle peut être *inconsistant.* En effet, parmi les trois états possibles d'une unité de travail, seul l'état oisif à un instant donné est sûr. Cette inconsistance peut alors être à l'origine de décisions de régulation *inappropriées* : une unité de travail surchargée peut devenir oisive

ou isolée, après avoir été sélectionnée pour transférer une partie de ses tâches en attente d'évaluation.

L'occurrence de ce phénomène dépend de la valeur du seuillage de l'indice de charge, de la granularité des tâches, ainsi que de la fréquence d'évaluation de la charge de travail (f_m).

Ceci est mis en évidence par les figures VI.4(b) et VI.5(b). Les fréquences d'échantillonnage considérées sont indiquées dans la table VI.2.

La fréquence f_0 indique l'exécution du graphe de tâches soumis à la stratégie *non-échantillonnée*. Le paramètre Δ est défini de telle sorte que le processeur envoie sa charge à chaque transition d'état et tant qu'il est surchargé.

En ce qui concerne les résultats présentés en figures VI.4(a) et VI.5(a), nous observons que la fréquence de mesure f_m doit être adaptée à la granularité des tâches, ceci afin de réduire le surcoût des communications de l'état de charge. Cette caractéristique apparaît nettement dans l'amélioration des performances d'exécution du graphe de tâches, avec l'approche échantillonnée, pour des valeurs de seuillage faibles.

Fréquences f_m	période (μs)
f_1	500
f_2	5000
f_3	20000

Table VI.2: *Fréquences d'évaluation de la charge d'un processeur.*

Des résultats expérimentaux montrent également que la fréquence de dissémination (noté f_d) de la charge d'un processeur vers le processeur de contrôle doit être inférieure à la fréquence d'évaluation (f_m) de la charge. En supposant que $f_d = f_m$, l'augmentation de la fréquence d'échantillonnage permet d'éviter une trop grande dérive de l'état global vu par le processeur de contrôle par rapport à l'état global réel du système, ce qui diminue le nombre de tentatives de transfert infructueuses. Cependant, cette augmentation de fréquence s'accompagne d'une surcharge des processeurs due à l'augmentation de la fréquence d'évaluation de la charge, ainsi que d'un surcoût de communication important.

La diminution de cette fréquence, quant à elle, permet de réduire les coûts de transfert et de surcharge des processeurs, mais cela engendre une dérive plus importante de l'état de charge maintenu par le processeur de contrôle. Cela a pour effet d'augmenter le nombre d'échecs de tentatives de transfert infructueuses. Dans ce cas de figure, l'indice de charge peut ne plus être significatif, car une transition d'état directe d'une unité d'un état surchargé à un état oisif est possible si cette fréquence est faible.

Le problème alors est de trouver un compromis, à savoir maintenir une bonne estimation de la charge globale du système, tout en minimisant les coûts de communication induits par une fréquence de dissémination trop élevée.

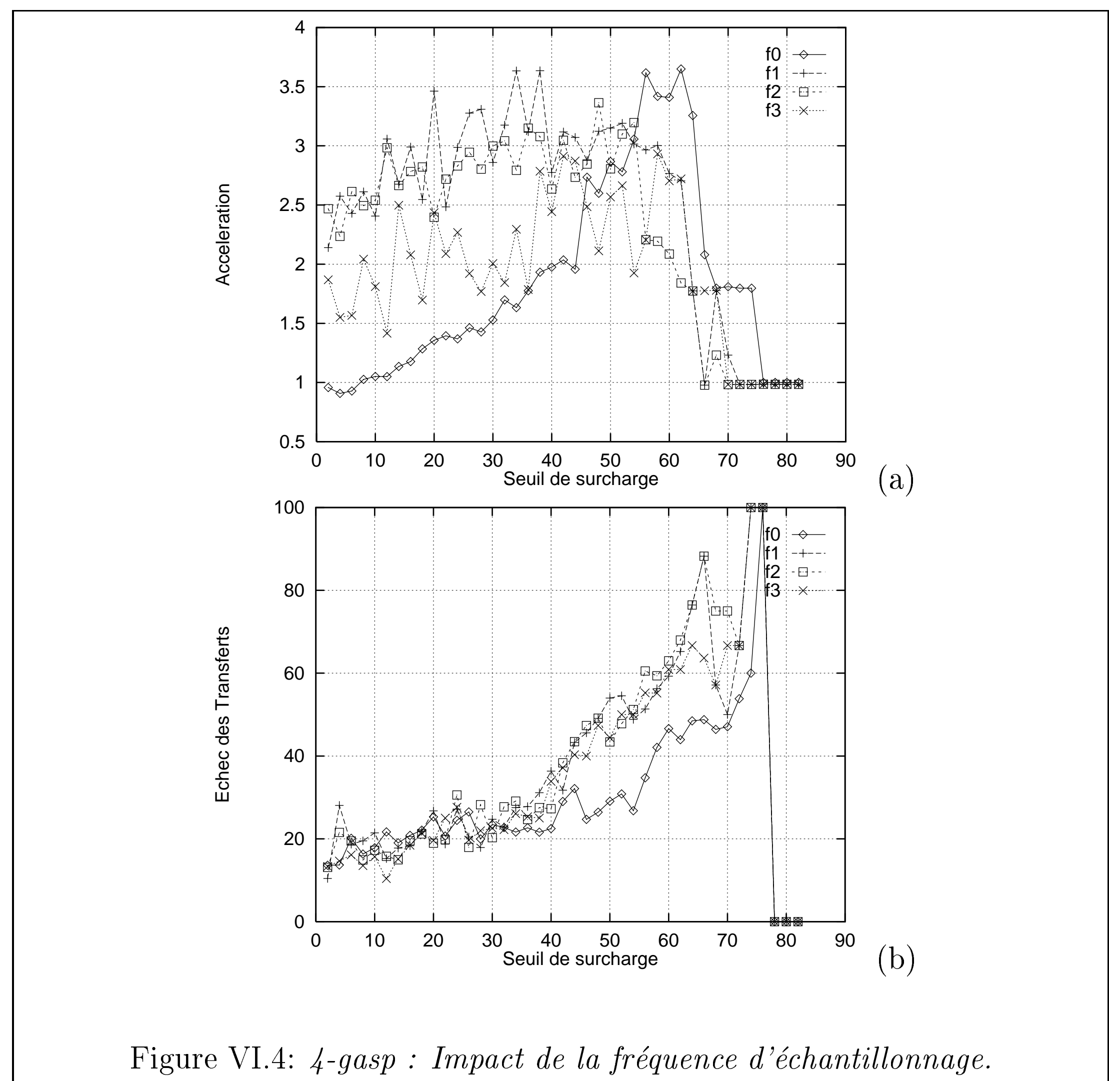


Figure VI.4: *4-gasp : Impact de la fréquence d'échantillonnage.*

VI.4.2 Algorithme de régulation

L'algorithme de régulation est la partie centrale d'une fonction de régulation de charge. Il est fondé sur les informations maintenues par le gestionnaire de l'état de charge du système pour effectuer des décisions de régulation de charge, dans le but d'optimiser le temps d'exécution du graphe de tâches. Nous nous intéressons à ce niveau à l'analyse des points suivants :

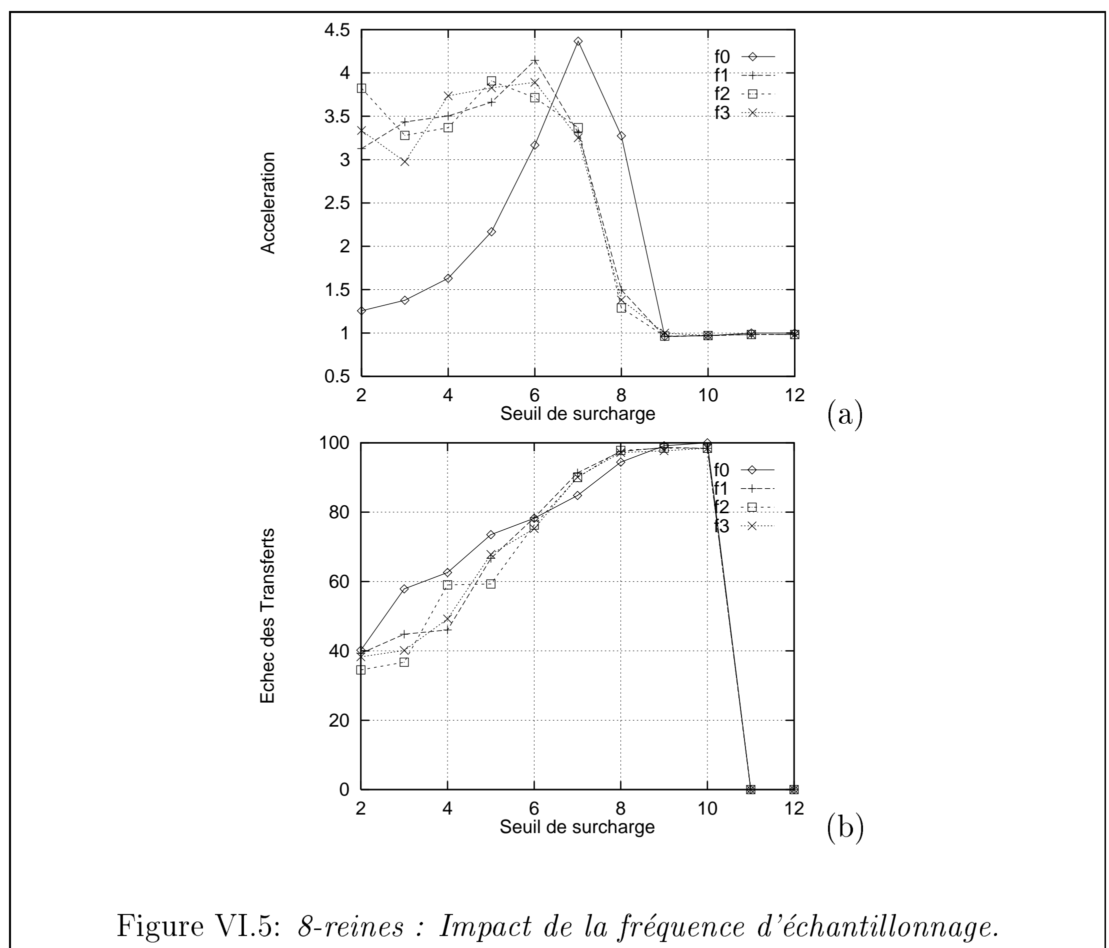


Figure VI.5: *8-reines : Impact de la fréquence d'échantillonnage.*

1. quel est l'incidence du seuillage sur les performances du système ?
2. peut-on contrôler la granularité de tâches transférées ?
3. quel impact une topologie a t-elle sur les performances du système ?

Nous analysons ces points à travers les composants d'une stratégie de régulation : (a) la stratégie d'activation qui a pour rôle le déclenchement de l'algorithme de régulation, (b) la stratégie de sélection des tâches à transférer et (c) la stratégie de localisation qui est chargée d'apparier un processeur surchargé et un processeur oisif.

a) Seuillage de l'indice de charge

Les valeurs de seuils déterminent l'état de charge de chaque processeur. Ces valeurs s'expriment sur la taille de la pile des tâches en attente d'évaluation sur chaque unité de travail.

Tout processeur ayant au moins k_{max} tâches en attente d'évaluation dans sa pile signale au processeur de contrôle qu'il est surchargé. Réciproquement, tout processeur n'ayant plus que k_{min} tâches dans sa pile informe le processeur de contrôle qu'il est sous-chargé. Dans les deux cas, cela a pour effet de déclencher l'algorithme de régulation de charge.

Considérant que la profondeur du graphe de tâches est h, les deux situations suivantes sont analysées :

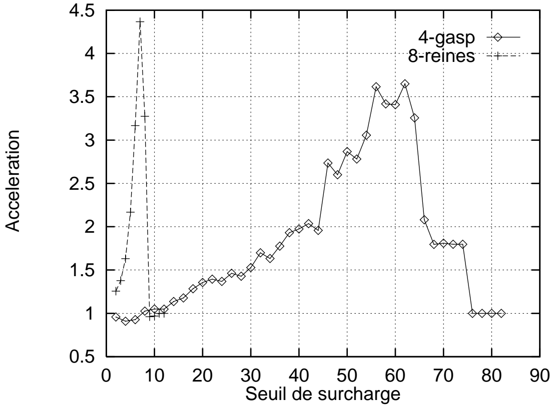


Figure VI.6: *4-gasp & 8-reines : Impact du seuillage.*

- $k_{min} = 0; k_{max} = k \ (0 < k < h)$.
- $k_{min} = 0; k_{max} = h$.

Les résultats présentés en figure VI.6 corroborent l'analyse théorique effectuée sur l'incidence des paramètres de seuillage (cf. section V.3.3), en ce sens qu'ils montrent que la fixation des paramètres de seuillage dépend des caractéristiques (comportement) du programme (granularité des tâches, nombre d'alternatives).

b) Contrôle de la granularité

Après que la stratégie d'activation ait décidé qu'un processeur est candidat pour une opération de transfert, la stratégie de sélection détermine les tâches à transférer. Les deux facteurs suivants sont alors à prendre en considération :

1. le surcoût induit par l'opération de transfert doit être minimisé ;
2. la granularité des tâches sélectionnées doit être telle qu'elle compense le surcoût occasionné par son transfert.

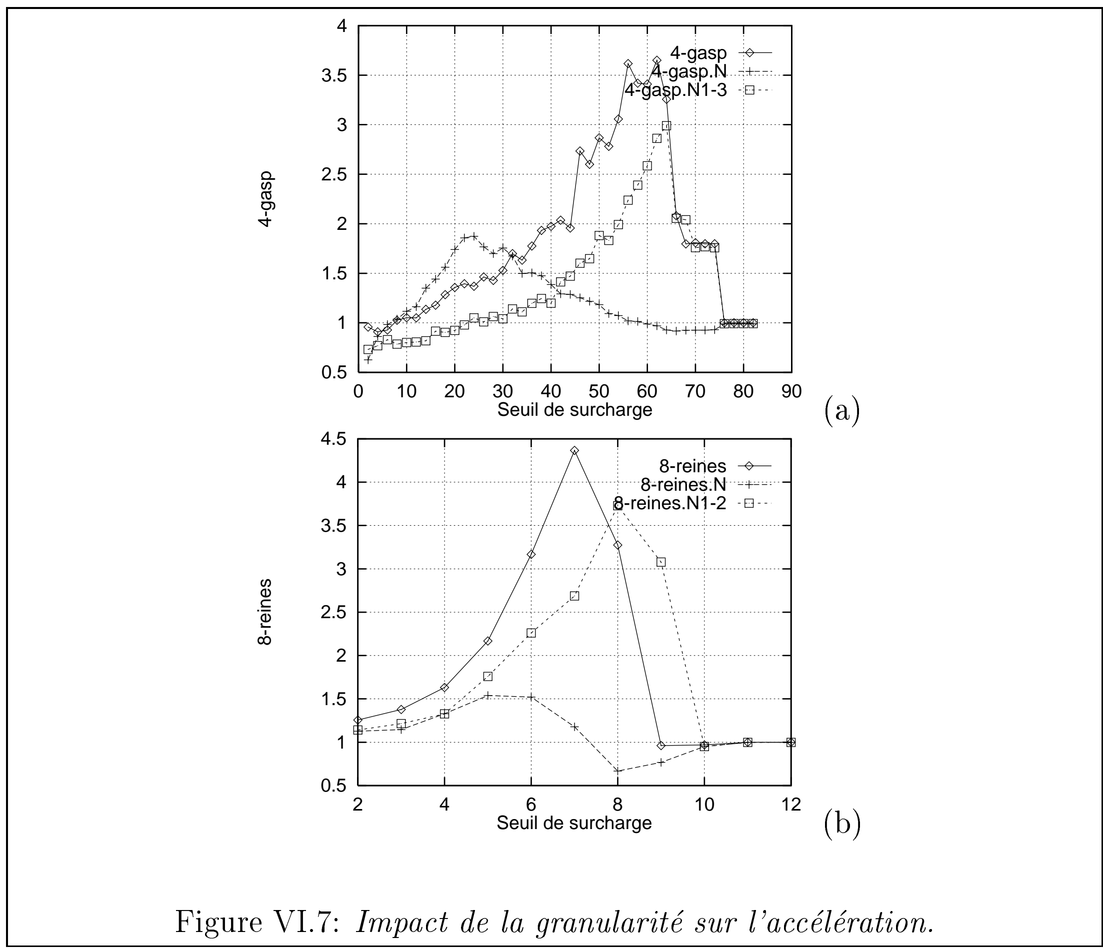


Figure VI.7: *Impact de la granularité sur l'accélération.*

Afin de souligner l'importance du choix d'une bonne stratégie, nous analysons l'influence des trois heuristiques de sélection suivantes :

1. sélection d'une seule tâche (la plus ancienne $Nb_{tâches} = 1$),

2. sélection d'un sous-ensemble de tâches selon la valeur de k_{max} du seuil de surcharge[1],

$$Nb_{tâches} = \begin{cases} \frac{Taille_i}{2} & \text{si} \quad 1 < \frac{Taille_i}{k_{max}} < 2. \\ k_{max} & \text{si} \quad \frac{Taille_i}{k_{max}} \geq 2. \end{cases}$$

3. sélection d'un sous-ensemble de tâches ($1 \leq Nb_{tâches} \leq \frac{k_{max}}{2}$).

[1] *$Taille_i$ désigne le nombre de tâches en attente d'évaluation sur le processeurs P_i à un instant donné de l'exécution.*

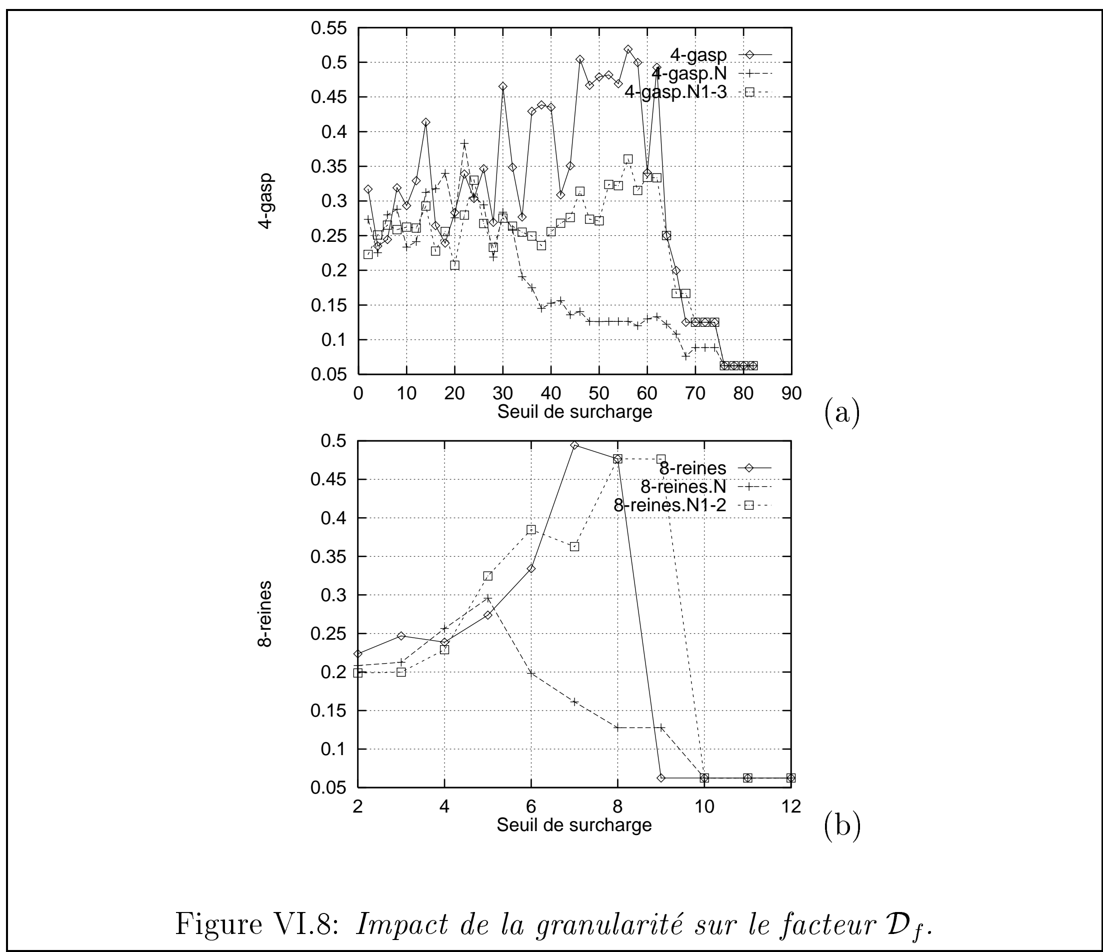


Figure VI.8: *Impact de la granularité sur le facteur* $\mathcal{D}_f$.

L'impact de ces trois heuristiques de sélection est représenté en figures VI.7 et VI.8. La première constatation que nous observons est que les meilleures performances d'exécution correspondent aux meilleures valeurs du facteur de distribution $\mathcal{D}_f$. La deuxième constatation concerne l'influence du nombre de tâches sélectionnées pour chaque transfert. On constate que les performances obtenues avec la première heuristique (une seule tâche sélectionnée à chaque transfert) sont meilleures que celles obtenues avec les deux autres heuristiques. Ce résultat ne corrobore pas notre intuition de départ (cf. section V.3.2).

c) Stratégie de localisation

La stratégie de localisation a pour objectif l'appariement d'une unité de travail surchargée et d' une unité de travail oisive. Plusieurs schémas de localisation peuvent être analysés pour évaluer l'impact d'une stratégie par rapport à une autre. Toutefois, le critère principal utilisé par le processeur de contrôle pour

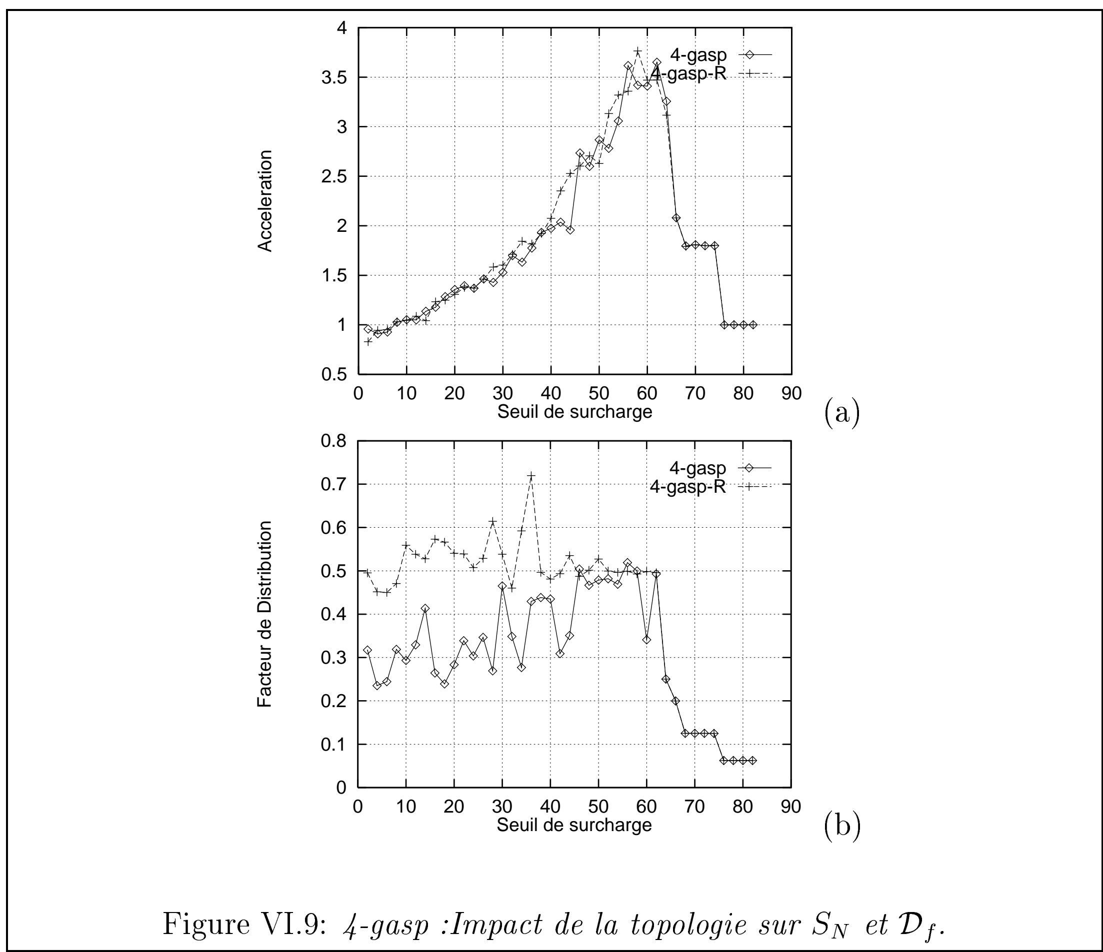


Figure VI.9: *4-gasp :Impact de la topologie sur* S_N *et* $\mathcal{D}_f$.

l'appariement des processeurs repose sur la minimisation des coûts de communication induits par ce transfert.

La satisfaction de ce critère est étroitement liée à la visibilité qu'a le processeur de contrôle des caractéristiques de l'architecture cible. Trois paramètres architecturaux peuvent affecter le critère de sélection du processeur cible du transfert : (a) le diamètre du réseau; (b) la topologie physique d'interconnexion des processeurs; (c) les caractéristiques du logiciel d'acheminement des données (routage).

Dans le cadre de l'implémentation actuelle, nous avons choisi d'analyser deux approches de sélection simples, selon la topologie physique décrite en section V.6.1 :

- sélection aléatoire, sans faire aucune hypothèse quant aux caractéristiques du réseau ;

- Pour un processeur surchargé P_{src} donné, le processeur de contrôle sélectionne un processeur P_{opt} de telle sorte que P_{opt} soit oisif et voisin immédiat

de P_{src}[2].

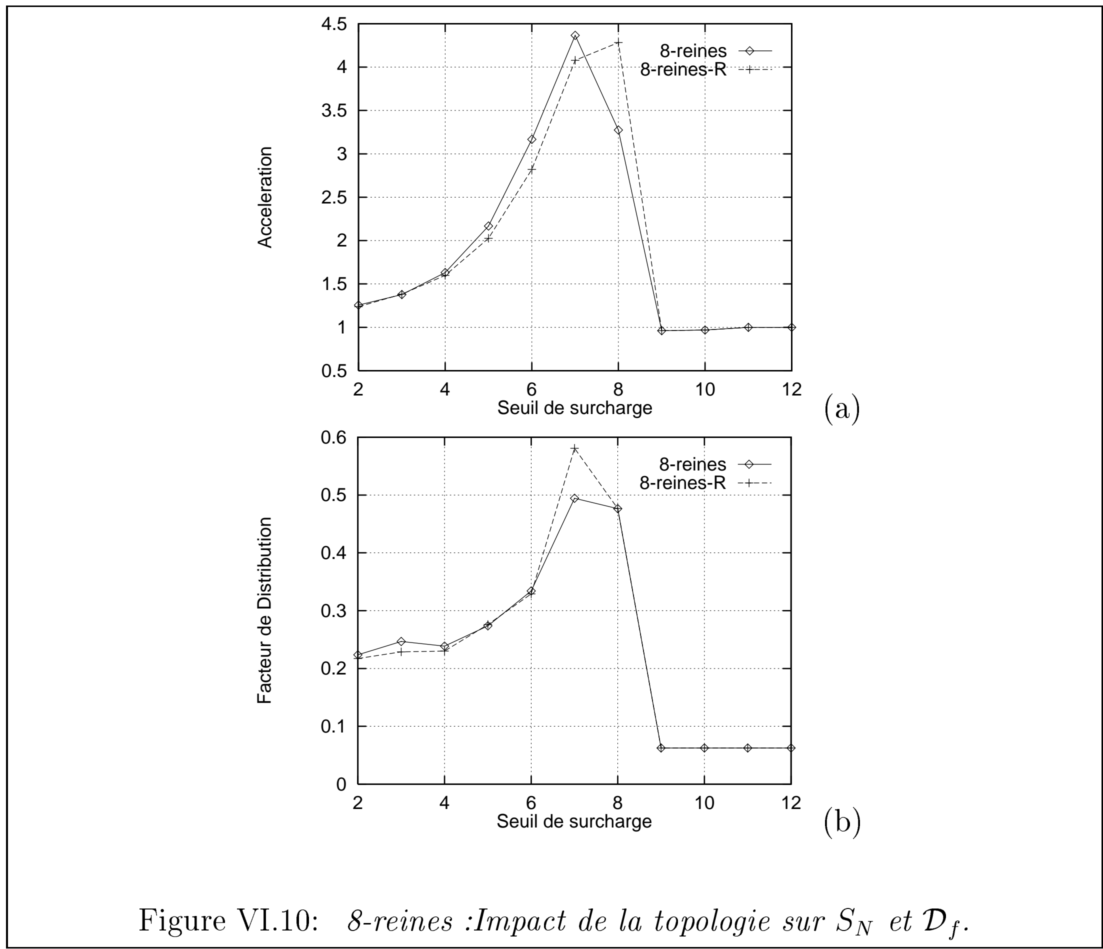


Figure VI.10: *8-reines :Impact de la topologie sur S_N et $\mathcal{D}_f$.*

Nous pouvons analyser ces deux approches de stratégies de localisation en fonction de l'accélération et à de l' amélioration du facteur de distribution des tâches telles qu'elles sont représentées sur les figures VI.9 et VI.10.

Nous constatons qu'il est possible d'obtenir de meilleurs facteurs de distribution en tenant compte des caractéristiques du réseau d'interconnexion alors que l'on ne remarque pas une amélioration sur le temps d'exécution (accélération).

VI.5 Conclusion : Bilan

Le contexte expérimental que nous avons présenté dans ce chapitre considère une classe de programmes Prolog au comportement relativement équilibré. Les

[2] *Si aucun voisin immédiat du processeur P_{src} n'est oisif, l'opération de régulation de charge n'est pas déclenchée.*

contraintes architecturales de la machine cible ne nous ont pas permis d'utiliser un large éventail de programmes de test. Toutefois, les premiers résultats obtenus, en utilisant des petits programmes Prolog, démontrent bien que les performances du système dépendent étroitement des valeurs données aux paramètres de notre fonction de régulation. Nous nous sommes particulièrement intéressés à l'étude des points suivants :

1. Quel est l'apport d'une stratégie d'évaluation fondée sur un échantillonnage périodique de l'état de charge d'un processeur, comparée à une stratégie fondée sur les événements de création (terminaison) de tâches ?
2. Quel est l'incidence des valeurs de seuil sur les performances d'une stratégie de régulation ?
3. Quel est l'apport d'une heuristique de contrôle de granularité fondée sur la sélection d'un groupe (sous-ensemble) de tâches ?
4. Quel est l'impact d'une topologie sur les performances d'une stratégie de régulation ?

En réalité, chacun de ces points identifie un paramètre de notre fonction de régulation de charge. L'analyse de ces points fait ressortir les éléments suivants :

1. L'utilisation d'une stratégie d'évaluation fondée sur un échantillonnage périodique de l'état de charge d'un processeur réduit le coût des communications de l'information de charge et améliore les performances d'exécution par rapport à une stratégie fondée sur les événements de création (terminaison) de tâches pour des valeurs de seuil faible.
2. L'utilisation d'une stratégie d'activation à seuils fixes pose le problème d'identification des valeurs appropriées correspondantes avant l'exécution (statiquement) de chaque programme. En effet, nous avons pu noter que la fixation des paramètres de seuillage appropriés dépend des caractéristiques du programme (granularité des tâches, nombre d'alternatives). Des valeurs de seuil efficaces pour un programme donné ne le sont pas systématiquement pour un autre programme.
3. L'utilisation d'une stratégie de contrôle de granularité fondée sur la sélection d'un sous-ensemble de tâches n'améliore pas nécessairement les performances d'exécution. Contrairement à notre intuition de départ, le partage de la charge de travail avec une heuristique de regroupement des tâches n'engendre pas automatiquement un gain de performances par rapport à la stratégie de sélection d'une seule tâche.

4. L'utilisation d'une stratégie de régulation tenant compte de la topologie physique d'interconnexion des processeurs permet d'améliorer l'efficacité de la distribution uniforme des tâches sur les processeurs. Cependant, cette distribution efficace ne s'accompagne pas toujours d'une amélioration des performances d'exécution.

Par conséquent, la performance de notre système logique OU-parallèle PLoSys dépend étroitement des valeurs attribuées aux paramètres de notre fonction de régulation. Ces valeurs doivent être adaptées pour chaque comportement de programme Prolog.

Chapitre VII

Conclusions et Perspectives

VII.1 Conclusions sur le travail effectué

Les travaux présentés dans cette thèse se situent dans le domaine conjoint du **Parallélisme** et de la **Programmation Logique**. Notre étude porte sur un des points clefs de l'exploitation **efficace** du parallélisme OU en programmation logique sur des architectures sans mémoire commune de type NORMA. L'efficacité d'un système logique OU-parallèle réside dans l'élaboration d'une règle appropriée d'attribution de travail aux processeurs. Les travaux décrits dans cette thèse avaient pour objectifs d'apporter des éléments de réponse aux problèmes suivants :

- Dans quelle mesure une règle générale d'équilibrage de charge est-elle bonne ?
- Existe-t-il une règle appropriée à un programme donné ? A une classe de programmes ?
- Cette règle peut-elle s'adapter dynamiquement à différentes classes de programmes ?

Se poser ces questions revient à poser une autre question : comment évaluer une fonction de régulation de charge ?

Une première phase de notre travail a porté sur la conception et le développement d'un environnement d'évaluation de stratégies de régulation. Dans une deuxième phase, une fonction de régulation de charge a été proposée et évaluée sur cet environnement.

L'approche que nous avons retenue pour la mise en œuvre de la plate-forme d'évaluation repose sur une technique de modélisation et de prise de mesures réelles (directes) sur une architecture parallèle sans mémoire commune. Cette technique consiste à modéliser l'exécution d'un programme Prolog par un graphe de tâches acyclique que l'on place sur chaque processeur de la plate-forme. L'objectif

de cette modélisation est d'émuler le comportement de l'évaluation d'un programme Prolog donné sur l'architecture cible.

L'avantage principal de notre approche est d'offrir une grande flexibilité dans la manipulation des paramètres modélisant le graphe de tâches. Cette flexibilité permet l'émulation à la fois de plusieurs comportements de programmes Prolog et de différentes caractéristiques de machines.

Dans quelle mesure une règle d'équilibrage de charge est-elle bonne ? Il apparaît dans notre étude que l'efficacité d'une règle de partage de charge est contrainte par le grain minimum de parallélisme exploitable sur la machine cible, grain en dessous duquel un écroulement de performance se produit. Nous avons identifié les contraintes régissant cette efficacité et nous avons pu constater qu'il est en général difficile de vérifier ces contraintes.

En effet, le caractère hautement dynamique de l'évaluation des programmes Prolog pose un problème complexe, dans la mesure où il est très difficile (voire impossible) de déterminer (prédire) le grain des opportunités de parallélisation. Deux principales approches ont été menées conjointement jusqu'ici afin de déterminer de façon plus ou moins raisonnable le grain des opportunités de parallélisation :

Une première approche fondée sur l'analyse de complexité pour pondérer les prédicats du programme a été écartée étant donné que :

- la génération des estimateurs à la compilation est loin d'être précise,
- le calcul de ces estimateurs à l'exécution génère des surcoûts qui viennent s'ajouter aux coûts induits par la fonction de régulation.

L'approche que nous avons choisi d'évaluer consiste à mettre en œuvre une heuristique d'estimation des coûts de calcul à partir d'éléments observables à tout instant de l'exécution du programme. Cette approche a pour avantage la simplicité et le faible coût de mise en œuvre. Par contre, elle suppose une corrélation des éléments observables mesurés à l'exécution avec les prédictions de granularité établies. Nous avons noté dans notre étude que cette corrélation n'était pas toujours vraie car elle dépend souvent des données manipulées par le programme.

Le problème majeur qui se pose alors dans les deux approches est que la valeur fournie par une estimation de granularité est définie par une métrique abstraite qui doit être ramenée à une échelle de temps physique, de manière à ce que l'on puisse vérifier les contraintes d'une parallélisation efficace. Les travaux dans ce domaine en sont encore au stade de la recherche.

Dans la mesure où il n'existe pas encore de solution connue à ce problème, l'approche que nous avons retenue pour la définition d'une fonction de régulation de charge pour le système PLoSys a été de concevoir une heuristique permettant

d'estimer ce grain et par là-même de décider de l'opportunité de la parallélisation. Une première heuristique considère que l'état de charge observable d'un processeur à un instant donné est déterminé par le nombre de points de choix cumulés à cet instant. Un critère de sélection (seuillage) sur l'indice de charge permet de répartir les processeurs en trois catégories : *Oisif, Isolé et Surchargé*. La régulation de charge du système consiste alors à répartir dynamiquement les points de choix (alternatives), de façon à maintenir équilibrée la charge des processeurs. La seconde heuristique concerne la vérification de la contrainte minimale de granularité exploitable. Cette heuristique consiste à sélectionner un sous-ensemble non vide de tâches sur le processeur surchargé (à partir du fond de la pile des tâches en attente d'évaluation) et de le transférer sur le processeur oisif. Cette heuristique repose sur le fait que plus on est haut dans l'arbre d'évaluation plus la granularité de la tâche est susceptible d'être importante. Dès lors, l'un des aspects de nos travaux à été d'analyser l'impact de notre stratégie sur différents comportements de programmes. Il ressort de cette étude que notre méthode de régulation de charge ne produit pas toujours un gain de performance du fait que :

- l'estimation de la charge à partir du nombre de points de choix cumulés à un instant donné sur un processeur ne correspond pas toujours à une charge de calcul réelle espérée (cf. section IV.5.2),
- la granularité attendue d'un sous-ensemble de tâches ne compense pas toujours le surcoût occasionné par son transfert (cf. section IV.6.2).

Par conséquent, notre fonction de régulation de charge ne peut pas garantir que l'accroissement du parallélisme produise une augmentation systématique de performance pour tous les programmes Prolog, dans la mesure où les contraintes (cf section IV.3.1) régissant cette efficacité ne sont pas toujours vérifiées (cf. sections V.3 et VI.4).

Règle de distribution & Classe de programmes Prolog : L'étude décrite dans cette thèse analyse l'impact de notre fonction de régulation de charge sur les performances d'un système logique OU-parallèle à partir de trois classes de comportement de programmes Prolog :

- *des programmes intrinséquement séquentiels* : l'arbre d'évaluation développé par ce type de programme est dégénéré. L'exploitation du parallélisme dans ce cas engendre inévitablement une perte de performance par rapport à une exécution séquentielle dès lors que le coût de transfert des tâches d'un processeur surchargé à un processeur oisif est supérieur à leur coût de traitement (cf. section V.3),

- *des programmes développant un large spectre de recherche équilibré* : ce type de programme comporte de nombreuses opportunités de parallélisme OU de granularité importante. Les performances affichées dans ce cas sont généralement plus efficaces qu'une exécution séquentielle quelle que soit la règle de partage utilisée (cf. section V.3),

- *le cas général* : il correspond au comportement de programme Prolog offrant des opportunités de parallélisme OU avec la possibilité d'avoir des opportunités de faible granularité. Pour cette classe de programmes, nous avons pu constater que le coût engendré par l'invocation de la fonction de régulation de charge et les décisions de régulation inappropriées peuvent dégrader les performances du système. Les caractéristiques (paramètres) de la stratégie de régulation utilisée ont, dans ce cas, un impact critique sur les performances du système (cf. section VI.4).

La question qui se pose alors est de savoir : dans quelle mesure une application Prolog réelle correspond-elle à l'une de ces trois classes de comportements de programmes ? Répondre à cette question nécessite d'effectuer un ensemble de mesures significatives des comportements parallèles des applications Prolog réelles.

Est ce que cette règle peut s'adapter dynamiquement à différentes classes de programmes ? Dans la mesure où l'on arrive à déterminer les valeurs appropriées des paramètres de régulation pour chaque programme Prolog, les résultats obtenus démontrent que l'on arrive à afficher de bonnes performances d'exécution avec notre fonction de régulation. Cependant, cela suppose une connaissance a priori du comportement des programmes (le nombre de tâches, la granularité des tâches) qu'il n'est pas toujours possible de déterminer avant l'exécution du programme.

En conclusion, cette thèse a montré que l'élaboration d'une fonction de régulation de charge garantissant que l'augmentation du parallélisme s'accompagne toujours d'un accroissement d'efficacité reste un problème complexe à résoudre. Si notre étude a démontré qu'une règle fondée uniquement sur des heuristiques de régulation simples permet d'obtenir un gain de performance pour une classe de programmes Prolog comportant du parallélisme OU et présentant un comportement équilibré (cf. section VI.4), elle a aussi démontrée qu'une telle règle ne pouvait pas garantir une efficacité parallèle toujours supérieure à une efficacité séquentielle, pour toutes classes de programmes Prolog, dans la mesure où cette règle ne peut pas s'adapter à des comportements de programmes variés.

Une de nos réflexions futures devra s'interroger sur la possibilité d'adapter dynamiquement les paramètres de notre fonction de régulation aux caractéristiques d'exécution des applications Prolog réelles.

VII.2 Perspectives : Problèmes ouverts

Le travail effectué dans cette thèse ouvre la voie à de nombreux prolongements possibles dont le but absolu serait de définir une fonction de régulation de charge générique, qui puisse convenir à différentes classes de programmes Prolog. Ces perspectives concernent essentiellement le traitement des problèmes que nous avons identifiés au cours de notre étude.

Un travail à court terme serait de migrer la plate-forme d'évaluation que nous avons développée vers une machine qui permettrait d'élargir l'évaluation de notre fonction de régulation sur un éventail de programmes Prolog plus important. Ce travail aurait pour objectif, d'une part, de valider les résultats présentés dans cette thèse et d'autre part, de raffiner notre fonction de régulation de charge.

Un premier point d'amélioration possible consisterait à raffiner l'heuristique d'évaluation de la charge d'un processeur. Une approche possible serait de concevoir, à partir d'une stratégie de calcul statistique simple portant sur des mesures observables à l'exécution, une estimation plus fine de la charge du processeur. Ce calcul permettrait, à un instant donné, de mesurer (estimer) l'évolution de la charge locale à chaque processeur. Le moniteur de trace d'exécution que nous avons développé permet actuellement d'extraire, pendant l'exécution, des paramètres intéressants concernant (cf. sections V.7 et V.8) :

- le taux d'échec des transferts de charge,
- le nombre de tâches exécutées à un instant donné,
- le nombre de tâches en attente à un instant donné.

Ces paramètres peuvent être étendus par des grandeurs telles que le nombre de tâches exécutées à un instant donné et le temps passé pour ce faire, permettant ainsi d'estimer par exemple :

- la fluctuation de la charge d'un processeur (la vitesse de production et de consommation des tâches),
- la durée moyenne d'une tâche,
- le temps résiduel d'exécution des tâches en attente.

Ainsi, il serait possible de prédire le futur à partir du comportement présent et passé. Cela suppose que l'exécution d'un programme Prolog ait des périodes de comportement stationnaires. Il est donc nécessaire d'analyser, par un ensemble de mesures significatives, le comportement parallèle d'applications Prolog réelles. Notre plate-forme d'évaluation pourrait facilement être exploitée pour la visualisation des comportements parallèles de programmes Prolog.

Une autre approche dans ce sens consisterait à combiner notre heuristique de régulation avec une analyse de complexité du programme, qui serait effectuée à la compilation. Cette analyse permettrait de pondérer les opportunités de parallélisation par une fonction de coût simple. Cette dernière serait évaluée à l'exécution et permettrait de mieux estimer le grain des tâches et par là même de décider de leur parallélisation. Cette approche nécessite un travail important au niveau de la compilation pour déterminer ces fonctions de coût.

Un autre point d'amélioration consisterait à utiliser les données extraites pendant l'exécution, pour adapter dynamiquement les paramètres de notre stratégie de régulation. La question qui se pose alors revient à déterminer : quand et comment contrôler la mise à jour des paramètres de régulation ? Une voie possible vers cette adaptation serait d'utiliser, par exemple, la fluctuation de la charge du processeur et le taux d'échec des transferts comme régulateurs du seuil de surcharge. On pourrait également s'inspirer de l'approche heuristique proposée dans [ABF93] pour le contrôle de la granularité dans le domaine de la parallélisation des langages fonctionnels.

Un autre problème important à considérer pour l'implantation de notre système logique parallèle concerne la gestion des prédicats à effets de bord. En effet, la prise en compte de ces prédicats avec le respect de la sémantique séquentielle de chacun introduit de nouvelles contraintes au niveau de la régulation de charge. Il reste à savoir s'il est opportun de conserver cette sémantique dans le cadre des implémentations parallèles. Nous avons présenté brièvement les principales solutions proposées à ce problème, mais il reste un travail expérimental important à effectuer à ce sujet (cf. section III.4.2-c). On pourrait s'interroger dans cette optique sur les extensions à apporter à notre modèle d'évaluation, pour la modélisation des prédicats à effets de bord et l'étude de leur impact sur les performances de notre fonction de régulation de charge.

Notons enfin, que la flexibilité du modèle et de la plate-forme d'évaluation que nous avons développés pour l'étude de stratégies de régulation de charge pour le système PLoSys, peut être exploitée (moyennant quelques extensions) dans un cadre plus général de systèmes (applications) pouvant être modélisés sous forme de graphe de tâches [PBCR96].

Pour conclure, l'expérience acquise au cours du développement de cette thèse nous a montré la difficulté de travailler sur des machines parallèles et la nécessité d'avoir des environnements de programmation qui permettent à l'utilisateur de s'abstraire de l'aspect gestion du parallélisme. Nous pensons que la programmation logique est un moyen effectif, d'une part, d'utilisation d'une large gamme de machines parallèles de manière uniforme et, d'autre part, de développement d'applications parallèles portables. Nous continuerons à œuvrer pour la conception et la réalisation d'un environnement de programmation logique parallèle toujours plus efficace.

Annexe A

Programmes Prolog d'essai

Cette annexe présente le code des programmes Prolog utilisés dans nos travaux expérimentaux pour l'évaluation de stratégies de régulation de charge (cf. section VI.3).

A.1 Programme du 4-gasp

```
g4 :- gasp4(L), fail.
g4 :- halt.


paire([0|[]]).
paire([1|L]) :- impaire(L).
paire([0|L]) :- paire(L).


impaire([1|[]]).
impaire([0|L]) :- impaire(L).
impaire([1|L]) :- paire(L).


ligne([A1,A2,A3,A4],
      [B1,B2,B3,B4],
      [C1,C2,C3,C4]) :- impaire([A1,A2,   B2,   C1,C2   ]),
                        impaire([A1,A2,A3,B1,B3,C1,C2,C3]),
                        impaire([A2,A3,A4,B2,B4,C2,C3,C4]),
                        impaire([A3,A4,   B3,   C3,C4   ]).


haute([A1,A2,A3,A4],
      [B1,B2,B3,B4]) :- impaire([A2,   B1,B2   ]),
                        impaire([A1,A3,B1,B2,B3]),
```

```
                         impaire([A2,A4,B2,B3,B4]),
                         impaire([A3,   B3,B4   ]).


basse([G1,G2,G3,G4],
      [H1,H2,H3,H4]) :- impaire([G1,G2,      H2]),
                         impaire([G1,G2,G3,H1,H3]),
                         impaire([G2,G3,G4,H2,H4]),
                         impaire([G3,G4,   H3   ]).


gasp4([L1,L2,L3,L4]) :- haute(L1,L2),
                         ligne(L1,L2,L3),
                         ligne(L2,L3,L4),
                         basse(L3,L4).
```

A.2 Programme des 8-reines

```
q8:- q([1,2,3,4,5,6,7,8],C), fail.
q8 :- halt.


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),
```

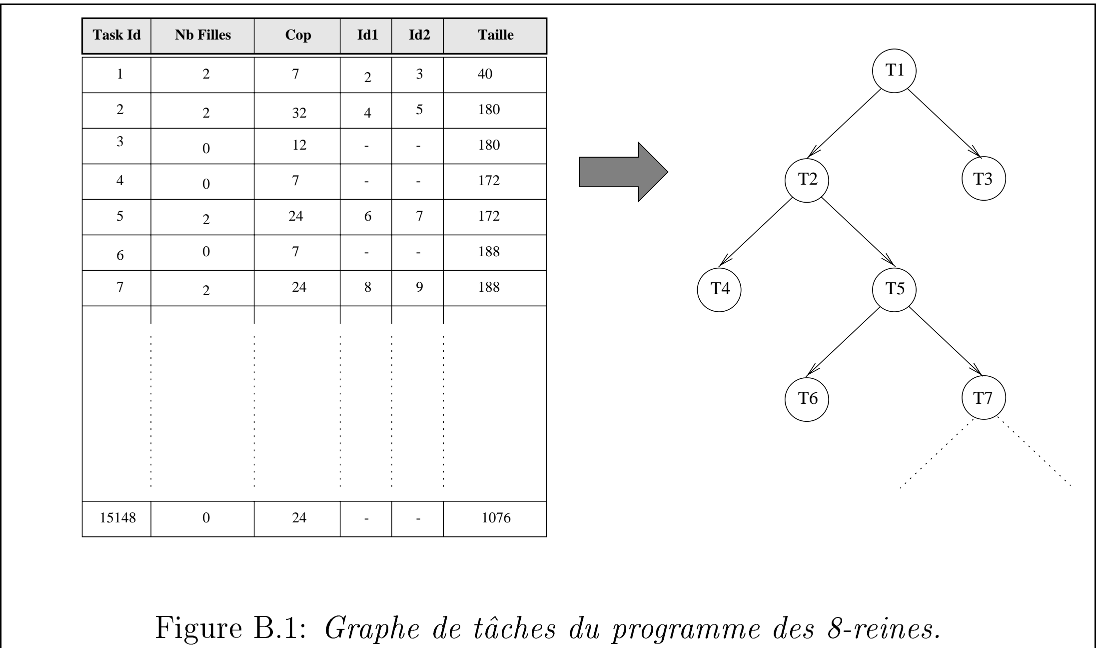
```
                nd(R,Q).


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

Annexe B

Graphes de tâches

Cette annexe présente la méthode que nous avons adoptée pour l'extraction des graphes de tâches correspondant aux programmes Prolog utilisés dans le jeu de test (cf. annexe A).

L'extraction du graphe de tâches est réalisée à partir d'une analyse post-mortem d'une trace d'exécution. Pour obtenir cette trace d'exécution, nous avons utilisé le moteur d'inférence WAMCC [Dia94].



Task Id	Nb Filles	Cop	Id1	Id2	Taille
1	2	7	2	3	40
2	2	32	4	5	180
3	0	12	-	-	180
4	0	7	-	-	172
5	2	24	6	7	172
6	0	7	-	-	188
7	2	24	8	9	188
15148	0	24	-	-	1076

Figure B.1: *Graphe de tâches du programme des 8-reines.*

Le système WAMCC comporte un compilateur qui traduit le programme source Prolog, en instructions de la machine abstraite de Warren [AK92a], avant de traduire ces mêmes instructions en langage C. L'utilisation de WAMCC est intéres-

sante, dans la mesure où ce moteur est facilement modifiable pour l'obtention des informations de trace. Ces modifications se limitent essentiellement à l'introduction de compteurs au niveau des instructions WAM (cf. table B.1) générées par le système WAMCC, afin d'annoter la trace d'exécution. Cette trace d'exécution est par la suite analysée pour construire le graphe de tâches correspondant au programme Prolog (cf. figure B.1).

Instructions	Nombre	%100	Nb / Tache
Get	6773	16.84	1.08
Put	2090	5.20	0.33
Unify	14832	36.88	2.37
Allocate	53	0.13	0.01
Deallocate	129	0.32	0.02
Call	266	0.66	0.04
Execute	2519	6.26	0.40
Proceed	356	0.89	0.06
Fail	4167	10.36	0.67
Switch	2781	6.91	0.44
Cut	1	0.00	0.00
Math	0	0.00	0.00
Functions	0	0.00	0.00
Builtins	0	0.00	0.00
Choice point	6251	15.54	1.00
Total	40218	100.00	6.43

Table B.1: *Informations de trace du programme 4-gasp.*

La trace d'exécution d'un programme permet d'extraire les informations suivantes :

- le nombre de points de choix créés,
- la profondeur de chaque point de choix,
- le nombre d'alternatives explorées à chaque point de choix,
- la taille des piles à chaque création d'un point de choix,
- le nombre d'instructions WAM exécutées par chaque alternative entre deux points de choix,

Programmes	Taille	h_{Max}	h_{Min}	δ_{Max}	δ_{Min}	δ_m	μ_{Max}	μ_{Min}	μ_m	ω_m
4-gasp	6252	85	1	263	12	25	3820	40	1945	3
8-reines	15148	47	1	352	12	71	1100	40	795	2

Table B.2: *Caractéristiques des programmes d'essai.*

La table B.2 présente les principales caractéristiques des programmes utilisés dans le jeu de test, à savoir :

- *Taille* représente la taille du graphe de tâches correspondant au programme Prolog du jeu de test,
- h_{Max} et h_{Min} décrivent respectivement la profondeur maximale et minimale d'une tâche dans le graphe,
- δ_{Max}, δ_{Min} et δ_m représentent respectivement le coût de traitement maximum, minimum et moyen d'une tâche dans le graphe (exprimé en nombre d'itérations élémentaires),
- μ_{Max}, μ_{Min} et μ_m indiquent respectivement la taille des données maximale, minimale et moyenne que l'on doit envoyer lors du transfert d'une tâche (opération de régulation de charge),
- ω_m représente le facteur de branchement moyen (nombre moyen de tâches filles par tâche) du graphe.

Annexe C

Interface Utilisateur : *PVT*

Cette Annexe présente l'outil graphique *PVT* (*PLoSys Visualization Tool*) que nous avons développé en vue de faciliter les interactions avec la machine cible (le *Méganode*).

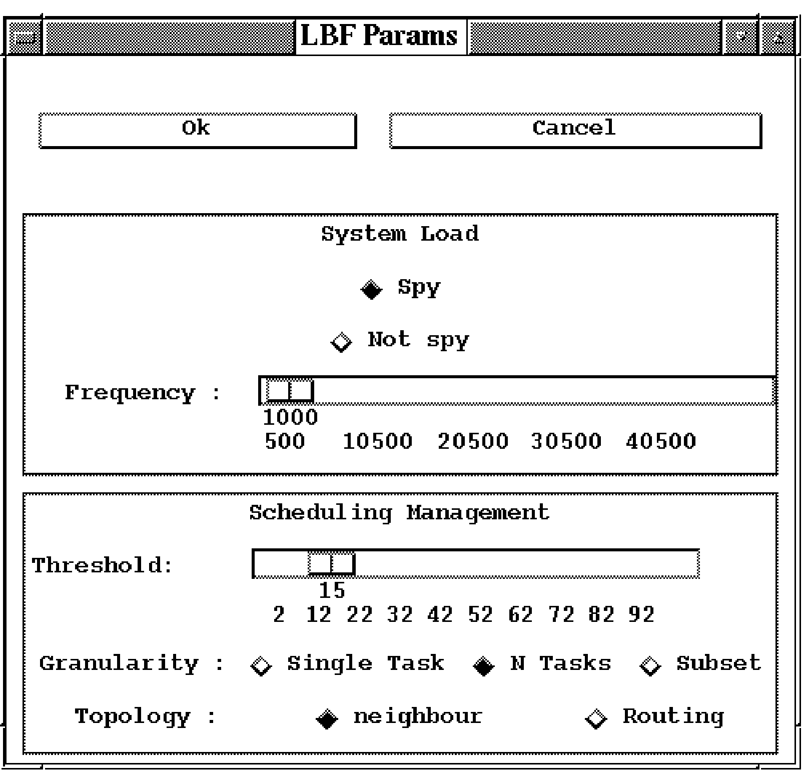


Figure C.1: *Paramètres de régulation*

PVT (cf. figure V.11) est un outil portable écrit en *Tcl/Tk* [Wel95], s'exécutant sur la machine hôte (Sun4) du *Méganode*. Les statistiques d'exécution sont vi-

sualisées en utilisant le logiciel Gnuplot [WK95]. Cet outil comprend les facilités suivantes :

- la génération de graphes de tâches, soit à partir de l'analyse de trace d'une exécution séquentielle d'un programme Prolog donné, soit à partir du générateur automatique de graphes de tâches ;
- le calibrage des paramètres des tâches en fonction des caractéristiques du processeur émulé (cf. figure C.2);
- la fixation des paramètres d'ordonnancement, pour l'exécution parallèle d'un graphe de tâches donné (cf. figure C.1);
- la visualisation des statistiques correspondantes à l'exécution du graphe de tâches, pour chaque processeur.

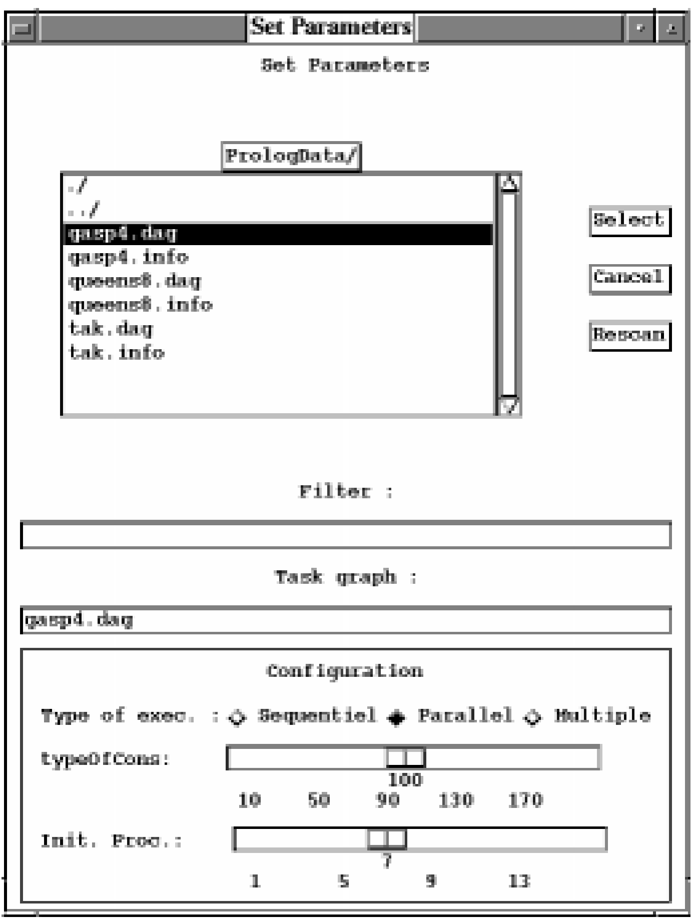


Figure C.2: *Paramètres de configuration*

Bibliographie

[Abb90] R. J. Abbott. Resourceful systems for fault tolerance, reliability and safety. *ACM Computing Surveys*, 22(1):35–68, 1990.

[ABF93] G. Aharoni, A. Barak, and Y. Farber. An adaptative granularity control algorithm for the parallel execution of functional programs. *Future Generation Computer Systems*, 9:163–174, 1993.

[Abr83] T. C. K. Chouand J. A. Abraham. Load redistribution under faillure in distributed system. *IEEE Transaction on Computers*, 32(9):799–808, September 1983.

[ACMP86] N. J. Aparicio, J. Cunha, L. Monteiro, and L. M Pareira. Delta-Prolog: a distributed backtracking extension with events. In *the Proceedings of the Third International Conference on Logic Programming*, pages 225–240, 1986. Lecture Notes in Computer Science, Vol. 225, Springer Verlag.

[AK90] Khayri A. M. Ali and Roland Karlsson. Full Prolog and scheduling OR-parallelism in MUSE. *International Journal of Parallel Programming*, 19(6):445–475, Décembre 1990.

[AK91] Khayri A. M. Ali and Roland Karlsson. Scheduling OR-parallelism in MUSE. In *Proceedings of the 1991 International Conference on Logic Programming*, Paris, Juin 1991.

[AK92a] H. Aït-Kaci. *Warren's Abstract Machine : A Tutorial Reconstruction.* MIT Press, 1992.

[AK92b] Khayri A. M. Ali and Roland Karlsson. Scheduling speculative work in MUSE and performance results. *International Journal of Parallel Programming*, 21(6), Décembre 1992.

[AKM91] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. MUSE: a parallel Prolog system. In *Proceedings of the Al-Azhar Engineering*

Second International Conference, volume XI, pages 271–282, Décembre 1991.

[AKM92a] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of MUSE on switch-based multiprocessor machine. Rapport de Recherche R92:07, Swedish Institute of Computer Science, The Royal Institute of Technology, Stockholm, Sweden, Mars 1992.

[AKM92b] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of MUSE on the BBN Butterfly TC2000. In *the Proceedings of the Conference on Parallel Architectures and Languages Europe*, Juin 1992.

[Ali86] Khayri A. M. Ali. OR-parallel execution of Prolog on a multi-sequential machine. *International Journal of Parallel Programming*, 15(3):189–214, Juin 1986.

[Ali88] Khayri A. M. Ali. OR-parallel execution of Prolog on BC-machine. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545, Seattle, WA., Aout 1988. MIT Press.

[Ali90] Khayri A. M. Ali. The MUSE approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, Avril 1990.

[AM88] H. Alshawi and D. B. Moran. The Delphi model and some preliminary experiments. In *Proc. of The Fifth Int'l. Conf. and Sym. on Logic Programming.* MIT Press, 1988.

[Bah91] R. Bahgat. *Pandora: Non-Deterministic Parallel Logic Programming.* PhD thesis, Department of Computing, Imperial College of Science and Technology, Février 1991. Published by World Scientific Publishing Co 1993.

[Bal91] H. E. Bal. Heuristic search in PARLOG using replicated work style parallelism. *Future Generation Computer Systems*, 6:303–315, 1991.

[Bar88] U. Baron. The parallel ECRC Prolog system PEPSys: an overview and evaluation of results. In *the Proceedings of the International Conference of the Fifth Generation Computer Systems*, pages 841–850, Tokyo, 1988.

[Bar90] J. Barklund. *Parallel Unification.* PhD thesis, Université d'Uppsala, 1990.

[BCHP95] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data-flow analysis of standard Prolog. In *International Conference of Logic Programming Workshop on Abstract Interpretation of Logic Languages*, Japan, Juin 1995.

[BDL+88] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling OR-parallelism: an argonne perspective. In *the Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605. MIT Press, Aout 1988.

[Bed94] R.C. Bedichek. *The Meerkat Multicomputer: Tradeoffs in Multicomputer design.* PhD thesis, Université de Washington, Juin 1994.

[BFGdK92] J. Briat, M. Favre, C. Resin Geyer, and J. Chassin de Kergommeaux. *Implementations of Distributed Prolog*, chapter OPERA: OR-parallel Prolog System on Supernode, pages 45–63. Series in Parallel Computing. Wiley Professional Computing, P. Kacsuk and M. J. Wise Editors, 1992.

[BGP95] H. Babu, G. Gupta, and E. Pontelli. Porting of the Muse Or-parallel Prolog to IPSC860. Technical report, Laboratory for Logic, Databases and Advanced Programming, New Mexico State University, September 1995. en cours de publication.

[BH95] F. Bueno and M. Hermenegildo. Analysis of concurrent constraint logic programs with fixed scheduling rule. In *International Conference of Logic Programming Workshop on Abstract Interpretation of Logic Languages*, Japan, Juin 1995.

[Bla92] B. A. Blake. Assignment of independent tasks to minimize completion time. *Software - Practice and Experience*, 22(9):723–734, September 1992.

[BLM93] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the language and its implementation. In *the Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, Juin 1993.

[Bok81] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[Bor84] P. Borgwardt. Parallel Prolog using stack segments on shared memory multiprocessors. In *the Proceedings of the 1984 International Symposium on Logic Programming*, pages 2–11, Atlantic City, NJ., 1984.

[Bra88a] Per Brand. Wavefront schedulling. Rapport interne, Gigalisp Project, 1988.

[Bra88b] I. Bratko. *Programmation en Prolog pour l'Intelligence Artificielle*. iia. InterEditions, 1988.

[BRPS91] A. J. Beaumont, S. Muthu Raman, and David H. D. Warren Péter Szeredi. Flexible scheduling OR-parallelism in Aurora: the bristol scheduler. In *the Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 403–420, Juin 1991. Lecture Notes in Computer Science, Vol. 506.

[Bru91] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10:91–124, 1991.

[BSS91] G. Bernard, D. Stève, and M. Simatic. Placement et migration de processus dans les systèmes répartis faiblement couplés. *Techniques et Science Informatiques*, 10(5), 1991.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Taneubaum. Programming languages for distributed computing systems. *Microprocessing and Microprogramming*, 20:167–172, 1989.

[BT83] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[BT92] V. Benjumea and J.M. Troya. An OR-parallel Prolog model for distributed memory systems. In *PLILP'93*, pages 291–301, Estonia, 1992.

[BW90] A. Burns and A. Wellings. *Real Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[CA88] W. F. Clocksin and H. Alshawi. A method for efficiently executing horn clause programs using multiple processors. *New Generation Computing*, 5:361–376, 1988.

[Car90] Mats Carlsson. *Design and Implementation of an OR-parallel Prolog Engine*. PhD thesis, Swedish Institute of Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1990.

[Cas81] L. M. Casey. Decentralized scheduling. *The Australian Computer Journal*, 13:58–63, 19981.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified model for static analysis of programs for construction or approximation of fix-points. In *the Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2, 3):103–179, Juillet 1992.

[CDD85] J-H Chang, A. M. Despain, and D. DeGroot. AND-parallelism of logic programs based on static data dependency analysis. In *Digest of Papers of COMPCON Spring 1985*, pages 218–225, 1985.

[CG86] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM TOPLAS*, 8(1), Janvier 1986.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some event-driven paradigms for the visualization of logic programs. In *the Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, Juin 1993.

[CH86] A. Ciepielewski and B. Hausman. Performance evaluation of an or-parallel execution model for logic programs. In *Symposium on Logic Programming*, pages 246–255, 1986.

[CH95] D. Cabeza and M. Hermenegildo. Distributed concurrent constraint execution in the CIAO system. In *Proceedings of the 1995 Compulog-Net Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, Septembre 1995.

[Cia93] P. Ciancarini. Blackbord programming in shared Prolog. In *Languages and Compilers for Parallel Computing*. MIT Press, Juin 1993.

[CK81] J. S. Conery and D. F. Kibler. Parallel interpretation of logic programs. In *the Proceedings of the conference on Functional Lanuages and Computer Architecture*, pages 163–170, Octobre 1981.

[CK83] J. S. Conery and D. F. Kibler. AND-parallelism in logic programs. In *the Proceedings of the International Joint Conference in AI*, 1983.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transaction on Software Engineering*, 14(2):141–154, Février 1988.

[CL73] C. L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[CL86] H. Chang and M. Livny. Distributed scheduling under deadline constraints: A comparaison of sender-initiated and receiver-initiated approaches. In *Proceedings of the Real-Time Symposium*, pages 175–180, December 1986.

[Col83] A. Colmerauer. Prolog, bases théoriques et développements. *Techniques et Sciences Informatiques*, 2(4):271–311, Juillet 1983.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communication of the ACM*, 33(7):69–90, 1990.

[Con83] J. S. Conery. *The AND/OR Process Model for Parallel Execution of Logic Programs*. PhD thesis, University of California, Irvine, 1983.

[Con87a] J. S. Conery. Bindings environments for parallel logic programs in Non-Shared memory multiprocessors. In *1987 IEEE International Symposium in Logic Programming*, pages 457–467, San Francisco, CA, 1987.

[Con87b] J. S. Conery. *Parallel Interpretation of Logic Programs*. Kluwer Acadimic Press, 1987.

[Cra92] J. A. Crammond. The abstract machine and implementation of parallel Prolog. *New Generation Computing*, 10(4):385–422, Aout 1992.

[CS89] A. Calderwood and Péter Szeredi. Scheduling OR-parallelism in Aurora - the manchester scheduler. In *the Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, Juin 1989.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles*. InterEditions, 1993.

[Cun94] V. D. Cung. *Contribution à l'algorithmique Non Numérique Paralléle : Exploration d'espaces de Recherche*. PhD thesis, Université de Paris VI, Avril 1994.

[CWY91] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I: a parallel Prolog system that transparently exploits both AND and OR-parallelism. In *The Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Avril 1991.

[DDG85] J. K. Peir D. D. Gajski. Essential issues in multiprocessor systems. *IEEE Computer*, pages 9–27, Juin 1985.

[Deb92] S. K. Debray. Special issue: Abstract interpretation. *Journal of Logic Programming*, 13(1,2), Juillet 1992. North-Holland.

[DeG84] D. DeGroot. Restricted and-parallelism. In *the International Conference on Fifth Generation Computer Systems*, Novembre 1984.

[DGHL94] S. K. Debray, López García, M. Hermenegildo, and N. Lin. Lower bound cost estimation for logic programs. Technical Report Clip4/94.0, UPM - Madrid, Spain, March 1994.

[DHN91] M. Debbage, M. Hill, and D. Nicole. *Virtual Channel Router: Version 2.0 User Guide*. Department of Electronics and Computer Science, University of Southampton, Octobe 1991.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *In Proceedings of the International Conference on Fifth Generation Computer Systems (ICFGCS'88)*, volume 1, Tokyo, December 1988.

[Dia94] D. Diaz. *Wamcc Prolog User's manual*. INRIA Rocencourt, Jully 1994.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. *Programmin Languages*, 1968. F. Genuys (Ed.).

[dK89] J. Chassin de Kergommeaux. *Implémentation et évaluation d'un Système Logique Parallèle*. PhD thesis, Université Joseph Fourier, Grenoble, France, Novembre 1989.

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DL84] M. Dincbas and J. P. Lepape. Meta control of logic programs in MetaLog. In *In Proceedings of the International Conference on Fifth Generation Computer Systems (ICFGCS'84)*, 1984.

[DLH90] S. K. Debray, N-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *the Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*. ACM Press, Juin 1990.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, 21(8):757–785, Aout 1991.

[DR94] S. K. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18(2):149–176, Février 1994.

[Els90] N. A. Elshiewy. Logic programming for real-time control of telecommunication switching systems. *Journal of Logic Programming*, pages 121–144, August 1990.

[Fav92] M. Favre. *Un Système Prolog Parallèle pour Machines à Mémoire Distribuée*. PhD thesis, Institut National Polytechnique, Grenoble, France, Novembre 1992.

[Fil94] A. M. Silva Filho. Prototyping real-time systems using logic programming. In *10th Logic Programming Workshop*, Zurich, 1994.

[Fly79] M. J. Flynn. Somme computer organization and their effectiveness. *IEEE Transactions on Computer*, pages 948–960, 1979.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, 1990.

[Fut93] I. Futo. Prolog with communicating processes: from T-Prolog to CSR-Prolog. In *the Proceedings of the 10th International Conference on Logic Programming*, pages 3–17, Juin 1993.

[Gey91] C. F. Resin Geyer. *Contribution à l'étude du Parallélisme OU en Prolog sur des Machines sans Mémoire Commune*. PhD thesis, Université Joseph Fourier, Grenoble, France, 1991.

[GG90] R. Gupta and P. Gopinath. A hierarchical approach to load balancing in distributed systems. In *Proceedings of the Fifth Distributed Memory Computing Conference*, volume II, pages 1000–1005, April 1990.

[GGLS91] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed systems. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144–145, Mai 1991.

[GH91] G. Gupta and H. Hermenegildo. ACE: And/OR-parallel copying-based execution of logic programs. In *ICLP'91 Workshop on Parallel Execution of Logic Programs*. Springer Verlag, 1991. Lecture Notes in Computer Science Vol. 569.

[GHD94] P. López García, M. Hermenegildo, and S. K. Debray. Towards granularity based control of parallelism in logic programs. In *Proceedings of PASCO'94*, 1994.

[GHP95] G. Gupta, M. Hermenegildo, and E. Pontelli. &ACE: A high-performance parallel Prolog system. In *IPPS'95. IEEE Computer Society*, Santa Barbara, CA, Avril 1995.

[Giu90] M. Giul. *The Control and the Execution of Parallel Logic Programs*. PhD thesis, University of Maryland, Department of Computer Science, 1990.

[GJ93a] G. Gupta and B. Jayaraman. Analysis of OR-parallel execution models. *ACM Transactions on Programming Languages*, 15(4):659–680, Septembre 1993.

[GJ93b] G. Gupta and B. Jayaraman. AND-OR parallelism on shared memory multiprocessors. *the Journal of Logic Programming*, 17(1):59–89, Octobre 1993.

[GM96] E. Godbert and G. Milhaud. Realistic natural language processing systems. In *PAP'96: Practical Application of Prolog*, London, UK, 1996.

[GO93] D. Gerogiannis and S. C. Orphanoudakis. Load balancing requierements in parallel implementations of image feature extraction tasks. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):994–1013, Septembre 1993.

[Gos91] A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.

[GP95] G. Gupta and E. Pontelli. An overview of the ACE project. In *Proceedings of the Esprit Compulog-Net Workshop on Parallelism and Implementation Technologies*, Utrecht, Septembre 1995.

[GSC92a] G. Gupta and V. Santos-Costa. Cut and side-effects in AND-OR parallel Prolog. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, 1992.

[GSC92b] G. Gupta and V. Santos-Costa. Shared paged binding array: A universal data-structure for parallel logic programming. Rapport Technique 92-CS-23, Department of Computer Science, New Mexico State University, 1992.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: integrating dependent and independent and-, and OR-parallelism. In *The International 1991 Logic Programming Symposium*, pages 152–166. MIT Press, Octobre 1991.

[GST94] H. Guyennet, F. Spies, and M. Tréhel. Modelling and simulation of dynamic load balancing using queuing theory. *Journal of Parallel Algorithms and Applications*, 5(1,2), 1994.

[GSUJ94] G. Gupta, Y. Sagiv, J. Ullman, and J. Widom Jaffar. Efficient and complete tests for database integrity constraint checking. In *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle, WA, May 1994.

[Gup91] G. Gupta. *Parallel Execution of Logic Programs on Shared Memory Multiprocessors.* PhD thesis, Université de North Carolina, Chapel Hill, 1991.

[Hau90] B. Hausman. *Pruning and Speculative Work in OR-Parallel Prolog.* PhD thesis, The Royal Institut of Technology, Stockholm, Sweden, 1990.

[Her86a] M. Hermenegildo. *An Abstract machine Based Execution Model for Computer architecture design and efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas at Austin, August 1986.

[Her86b] M. Hermenegildo. An abstract machine for restricted AND-parallel execution of logic programs. In *Third International Conference on Logic Programming*, pages 25–40, Imperial College, 1986. Springer-Verlag. Lecture Notes in Computer Science Vol. 225.

[Her94] F. Hermery. *Etude de la Répartition Dynamique d'Activités sur Architectures Décentralisées.* PhD thesis, Université des Sciences et Technologies de Lille, LIFL, Juin 1994.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog system: Exploiting independent AND-parallelism. *New Generation Computing*, 9(3, 4):233–257, 1991.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In *Proceedings of the International Conference on Logic Programming*. MIT Press, Juin 1990.

[HJM86] J. G. Harp, C. R. Jesshope, and T. Muntean. Supernode Project P1085: The development and application of low cost high performance multiprocessors machine. In *In Proceedings of ESPRIT'86: Results and Achievements*, Bruxelles, 1986.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, Aout 1992.

[Inm90] Inmos Limited. *ANSI C Toolset User Manual*, August 1990.

[Jem92] A. Jemai. *Etude d'un Processeur Risc pour un système Symbolique Parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, Juin 1992.

[JJ94] Jaffar Joxan and Maher Michael J. Constraint logic programming: A survey. *Journal of Logic Programming*, 19, 20, May 1994.

[JL92] Dean Jacobs and Anno Langen. Static analysis of logic programs for independent and-parallelism. *Journal of Logic Programming*, 13(2, 3):291–314, Juillet 1992.

[Kac90] P. Kacsuk. *Execution Models of Prolog for Parallel Computers*. MIT Press, 1990. Research Monograph.

[Kac93] P. Kacsuk. LOGFLOW-2: A transputer based data driven parallel Prolog machine. In *Proceedings of the 1993 World Transputer Congress*, pages 1154–1169, Aachen, 1993.

[Kac94a] P. Kacsuk. LOGFLOW: Prolog on massively parallel machines. In *Proceedings of the th International Conference on AI and Automation-Control Systems of Robots*, pages 71–80, 1994.

[Kac94b] P. Kacsuk. Wavefront scheduling in LOGFLOW. In *Proceedings of EuroMicro Workshop on Parallel and Distributed Processin*, pages 503–510, Malaga, Spain, 1994.

[Kal85] L. V. Kalé. *Parallel Architectures for Problem Solving*. PhD thesis, Department of Computer Science, SONY-Stony Brook, 1985.

[Kal87] L. V. Kalé. The Reduce-OR process model for parallel evaluation of logic prorams. In *Proceedings of the Fourth International Conference of Loic Programming*, pages 616–632, Melbourne, May 1987.

[Kan90] S. E. Kannat. Etude des techniques d'optimisation de la compilation du langage Prolog. Dea d'informatique, Institut National Polytechnique, Grenoble, Juin 1990.

[Kan94] S. E. Kannat. Plate-forme d'évaluation de stratégies de régulation de charge : Chaîne de développement. Rapport interne PLoSys, LMC-IMAG, Grenoble, France, 1994.

[Kan95] S. E. Kannat. An environment to study dynamic load balancing functions and its application to the parallel logic system PLoSys. In *Esprit Compulog-Net Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, Septembre 1995.

[Kar92] R. Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 1992.

[KC87] J. F. Kurose and R. Chipalkatti. Load sharing in soft real-time distributed computer systems. *IEEE Transaction on Computers*, C-36:993–1000, 1987.

[KC95] S. E. Kannat and A. S. Carissimi. Farramenta de avaliacao de estrategias de balanceamento dinamico de charga. In *Congress of the Brazilian Computer Society, XXII Software and Hardware Symposium and Panel'95 - XXI Latin American Conference on Informatics*, pages 587–597, Canel, RG, Brasil, Aout 1995.

[KCMB96] S. E. Kannat, A. S. Carissimi, E. Morel, and J. Briat. Task scheduling for parallel execution of logic programs on distributed memory architectures. In *Proceedings of the International Conference on Telecommunication, Distribution and Parallelism*, pages 123–139, France, Juin 1996.

[KE92] P. Kacsuk and M. J. Wise (Ed.). *Implementations of Distributed Prolog.* Parallel Computing. Wiley Professional Computing, 1992.

[KG92] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. Technical Report TR-91-18 (Revised), Univerity of Minnesota, Department of Computer Science, Minneapolis, Novembre 1992.

[KKMB94] S. E. Kannat, J. P. Kitajima, E. Morel, and J. Briat. A platform to study dynamic load balancing functions for parallel logic systems. In *Proceedings of the IEEE International Workshop on Parallel Processing*, pages 580–585, Bangalore, India, Décembre 1994.

[KL88] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software Engineering*, January 1988.

[KL94] S. E. Kannat and Y. Laribi. Régulation dynamique de charge et tolérance aux pannes dans les architectures distribuées à base de micro-noyau. In *Journées des Jeunes Chercheurs en Architectures de Machines, Systèmes*, pages 65–71, Monastir, Tunisie, Décembre 1994.

[Kle85] L. Kleinrock. Distributed systems. *Communication of the ACM*, 28:1200–1213, November 1985.

[KM94] S. E. Kannat and E. Morel. Régulation dynamique de charge : etude, analyse, plate-forme d'évaluation pour le système logique parallèle PLoSys. In *Journées des Jeunes Chercheurs en Architectures de Machines, Systèmes*, pages 87–97, Monastir, Tunisie, Décembre 1994.

[KMB94] S. E. Kannat, E. Morel, and J. Briat. Régulation dynamique de charge dans PLoSys. In *6ème Rencontres Francophones du Parallélisme*, page 308, ENS Lyon, Juin 1994.

[KMB95] S. E. Kannat, E. Morel, and J. Briat. Plate-forme d'évaluation de stratégies de régulation dynamique de charge pour le système logique parallèle PLoSys. In *7ème Rencontres Francophones du Parallélisme*, pages 171–175, Mons, Belgique, May 1995.

[KN90] A. Krall and U. Neumerkel. The vienna abstract machine. In *In Proceedings of Symposium on Programming Languages Implementation and Logic Programming*. Springer-Verlag, 1990.

[Kow79a] R. Kowalski. Algorithme = logic + control. *Communication of the ACM*, 22(7):424–436, Juillet 1979.

[Kow79b] R. Kowalski. *Logic for Problem Solving*. Elsevier Science Publishing, 1979.

[KP94] J. P. Kitajima and B. Plateau. Modelling parallel program behaviour in alpes. *Information and Software Technology*, 36(7):457–464, July 1994.

[KR87] V. Kumar and V. N. Rao. Parallel depth-first search: Implementation & analysis. *International Journal of Parallel Programming*, 16(6):479–519, 1987.

[KR88] V. Kumar and K. Rameshand V. N. Rao. Parallel heuristic search of state-space graphs: A summary of results. Technical Report TR-88-14, Department of Computer Sciences, The university of Texas at Austin, April 1988.

[KR90] V. Kumar and V. N. Rao. *Parallel Algorithms for Machine Intelligence and Vision*, chapter Scalable Parallel Formulations of Depth-First Search. Springer-Verlag, 1990.

[Kum86] K. Kumon. Kabu-wake: A new parallel method and its evaluation. In *Proceedings of CompCon'86*, pages 168–172, 1986.

[Kur88] T. Kurozumi. Present status and plans for research and development. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 3–15, Tokyo, 1988.

[KV74] R. Kowalski and M. VanEmden. The semantics of predicate logic as programming language. *Journal of the ACM*, 23(4):733–742, Octobre 1974.

[Lim88] Inmos Limited. *Occam 2 Reference Manual*. International Series in Computer Science. Prentice Hall, 1988.

[Lin88] Yow-Jian Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, Austin, Aout 1988.

[Lin91] Zheng Lin. A distributed task scheduling sheme for parallel execution of logic programs. Technical Report CS-TR-2674, University of Maryland, Department of Computer Science, 1991.

[LK87] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[LK91] Yow-Jian Lin and Vipin Kumar. AND-parallel execution of logic programs on a shared-memory multiprocessor. *Journal of Logic Programming*, 10:155–178, Janvier 1991.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LW90] G-. J. Li and B. W. Wah. Computational efficiency of parallel combinatorial OR-tree searches. *IEEE Transactions on Software Engineering*, 16(1):13–31, Janvier 1990.

[LWH90] E. Lusk, D. H. D. Warren, and Seif Haridi. The Aurora OR-parallel Prolog system. *New Generation Computing*, 7(2, 3):243–271, 1990.

[MdlBH94] K. Marriot, M. García de la Banda, and M. Hermenegildo. Analysing logic programs with dynamic scheduling. In *Proceedings of the 20th. Annual Conference on Principles of Programming Languages*. ACM, 1994.

[MG89] C. McGreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32, 1989.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2, 3):315–347, Juillet 1992.

[Mil93] D. S. Milojičić. *Load Distribution: Implementation for the Mach Microkernel*. Verlag Viewg, 1993.

[Mor96] E. Morel. *Environnement de Programmation Parallèle : Application à Prolog*. PhD thesis, Université Joseph Fourier, Grenoble, Novembre 1996.

[MT91] T. Muntean and E.-G. Talbi. Méthodes de placement statiques de processus sur architectures parallèles. *TSI*, 10(5):355–373, 1991.

[Nil90] M. Nilsson. Mobile robot control with concurrent logic languages. In *Proceedings of Euromicro Workshop on Real-time*, pages 42–46, 1990.

[Nor96] M. Norman. The grand challenge cosmology consortium. Technical report, MIT, 1996. http://zeus.ncsa.uiuc.edu.

[Oud87] O. Oudot. *Utilisation des Modes Directionnels dans la Résolution*. PhD thesis, Institut National Polytechnique, Grenoble, France, 1987.

[Paa88] J. A. Paaki. Note on the speed of Prolog. *SIGPLAN Notices*, 23(8), 1988.

[Pas87] D. Pase. Load balancing heuristics and network topologies for distributed evaluation of Prolog. Technical Report CS/E 87-005, Oregon Graduate Center, Avril 1987.

[PBCR96] B. Plateau, J. Briat, M. Christaller, and J. L. Roch. ATHAPASCAN : Programmation parallèle et regulation de charge. Rapport Appache, LMC- IMAG, Grenoble, France, 1996. http://www-apache.imag.fr/apache.

[Pel87] S. Pelhat. Les boucles dans Prolog : Structures et Origines. In *Actes du Séminaire de Programmation Logique*, pages 153–172, Trégastel, Mai 1987.

[Rei88] P. B. A. Reintjes. VLSI design environment in Prolog. In *In Proceedings of the fifth International Conference on Logic Programming*, pages 70–81, Seattle, 1988.

[RK90] T. J. Reynolds and P. Kefalas. OR-parallel Prolog and search problems in AI applications. In *North American Conference on Logic Programming*, Austin, October 1990.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, Janvier 1965.

[Rob88] Ph. Robert. *Une Machine Abstraite pour la Mise en Œuvre du Parallélisme OU/ET en Programmation Logique.* PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, Juin 1988.

[Rou75] P. Roussel. *Prolog : Manuel de Référence et d'Utilisation.* GIA, Université d'Aix-Marseille II, Septembre 1975.

[Row88] N. C. Rowe. *Artificial Intelligence Through Prolog.* Prentice-Hall, 1988.

[Roy84] P. Van Roy. A Prolog compiler for the PLM. Technical Report UCB/CSD 84/203, University of California at Berkeley, EECS, 1984.

[Roy90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming ?* PhD thesis, University of California at Berkeley, Novembre 1990.

[Roz88] M. Rozier. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, Décembre 1988.

[Sal90] V. A. Saletore. A distributed and adaptative dynamic load balancing scheme for parallel processing medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference*, volume II, pages 994–999, April 1990.

[SCY91] P. Szeredi, M. Carlsson, and Rong Yang. Interfacing engines and schedulers in OR-parallel Prolog systems. In *Proceedings of the Conference on Parallel Architectures and Lanuages Europe (PARL'91)*. Springer-Verlag, Juin 1991. Lecture Notes in Computer Science, Vol. 506.

[Sea75] B. D. Searls. The linguistics of DNA. *American Scientist*, 80(6):579–591, November 1975.

[Sea92] B. D. Searls. Prolog and the human genome project. In *International Conference on Practical Prolog Apllications*, London, April 1992.

[SH91] K. Shen and M. Hermenegildo. A simulation study of OR- and independent AND-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, Octobre 1991.

[SH96] M. Siormanolakis and J. Hermle. Using Prolog for a railway control system. In *PAP'96: Practical Application of Prolog*, London, UK, 1996.

[Sha87] E. Shapiro. *Concurrent Prolog : Collected Papers*. MIT Press, 1987.

[Sha89] E. Shapiro. The family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, Septembre 1989.

[She92a] K. Shen. Exploiting dependent AND-parallelism in Prolog: the dynamic, dependent AND-parallel scheme. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

[She92b] K. Shen. *Studies in AND/OR Parallelism*. PhD thesis, University of Cambridge, 1992.

[Sho94] Y. Shoham. *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1994.

[Sin92] Raéd Sindaha. The Dharma Scheduler-Definitive scheduling in Aurora on multiprocessors architecture. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 296–303. IEEE Computer Society Press, Décembre 1992.

[SJ96] D. Sedlock and J. Jorg. Managing software projects with Prolog and the WWW. In *PAP'96: Practical Application of Prolog*, London, UK, 1996.

[SKS92] M. G. Shivarattri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, Décembre 1992.

[Smi93] D. A. Smith. MultiLog: Data OR-parallel logic programming. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 314–332. MIT Press, Juin 1993.

[SP89] A. Singhal and Y. Patt. Unification parallelism: How much can be exploited ? In *Proceedings of the North American Conference on Logic Programming*, pages 1135–1148. MIT Press, 1989.

[SS86] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press Series in Logic Programming, 1986.

[Sta89] J. A. Stankovic. Decentralized decision making in hard real-time systems. *IEEE Transactions on Computers*, C-38:341–355, 1989.

[Sze91a] P. Szeredi. Solving optimisation problems in the Aurora parallel Prolog system. In *Proc. of Parallel Execution of Logic Programs ICLP'91 Pre-conf. Work.*, June 1991.

[Sze91b] P. Szeredi. Using dynamic predicates in an OR-parallel Prolog system. In *Proceedings of the Int. Conf. and Sym. on Logic Programming*. MIT Press, 1991.

[Tan92] A. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[Tar95] P. Tarau. The binProlog experience: Implementing a high-performance continuation passing Prolog engine. Technical report, Département d'Informatique, Université de Moncton, 1995.

[Tic91] E. Tick. *Parallel Logic Programming.* MIT Press, 1991.

[Tic92] E. Tick. Visualizing parallel logic programming with VISTA. In *Proceedings of the International Conference of the Fifth Generation Computer Systems*, pages 934–942, Tokyo, 1992.

[Tin88] P. Tinker. Performance of an parallel logic programming system. *International Journal of Parallel Programming*, 17(1):59–92, 1988.

[Ued86] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, 1986.

[War77] D. H. D. Warren. Implementing Prolog - compiling logic programs. Research Report 39, 40, University of Edinburg, 1977.

[War83] D. H. D. Warren. An abstract instruction set for Prolog. Tech. note 309, SRI International, 1983.

[War87] D. H. D. Warren. The SRI model for OR-parallel execution of Prolog-abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102. IEEE Computer Society Press, 1987.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*, January 1995. Draft Updated for Tcl 7.4 and Tk 4.0.

[Wes87] H. Westphal. The PEPSys model: Combining backtracking, AND- and OR-parallelism. In *1987 IEEE International Symposium in Logic Programming*, pages 436–448, San Francisco, CA, 1987.

[Wis86] D. S. Wise. *Prolog Multiprocessors*. Prentice-Hall, 1986.

[WK95] T. Williams and C. Kelly. *GNUPLOT 3.5: Interactive Plotting Program*, 1995.

[WM85] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34:204–217, March 1985.

[XH90] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in message-passing supercomputer. In *Proceedings of the Supercomputing'90*, pages 888–897, November 1990.

[YCD94] M. K. Yang, R. Chita, and R. Das. Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):74–86, January 1994.

[YN84] H. Yasuhara and K. Nitadori. ORBIT: A parallel computing model of Prolog. *New Generation Computing*, 2:227–228, 1984.

[YS89] M. H. Schultz Y. Saad. Data communication in parallel architectures. *Journal of Parallel Computing*, 11, 1989.

[ZF88] S. Zhou and D. Ferrari. A trace driven simulation study of dynamic load balancing. *IEEE Transactions on Software engineering*, 14:1327–1341, September 1988.

[ZT92] X. Zhong and E. Tick. Towards an efficient compile-time granularity algorithm. In *International Conference on Fifth Generetion Computer System*, pages 809–816, Tokyo, 1992.