USTL
FLANDRES ARTOIS

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

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° Ordre: 877

Année: 1992

# THESE

Nouveau régime

présentée à

**l'Université des Sciences et Techniques de LILLE FLANDRES ARTOIS**

pour obtenir le titre de

**DOCTEUR en INFORMATIQUE**

par

**Abdelhafid BOURZOUFI**



# Définition et Evaluation d'une Machine Abstraite Parallèle pour un Modèle OU-parallèle Multi-séquentiel de PROLOG.

Soutenue le 14 Février 1992 devant la commission d'examen

Membres du Jury:

| | |
|---|---|
| Président : | M. MERIAUX |
| Rapporteur : | J. P. DELAHAYE |
| Rapporteur : | C. PERCEBOIS |
| Directeur de thèse : | B. TOURSEL |
| Examinateur : | P. DEVIENNE |
| Examinateur : | G. GONCALVES |
| Examinateur : | P. LECOUFFE |

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

---

## DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

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

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

## PROFESSEUR EMERITE

M. A. LEBRUN

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

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

## PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

## PROFESSEURS - CLASSE EXCEPTIONNELLE

| | |
|---|---|
| M. CHAMLEY Hervé | Géotechnique |
| M. CONSTANT Eugène | Electronique |
| M. ESCAIG Bertrand | Physique du solide |
| M. FOURET René | Physique du solide |
| M. GABILLARD Robert | Electronique |
| M. LABLACHE COMBIER Alain | Chimie |
| M. LOMBARD Jacques | Sociologie |
| M. MACKE Bruno | Physique moléculaire et rayonnements atmosphériques |

| | |
|---|---|
| M. MIGEON Michel | EUDIL |
| M. MONTREUIL Jean | Biochimie |
| M. PARREAU Michel | Analyse |
| M. TRIDOT Gabriel | Chimie appliquée |

## PROFESSEURS - 1ère CLASSE

| | |
|---|---|
| M. BACCHUS Pierre | Astronomie |
| M. BIAYS Pierre | Géographie |
| M. BILLARD Jean | Physique du Solide |
| M. BOILLY Bénoni | Biologie |
| M. BONNELLE Jean Pierre | Chimie-Physique |
| M. BOSCQ Denis | Probabilités |
| M. BOUGHON Pierre | Algèbre |
| M. BOURIQUET Robert | Biologie Végétale |
| M. BRASSELET Jean Paul | Géométrie et topologie |
| M. BREZINSKI Claude | Analyse numérique |
| M. BRIDOUX Michel | Chimie Physique |
| M. BRUYELLE Pierre | Géographie |
| M. CARREZ Christian | Informatique |
| M. CELET Paul | Géologie générale |
| M. COEURE Gérard | Analyse |
| M. CORDONNIER Vincent | Informatique |
| M. CROSNIER Yves | Electronique |
| Mme DACHARRY Monique | Géographie |
| M. DAUCHET Max | Informatique |
| M. DEBOURSE Jean Pierre | Gestion des entreprises |
| M. DEBRABANT Pierre | Géologie appliquée |
| M. DECLERCQ Roger | Sciences de gestion |
| M. DEGAUQUE Pierre | Electronique |
| M. DESCHEPPER Joseph | Sciences de gestion |
| Mme DESSAUX Odile | Spectroscopie de la réactivité chimique |
| M. DHAINAUT André | Biologie animale |
| Mme DHAINAUT Nicole | Biologie animale |
| M. DJAFARI Rouhani | Physique |
| M. DORMARD Serge | Sciences Economiques |
| M. DOUKHAN Jean Claude | Physique du solide |
| M. DUBRULLE Alain | Spectroscopie hertzienne |
| M. DUPOUY Jean Paul | Biologie |
| M. DYMENT Arthur | Mécanique |
| M. FOCT Jacques Jacques | Métallurgie |
| M. FOUQUART Yves | Optique atmosphérique |
| M. FOURNET Bernard | Biochimie structurale |
| M. FRONTIER Serge | Ecologie numérique |
| M. GLORIEUX Pierre | Physique moléculaire et rayonnements atmosphériques |
| M. GOSSELIN Gabriel | Sociologie |
| M. GOUDMAND Pierre | Chimie-Physique |
| M. GRANELLE Jean Jacques | Sciences Economiques |
| M. GRUSON Laurent | Algèbre |
| M. GUILBAULT Pierre | Physiologie animale |
| M. GUILLAUME Jean | Microbiologie |
| M. HECTOR Joseph | Géométrie |
| M. HENRY Jean Pierre | Génie mécanique |
| M. HERMAN Maurice | Physique spatiale |
| M. LACOSTE Louis | Biologie Végétale |
| M. LANGRAND Claude | Probabilités et statistiques |

| | |
|---|---|
| M. LATTEUX Michel | Informatique |
| M. LAVEINE Jean Pierre | Paléontologie |
| Mme LECLERCQ Ginette | Catalyse |
| M. LEHMANN Daniel | Géométrie |
| Mme LENOBLE Jacqueline | Physique atomique et moléculaire |
| M. LEROY Jean Marie | Spectrochimie |
| M. LHENAFF René | Géographie |
| M. LHOMME Jean | Chimie organique biologique |
| M. LOUAGE Françis | Electronique |
| M. LOUCHEUX Claude | Chimie-Physique |
| M. LUCQUIN Michel | Chimie physique |
| M. MAILLET Pierre | Sciences Economiques |
| M. MAROUF Nadir | Sociologie |
| M. MICHEAU Pierre | Mécanique des fluides |
| M. PAQUET Jacques | Géologie générale |
| M. PASZKOWSKI Stéfan | Mathématiques |
| M. PETIT Francis | Chimie organique |
| M. PORCHET Maurice | Biologie animale |
| M. POUZET Pierre | Modélisation - calcul scientifique |
| M. POVY Lucien | Automatique |
| M. PROUVOST Jean | Minéralogie |
| M. RACZY Ladislas | Electronique |
| M. RAMAN Jean Pierre | Sciences de gestion |
| M. SALMER Georges | Electronique |
| M. SCHAMPS Joël | Spectroscopie moléculaire |
| Mme SCHWARZBACH Yvette | Géométrie |
| M. SEGUIER Guy | Electrotechnique |
| M. SIMON Michel | Sociologie |
| M. SLIWA Henri | Chimie organique |
| M. SOMME Jean | Géographie |
| Melle SPIK Geneviève | Biochimie |
| M. STANKIEWICZ François | Sciences Economiques |
| M. THIEBAULT François | Sciences de la Terre |
| M. THOMAS Jean Claude | Géométrie - Topologie |
| M. THUMERELLE Pierre | Démographie - Géographie humaine |
| M. TILLIEU Jacques | Physique théorique |
| M. TOULOTTE Jean Marc | Automatique |
| M. TREANTON Jean René | Sociologie du travail |
| M. TURRELL Georges | Spectrochimie infrarouge et raman |
| M. VANEECLOO Nicolas | Sciences Economiques |
| M. VAST Pierre | Chimie inorganique |
| M. VERBERT André | Biochimie |
| M. VERNET Philippe | Génétique |
| M. VIDAL Pierre | Automatique |
| M. WALLART Françis | Spectrochimie infrarouge et raman |
| M. WEINSTEIN Olivier | Analyse économique de la recherche et développement |
| M. ZEYTOUNIAN Radyadour | Mécanique |

## PROFESSEURS - 2ème CLASSE

| | |
|---|---|
| M. ABRAHAM Francis | |
| M. ALLAMANDO Etienne | Composants électroniques |
| M. ANDRIES Jean Claude | Biologie des organismes |
| M. ANTOINE Philippe | Analyse |
| M. BALL Steven | Génétique |
| M. BART André | Biologie animale |
| M. BASSERY Louis | Génie des procédés et réactions chimiques |
| Mme BATTIAU Yvonne | Géographie |
| M. BAUSIERE Robert | Systèmes électroniques |
| M. BEGUIN Paul | Mécanique |
| M. BELLET Jean | Physique atomique et moléculaire |
| M. BERNAGE Pascal | Physique atomique, moléculaire et du rayonnement |
| M. BERTHOUD Arnaud | Sciences Economiques |
| M. BERTRAND Hugues | Sciences Economiques |
| M. BERZIN Robert | Analyse |
| M. BISKUPSKI Gérard | Physique de l'état condensé et cristallographie |
| M. BKOUCHE Rudolphe | Algèbre |
| M. BODARD Marcel | Biologie végétale |
| M. BOHIN Jean Pierre | Biochimie métabolique et cellulaire |
| M. BOIS Pierre | Mécanique |
| M. BOISSIER Daniel | Génie civil |
| M. BOIVIN Jean Claude | Spectrochimie |
| M. BOUCHER Daniel | Physique |
| M. BOUQUELET Stéphane | Biologie appliquée aux enzymes |
| M. BOUQUIN Henri | Gestion |
| M. BROCARD Jacques | Chimie |
| Mme BROUSMICHE Claudine | Paléontologie |
| M. BUISINE Daniel | Mécanique |
| M. CAPURON Alfred | Biologie animale |
| M. CARRE François | Géographie humaine |
| M. CATTEAU Jean Pierre | Chimie organique |
| M. CAYATTE Jean Louis | Sciences Economiques |
| M. CHAPOTON Alain | Electronique |
| M. CHARET Pierre | Biochimie structurale |
| M. CHIVE Maurice | Composants électroniques optiques |
| M. COMYN Gérard | Informatique théorique |
| Mme CONSTANT Monique | Composants électroniques et optiques |
| M. COQUERY Jean Marie | Psychophysiologie |
| M. CORIAT Benjamin | Sciences Economiques |
| Mme CORSIN Paule | Paléontologie |
| M. CORTOIS Jean | Physique nucléaire et corpusculaire |
| M. COUTURIER Daniel | Chimie organique |
| M. CRAMPON Norbert | Tectolique géodynamique |
| M. CURGY Jean Jacques | Biologie |
| M. DANGOISSE Didier | Physique théorique |
| M. DE PARIS Jean Claude | Analyse |
| M. DECOSTER Didier | Composants électroniques et optiques |
| M. DEJAEGER Roger | Electrochimie et Cinétique |
| M. DELAHAYE Jean Paul | Informatique |
| M. DELORME Pierre | Physiologie animale |
| M. DELORME Robert | Sciences Economiques |
| M. DEMUNTER Paul | Sociologie |
| Mme DEMUYNCK Claire | Physique atomique, moléculaire et du rayonnement |
| M. DENEL Jacques | Informatique |
| M. DEPREZ Gilbert | Physique du solide - christallographie |

| | |
|---|---|
| M. DERIEUX Jean Claude | Microbiologie |
| M. DERYCKE Alain | Informatique |
| M. DESCAMPS Marc | Physique de l'état condensé et cristallographie |
| M. DEVRAINNE Pierre | Chimie minérale |
| M. DEWAILLY Jean Michel | Géographie humaine |
| M. DHAMELINCOURT Paul | Chimie physique |
| M. DI PERSIO Jean | Physique de l'état condensé et cristallographie |
| M. DUBAR Claude | Sociologie démographique |
| M. DUBOIS Henri | Spectroscopie hertzienne |
| M. DUBOIS Jean Jacques | Géographie |
| M. DUBUS Jean Paul | Spectrométrie des solides |
| M. DUPONT Christophe | Vie de la firme |
| M. DUTHOIT Bruno | Génie civil |
| Mme DUVAL Anne | Algèbre |
| Mme EVRARD Micheline | Génie des procédés et réactions chimiques |
| M. FAKIR Sabah | Algèbre |
| M. FARVACQUE Jean Louis | Physique de l'état condensé et cristallographie |
| M. FAUQUEMBERGUE Renaud | Composants électroniques |
| M. FELIX Yves | Mathématiques |
| M. FERRIERE Jacky | Tectonique - Géodynamique |
| M. FISCHER Jean Claude | Chimie organique, minérale et analytique |
| M. FONTAINE Hubert | Dynamique des cristaux |
| M. FORSE Michel | Sociologie |
| M. GADREY Jean | Sciences économiques |
| M. GAMBLIN André | Géographie urbaine, industrielle et démographie |
| M. GOBLOT Rémi | Algèbre |
| M. GOURIEROUX Christian | Probabilités et statistiques |
| M. GREGORY Pierre | I.A.E. |
| M. GREMY Jean Paul | Sociologie |
| M. GREVET Patrice | Sciences Economiques |
| M. GRIMBLOT Jean | Chimie organique |
| M. GUELTON Michel | Chimie physique |
| M. GUICHAOUA André | Sociologie |
| M. HAIMAN Georges | Modélisation,calcul scientifique, statistiques |
| M. HOUDART René | Physique atomique |
| M. HUEBSCHMANN Johannes | Mathématiques |
| M. HUTTNER Marc | Algèbre |
| M. ISAERT Noël | Physique de l'état condensé et cristallographie |
| M. JACOB Gérard | Informatique |
| M. JACOB Pierre | Probabilités et statistiques |
| M. JEAN Raymond | Biologie des populations végétales |
| M. JOFFRE Patrick | Vie de la firme |
| M. JOURNEL Gérard | Spectroscopie hertzienne |
| M. KOENIG Gérard | Sciences de gestion |
| M. KOSTRUBIEC Benjamin | Géographie |
| M. KREMBEL Jean | Biochimie |
| Mme KRIFA Hadjila | Sciences Economiques |
| M. LANGEVIN Michel | Algèbre |
| M. LASSALLE Bernard | Embryologie et biologie de la différenciation |
| M. LE MEHAUTE Alain | Modélisation,calcul scientifique,statistiques |
| M. LEBFEVRE Yannic | Physique atomique,moléculaire et du rayonnement |
| M. LECLERCQ Lucien | Chimie physique |
| M. LEFEBVRE Jacques | Physique |
| M. LEFEBVRE Marc | Composants électroniques et optiques |
| M. LEFEVRE Christian | Pétrologie |
| Melle LEGRAND Denise | Algèbre |
| M. LEGRAND Michel | Astronomie - Météorologie |
| M. LEGRAND Pierre | Chimie |
| Mme LEGRAND Solange | Algèbre |
| Mme LEHMANN Josiane | Analyse |
| M. LEMAIRE Jean | Spectroscopie hertzienne |

| | |
|---|---|
| M. LE MAROIS Henri | Vie de la firme |
| M. LEMOINE Yves | Biologie et physiologie végétales |
| M. LESCURE François | Algèbre |
| M. LESENNE Jacques | Systèmes électroniques |
| M. LOCQUENEUX Robert | Physique théorique |
| Mme LOPES Maria | Mathématiques |
| M. LOSFELD Joseph | Informatique |
| M. LOUAGE Francis | Electronique |
| M. MAHIEU François | Sciences économiques |
| M. MAHIEU Jean Marie | Optique - Physique atomique |
| M. MAIZIERES Christian | Automatique |
| M. MANSY Jean Louis | Géologie |
| M. MAURISSON Patrick | Sciences Economiques |
| M. MERIAUX Michel | EUDIL |
| M. MERLIN Jean Claude | Chimie |
| M. MESMACQUE Gérard | Génie mécanique |
| M. MESSELYN Jean | Physique atomique et moléculaire |
| M. MOCHE Raymond | Modélisation,calcul scientifique,statistiques |
| M. MONTEL Marc | Physique du solide |
| M. MORCELLET Michel | Chimie organique |
| M. MORE Marcel | Physique de l'état condensé et cristallographie |
| M. MORTREUX André | Chimie organique |
| Mme MOUNIER Yvonne | Physiologie des structures contractiles |
| M. NIAY Pierre | Physique atomique,moléculaire et du rayonnement |
| M. NICOLE Jacques | Spectrochimie |
| M. NOTELET Francis | Systèmes électroniques |
| M. PALAVIT Gérard | Génie chimique |
| M. PARSY Fernand | Mécanique |
| M. PECQUE Marcel | Chimie organique |
| M. PERROT Pierre | Chimie appliquée |
| M. PERTUZON Emile | Physiologie animale |
| M. PETIT Daniel | Biologie des populations et écosystèmes |
| M. PLIHON Dominique | Sciences Economiques |
| M. PONSOLLE Louis | Chimie physique |
| M. POSTAIRE Jack | Informatique industrielle |
| M. RAMBOUR Serge | Biologie |
| M. RENARD Jean Pierre | Géographie humaine |
| M. RENARD Philippe | Sciences de gestion |
| M. RICHARD Alain | Biologie animale |
| M. RIETSCH François | Physique des polymères |
| M. ROBINET Jean Claude | EUDIL |
| M. ROGALSKI Marc | Analyse |
| M. ROLLAND Paul | Composants électroniques et optiques |
| M. ROLLET Philippe | Sciences Economiques |
| Mme ROUSSEL Isabelle | Géographie physique |
| M. ROUSSIGNOL Michel | Modélisation,calcul scientifique,statistiques |
| M. ROY Jean Claude | Psychophysiologie |
| M. SALERNO Françis | Sciences de gestion |
| M. SANCHOLLE Michel | Biologie et physiologie végétales |
| Mme SANDIG Anna Margarette | |
| M. SAWERYSYN Jean Pierre | Chimie physique |
| M. STAROSWIECKI Marcel | Informatique |
| M. STEEN Jean Pierre | Informatique |
| Mme STELLMACHER Irène | Astronomie - Météorologie |
| M. STERBOUL François | Informatique |
| M. TAILLIEZ Roger | Génie alimentaire |
| M. TANRE Daniel | Géométrie - Topologie |
| M. THERY Pierre | Systèmes électroniques |
| Mme TJOTTA Jacqueline | Mathématiques |
| M. TOURSEL Bernard | Informatique |
| M. TREANTON Jean René | Sociologie du travail |

| | |
|---|---|
| M. TURREL Georges | Spectrochimie infrarouge et raman |
| M. VANDIJK Hendrik | |
| Mme VAN ISEGHEM Jeanine | Modélisation,calcul scientifique,statistiques |
| M. VANDORPE Bernard | Chimie minérale |
| M. VASSEUR Christian | Automatique |
| M. VASSEUR Jacques | Biologie |
| Mme VIANO Marie Claude | |
| M. WACRENIER Jean Marie | Electronique |
| M. WARTEL Michel | Chimie inorganique |
| M. WATERLOT Michel | géologie générale |
| M. WEICHERT Dieter | Génie mécanique |
| M. WERNER Georges | Informatique théorique |
| M. WIGNACOURT Jean Pierre | |
| M. WOZNIAK Michel | Spectrochimie |
| Mme ZINN JUSTIN Nicole | Algèbre |

Je tiens à remercier,

Monsieur Michel MERIAUX qui a bien voulu me faire l'honneur de présider le jury;

Monsieur Jean Paul DELAHAYE d'avoir accepté d'être rapporteur de cette thèse, et pour les remarques très encourageantes qu'il y a fait;

Monsieur Christian PERCEBOIS, non seulement pour sa participation au jury en tant que rapporteur, mais aussi pour ses remarques et critiques qui ont permis, d'ores et déjà, d'améliorer la rédaction du document;

Monsieur Bernard TOURSEL qui m'a pleinement accueilli dans son équipe, et d'être l'initiateur de ce travail dont il a suivi la progression;

Monsieur Gilles GONCALVES qui a m'a suivi tout au long de ce travail, pour son aide précieux et sa disposition à tout moment;

Monsieur Pierre LECOUFFE et Phillipe DEVIENNE d'avoir accepté d'examiner cette thèse.

Je tiens également à remercier toute l'équipe de recherche, et plus particulièrement les collègues du bureau 310, pour la parfaite ambiance de travail.

*A mes parents,*

## -=- liste des tables -=-

# INTRODUCTION

Longtemps considéré comme une technologie futuriste, le parallélisme sort de l'ombre (de ces 20 dernières années) pour entrer dans sa phase commerciale. En effet, la répétition de processeurs, ou d'organes de traitement seulement, était jusque là difficile à réaliser et coûtait très cher. Aujourd'hui la technologie est prête, et l'on voit au rendez-vous : Alliant, Convex, BBN, Thinking Machine, Intel, Sequent, Ncube, ... . Les progrès technologiques actuels autorisent la réalisation de super-ordinateurs à des coûts raisonnables. La notion même de multiprocesseur est considérée comme faisant partie des choix d'un système industriel.

Cette évolution n'est pas totalement achevée du fait de l'écart existant entre les progrès effectués dans le domaine du matériel, et ceux réalisés dans le domaine du logiciel des super-ordinateurs. Ce retard du logiciel s'explique d'une part, par la difficulté de programmation des machines parallèles et par les problèmes de mise au point qu'entraîne le parallélisme, et d'autre part, par l'inadaptation des langages classiques.

Les langages de programmation traditionnels ou procéduraux, comme Fortran, Pascal, C, ADA,..., basés sur le concept d'assignation, sont dédiés au modèle de calcul imposé par l'architecture de von Neumann. Dans ces langages, un programme est constitué par une suite d'instructions destinées à commander la machine et lui faire changer d'état, le contrôle de l'exécution, spécifié par l'ordre d'exécution des instructions, restant à la charge du programmeur. Aussi ce dernier est-il contraint de se concentrer sur des détails de mise en oeuvre d'algorithmes tels que la gestion de la mémoire et l'ordre d'exécution, plutôt que sur ce qui doit être fait, c'est à dire sur l'algorithme lui même. La dépendance des langages procéduraux vis à vis de la machine conventionnelle rend difficile leur adaptation à des architectures parallèles.

Des transformations ou extensions visant à "paralléliser" les langages procéduraux préexistants ont été utilisées et ont donné naissance à de nouveaux langages de programmation comme OCCAM, C-parallèle, ..., etc. Dans ces langages, le programmeur a la charge de partitionner l'algorithme et d'élaborer les protocoles de communication et de synchronisation entre ces tâches. Les problèmes des langages parallèles sont, d'une part, cette difficulté de programmer avec de tels langages, qui les rend souvent hantés par l'esprit von Neumann, et d'autre part, l'inexistence d'un modèle parallèle reconnu universel, sans lequel il n'y a pas d'algorithmique possible.

Il faut donc changer de modèle d'exécution, ce que propose la programmation déclarative. Les langages déclaratifs, qu'ils soient fonctionnels ou logiques, sont issus de modèles théoriques (mathématiques). Ces modèles sont caractérisés par l'indépendance des résultats vis à vis de l'historique du calcul, et, de ce fait, ils présentent un parallélisme intrinsèque qui peut faire l'objet d'une extraction automatique.

Il semble ainsi intéressant de marier les architectures parallèles, qui fournissent l'efficacité, et la programmation déclarative, qui permet une certaine aisance de programmation et une maîtrise des problèmes complexes.

La programmation logique a connu ces dernières années une multitude de recherches visant à exécuter les programmes logiques de façon parallèle. La principale explication de cette envolée de recherches réside dans le fait que la programmation logique offre, d'une part, un pouvoir d'expression élevé, et d'autre part, un haut degré de parallélisme implicite qui peut être exploité dans des implantations sur des architectures parallèles. Face à l'émergence des multiprocesseurs et l'urgente nécessité de langages adaptés à de telles machines, nous placerons cette thèse dans le cadre de l'évolution des systèmes logiques parallèles. L'objectif de ce travail est de démontrer l'efficacité du parallélisme implicite de la programmation logique et particulièrement celle du parallélisme implicite du langage Prolog, en tenant compte de toutes ses propriétés non logiques et en particulier des effets de bords. Cette efficacité n'est possible que lorsque les problèmes exposés ci-après sont résolus, dans la mesure où le parallélisme n'est ni explicité ni pensé au niveau de l'application.

**Extraction automatique du parallélisme :**

De manière générale, l'extraction automatique du parallélisme consiste à déceler, dans les applications exprimées dans un langage séquentiel classique, le parallélisme existant à un niveau syntaxique, c'est à dire les actions qui peuvent s'exécuter concurremment. La parallélisation se fait de façon statique, au moment de la compilation, par analyse de la dépendance et la restructuration du code ou des données. L'analyse repose donc essentiellement sur une réflexion sur le code et les données. De nombreux travaux sur la parallélisation/ vectorisation ont exploré cette approche pour les langages impératifs et en particulier pour Fortran. Mais la difficulté, dans l'utilisation d'une telle approche en programmation logique, est l'absence de frontière entre le code et les données dans un programme logique. Il faut donc trouver un modèle d'exécution dont l'essence est parallèle.

**Contrôle de la granularité :**

Une exécution parallèle est un ensemble de tâches généralement communicantes, qui collaborent pour réaliser l'application. La granularité caractérise la quantité moyenne de travail réalisée dans une tâche sans que celle-ci ait recours à la communication avec d'autres tâches. Plus la granularité est grosse, plus le taux de communication, qui peut être un facteur pénalisant, est faible, et, en conséquence, l'efficacité du parallélisme peut être assurée. Mais le choix de la granularité dépend de l'architecture cible. Dans une architecture massivement parallèle, il est préférable d'avoir une fine granularité car lorsque la granularité diminue, la sémantique de l'application est déportée de l'intérieur des tâches vers les connexions logiques qui les unissent.

**Gestion des effets de bord :**

Si l'on considère Prolog comme un langage de programmation à part entière, il est impossible d'ignorer les effets de bord qui permettent la réalisation des interfaces utilisateur et la mise à jour de la base de données. Ce problème est délicat dans le cas d'un système logique parallèle, à cause de l'utilisation des prédicats *assert*, *retract* et *cut* qui permettent de modifier dynamiquement le programme et le contrôle de l'exécution.

**Stratégie de répartition du travail :**

Il s'agit de projeter la machine virtuelle (le modèle d'exécution) sur une machine physique. La machine virtuelle crée des tâches qui doivent être exécutées par la machine physique en employant au maximum toutes ses ressources. Ce problème n'est, en général, guère facile à résoudre, à cause du caractère dynamique de la projection.

Ces problèmes sont similaires à ceux existant dans d'autres domaines de programmation parallèle. A travers cet exposé, nous montrerons les particularités sous-jacentes à l'implantation de Prolog sur des architectures multiprocesseurs, et les solutions que nous y avons apportées.

## Plan de thèse

### Chapitre I

En première partie, ce chapitre présente les principes de base de Prolog ainsi que les principales techniques d'implantation séquentielle. Dans la seconde partie de ce chapitre, nous examinons les possibilités de parallélisme offertes par la programmation logique.

### Chapitre II

Ce chapitre fait le point des recherches menées dans le domaine de la programmation logique parallèle. Les deux principaux axes de ces recherches, langages gardés et systèmes non-déterministes, y sont présentés. Enfin, une critique de l'existant nous permet de situer nos travaux par rapport à des travaux réalisés ou en cours de réalisation.

### Chapitre III

Nous commençons ici par donner les objectifs de l'étude d'un modèle d'exécution OU-parallèle de programmes logiques. Nous présentons ensuite les principaux traits caractéristiques de ce modèle, à savoir, la technique de liaison utilisée pour gérer les liaisons multiples provoquées par le parallélisme OU, ainsi que le contrôle du parallélisme.

### Chapitre IV

Nous décrivons dans ce chapitre une machine abstraite parallèle pour la mise en oeuvre du modèle proposé, en spécifiant d'une part, les extensions apportées à une machine abstraite séquentielle (la *WAM*), et d'autre part, la gestion des processeurs virtuels et la répartition des tâches entre eux.

### Chapitre V

Dans ce chapitre, nous soulevons les problèmes posés par les effets de bord dans une exécution OU-parallèle. Nous montrons ensuite, comment des prédicats à effet de bord, et plus particulièrement les prédicats : *read*, *write*, *cut*, *assert*, *retract*, peuvent être pris en compte dans notre modèle de calcul.

### Chapitre VI

Dans ce chapitre, nous donnons une description de la simulation de cette machine abstraite parallèle sur station de travail Sun. Nous présentons ensuite des programmes de test ayant permis d'effectuer des mesures de performance, et donnons, enfin, les résultats d'exécution de ces programmes, sur cette machine abstraite parallèle.

## Travaux de l'auteur :

L'auteur a contribué, dans un premier temps, à une étude visant à l'implantation d'un modèle de calcul à fin grain de parallélisme [ Han ] pour l'exécution des programmes logiques sur des machines massivement parallèles (plusieurs centaines de processeurs). Les problèmes traités dans cette étude [Bou 90] sont la gestion des environnements et la représentation des termes, dans l'hypothèse qu'un processeur ne dispose que d'une mémoire privée et ne communique avec les autres processeurs que par le biais d'un réseau d'interconnexion.

Conjointement, aux travaux liés à la gestion des environnements, un simulateur a été réalisé dans le cadre de la thèse de I. Hannequin [Han 91], et qui a permis d'évaluer le parallélisme développé par le modèle de calcul. Les résultats de simulation sur les programmes testés ont montré un niveau de parallélisme moyen (une douzaine de processus), et ce malgré l'utilisation d'un modèle à fin grain de parallélisme. Les problèmes de synchronisations et de communications, ainsi que les contraintes imposées par le respect de la sémantique opérationnelle de Prolog, sont à l'origine du degré du parallélisme observé. Ces résultats nous ont amenés à étudier une seconde approche exploitant un grain de parallélisme plus gros, et où le respect de la sémantique opérationnelle du langage Prolog reste parmis les principaux objectifs à atteindre.

Les travaux liés à cette thèse, s'inscrivent dans le cadre de cette nouvelle orientation. Ils ont débuté par une étude de l'état de l'art dans le domaine de la programmation logique parallèle. Cette étude a abouti à l'élaboration du modèle de calcul présenté ici. Afin de valider ce modèle et d'effectuer des mesures de performances, une machine abstraite parallèle à été spécifiée et implantée sur une machine séquentielle (station de travail Sun).

# -=- Chapitre I -=-

*PROLOG :*

*Résolution et contrôle,*

*implantation,*

*parallélisme.*

Adapté à la manipulation symbolique, le langage Prolog est considéré comme un langage de programmation de très haut niveau. Basé sur un modèle abstrait dérivé de la logique mathématique, il permet de décrire et de résoudre des problèmes complexes.

La simplicité et la puissance de Prolog, ainsi que l'existence d'une sémantique bien définie à partir du calcul des prédicats, expliquent son succès dans divers domaines d'application, qui vont de la compilation et du traitement de la langue naturelle, pour lequel il a été conçu [Col 72], à la définition de systèmes experts et de bases de données relationnelles.

# 1 - Résolution et contrôle Prolog :

Dans ce paragraphe, nous ne présenterons pas formellement les bases théoriques de la programmation logique, à savoir le principe de résolution et le calcul des prédicats qui sont décrits en détail dans [Rob 65] et [Del 86], mais nous rappellerons comment Prolog effectue une résolution sur un système de clauses de Horn.

## 1.1 - Objets manipulés dans Prolog :

Dans les langages de programmation classiques, une valeur est attachée à chaque variable. L'exécution d'un programme consiste à effectuer toutes sortes d'opérations sur ces valeurs afin d'obtenir des valeurs finales pour les différentes variables. En Prolog, une variable représente une valeur inconnue et le but du déroulement d'un programme consiste non pas à modifier cette valeur, mais à la déterminer. Une variable se comporte donc comme une inconnue dans une équation mathématique, mais la différence principale est qu'une variable représente quelque chose de plus complexe que l'on appelle un *terme*.

Un terme peut être :

| | |
|---|---|
| **Un symbole** | suite de caractères (généralement minuscules) ne débutant pas par un chiffre |
| **Un nombre** | nombre entier ou flottant |
| **Une variable** | suite de caractères commençant généralement par une lettre majuscule |
| **Un terme fonctionnel** | de la forme f(t1,...,tn) où f est un symbole de fonction d'arité n, et où t1,..., et tn sont des termes |
| **Une liste** | regroupement d'objets Prolog. Une liste est de la forme [A\|B] où A et B sont des objets Prolog |

## 1.2 - Unification :

L'unification constitue le coeur de tout interprète Prolog. Elle consiste à déterminer, lorsqu'elle existe, l'instance commune de deux termes. Plusieurs algorithmes ont été proposés pour effectuer une telle opération, et sont développés dans [Rob 65] et [Cha 73]. Nous ne reviendrons pas sur ces algorithmes, mais expliquerons uniquement le principe de l'unification.

**Définition 1 :**

Une substitution $\sigma$ est un ensemble fini de couples (variable, terme) tel que deux couples quelconques n'ont pas la même variable et la variable et le terme d'un couple sont différents. On dira qu'une variable apparaissant dans la partie gauche d'un couple est **liée** ou **instanciée** au terme de ce couple. Une variable apparaissant dans une substitution, mais jamais dans la partie gauche d'un couple, est dite **libre**.

Exemple : Soit $\sigma = \{ ( X , a ) ; ( Y , f ( X , Z ) ) \}$
Les variables X et Y sont liées respectivement aux termes a et f ( X , Z ).
Par contre, la variable Z est libre.

**Définition 2 :**

Appliquer une substitution à un terme t consiste à remplacer toutes les variables apparaissant dans t par les termes associés suivant s.

Exemple : Soit $t = h ( X , Y )$ et $\sigma = \{ ( X , a ) ; ( Y , f ( X , Z ) ) \}$
l'application de la substitution $\sigma$ à t donne comme résultat : $\sigma(t) = h ( a , f ( a , Z ) )$.

**Définition 3 :**

Deux termes t1 et t2 sont **unifiables** s'il existe une substitution $\sigma$ telle que : $\sigma(t1) = \sigma(t2)$.
On dira que $\sigma$ est un **unificateur** de t1 et t2.

Exemple : Soit $t1 = f ( X , a )$ et $t2 = f ( g(Y) , Y )$.
$\sigma = \{ (X , g(a) ) ; ( Y , a ) \}$ est un unificateur de t1 et t2.

**Définition 4 :**

L'existence d'un unificateur n'est pas unique. Un unificateur est dit **le plus général unificateur** de deux termes t1 et t2, si et seulement si, pour tout unificateur $\theta$ de ces deux termes, il existe une substitution $\lambda$ telle que $\theta = \lambda \circ \sigma$, o étant la composition de deux fonctions.

Exemple : Soit t1 = f(X, a) et t2 = f(Y, Z).
L'unificateur le plus général est $\sigma = \{ (X,Y) ; (Z,a) \}$
Un deuxième unificateur de t1 et t2 est $\theta = \{(X, Y) ; (Y, a) ; (Z, a)\}$,
mais n'est pas le plus général car la substitution $\lambda = \{(Y,a)\}$ donne $\theta = \lambda \circ \sigma$.

L'unification est donc une opération qui permet de trouver l'unificateur le plus général de deux termes s'ils sont unifiables. Le premier algorithme permettant son calcul a été proposé par J.A. Robinson [Robinson 1965].

## 1.3 - Clauses de Horn :

Un programme logique est une conjonction de clauses. Une clause C du calcul des prédicats est une formule de la forme :

B1 ou B2 . . . . ou Bm <- A1 et A2 . . . An (0 <= m , n)

ou, de manière équivalente, et sous-forme normale disjonctive :

B1 ou B2 . . . . ou Bm ou (non A1) ou (non A2) . . . (non An).

N.B. : les Ai et Bi sont des littéraux ou des négations de littéraux.

Un littéral est une formule atomique de la forme p(t1, . . .,tk) où :
- p est un symbole de prédicat,
- k représente l'arité de p,
- t1. . .tk (arguments) sont des termes (définis dans le paragraphe précédent).

Une clause C est dite clause de Horn si et seulement si m <= 1, i.e C possède au plus un littéral dans sa partie gauche.

De ce fait, on distingue 4 types de clauses de Horn, décrites en page suivante.

**Type règle :**

| B <- A1 et A2 . . . et An (m = 1 et n > 0) |
|---|

La sémantique déclarative de cette clause est :

pour toutes les variables X1, . . ,Xk ayant une occurrence dans A1,A2,. . .,An et B, si (A1 et A2 . . . et An) est vrai alors B est vrai.

On dit que le littéral B forme la tête de la clause tandis que la partie droite forme le corps de la clause.

**Type fait :**

| B <- (ou B) (m = 1 et n = 0) |
|---|

La sémantique déclarative de cette clause est :

pour toutes les variables X1, . . ,Xk apparaissant dans B, B est vrai.

**Type but (ou type démenti) :**

| <- A1 et A2 . . . et An (m = 0 et n > 0) |
|---|

La sémantique déclarative de cette clause est :

pour toutes les variables X1, . . ,Xk ayant une occurrence dans A1,A2,. . .An, (A1 et A2 . . . et An) est faux.

**Type clause vide :**

| ❐ (m = 0 et n = 0) |
|---|

toujours interprétée comme faux.

## 1.4 - Base de connaissances :

La base de connaissances d'un système Prolog est une conjonction de clauses de Horn de type règle ou fait. Les clauses définies et ayant le même symbole de prédicat en partie gauche, peuvent être regroupées par paquets de clauses. Lorsque la sémantique procédurale est considérée, on dit qu'un paquet de clause forme une procédure par analogie aux langages procéduraux.

```
connect (a, b).
connect (a, c).
connect (c, e).
connect (d, e).

chemin (X, Y) :- connect (X, Y).
chemin (X,Y)  :- connect (X,Z), chemin (Z, Y).
```

figure 1.1 : Exemple de base de connaissances

La "procédure" *connect* définie par 4 faits, décrit un graphe orienté à 5 noeuds.
La "procédure" *chemin* définie par 2 clauses indique si deux noeuds sont liés par un chemin.

## 1.5 - Résolution Prolog :

L'exécution d'un programme Prolog consiste à poser des questions dans le cadre du système défini par la base de connaissances. Une question est considérée comme une clause de type but de la forme :

```
                <- Q1 Q2 . . . Qn
ou encore    Q1 Q2  Qn?
```

Une telle question, où figurent les variables X1, . . ,Xk, signifie :

"existe-t-il des valeurs de X1, . . .,Xm telles que (Q1 et Q2 . . .et Qn) est vrai, compte tenu de la théorie exprimée par la base de connaissances ?".

La résolution sur laquelle est basée Prolog (SLD-résolution) est une procédure qui permet de démontrer qu'une conjonction de littéraux (question) est une conséquence de l'ensemble des clauses définissant la base de connaissances. Pour cela, Prolog effectue une réfutation de la négation de la question, i.e qu'il montre que le système constitué par la base de connaissances et la négation de la question est inconsistant (ou contradictoire).

**Principe de la résolution :**

On considère la question comme un résolvant initial.
Soit le résolvant <- A1, . . ., Ak , . . . Am où m >= k >= 1
Soit Ak le littéral sélectionné,
--> pour résoudre Ak, il faut choisir une clause H <- B1, . . ,Bn où n 0, telle que Ak et H aient pour unifiant général, θ.
Le nouveau résolvant est obtenu en appliquant la substitution θ, après avoir remplacé Ak par le corps de la clause sélectionnée : on appelle cette opération une *dérivation*. Le nouveau résolvant est : <- q(A1, . . ., Ak-1, B1, . . ., Bn , Ak+1, . . ., Am).

Prolog est basé sur un cas particulier de cette résolution qui consiste à utiliser une règle de sélection permettant de choisir le littéral le plus à gauche du résolvant. L'organisation de la base de connaissances énoncée précédemment et cette règle de sélection donnent à Prolog une stratégie de résolution bien définie, appelée en **profondeur d'abord** (*depth first* en anglais). Comme nous le verrons dans la section suivante, cette stratégie se prête bien à une implantation par piles.

Un programme Prolog est donc constitué par un but et un ensemble de clauses. La résolution particulière de Prolog fournit au programme une sémantique opérationnelle bien définie.

D'un point de vue procédural, on peut considérer que l'unification représente une généralisation de passage de paramètres ("passage de paramètres dans les deux sens : fomels<>effectifs"), et qu'une étape du processus de résolution peut être assimilée à un appel de procédure des langages classiques. Mais cette dernière analogie ne peut se faire facilement à cause du non-déterminisme, i.e l'existence des clauses alternatives. En effet, lorsqu'une procédure (paquet de clauses) est invoquée, la première clause est sélectionnée et l'unification est tentée. Les autres clauses, dites alternatives, sont provisoirement ignorées : il y alors création d'un point de choix. Lorsqu'une unification échoue, l'exécution courante est abandonnée et le contrôle est rendu à la plus récente des alternatives rencontrées (rappel). Ce mécanisme de contrôle est appelé **retour-arrière** (*backtracking*).

Le non-déterminisme géré par le retour-arrière et l'unification sont les caractéristiques principales de Prolog, qui font de ce langage un langage puissant et attrayant pour la résolution des problèmes.

## 1.6 - Représentation arborescente :

Le contrôle de Prolog procède par essais successifs afin de déterminer l'ensemble des solutions d'un problème donné. Ce processus peut être représenté par le parcours d'un arbre appelé arbre de recherche. Il est possible de distinguer deux types d'arbres de recherche : les arbres Et-Ou et les arbres SLD.

### 1.6.1 - Arbre Et-Ou :

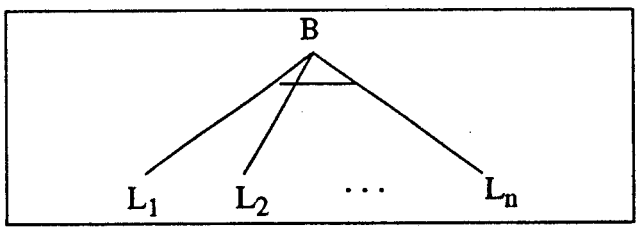
Les arbres Et-Ou utilisés en intelligence artificielle permettent de représenter une catégorie d'algorithmes dits à essais successifs : "Pour traiter un problème P, il suffit de traiter un des problèmes équivalents P1 **ou** P2 **ou** . . .Pn ("étape Ou"), et pour traiter un problème Pi, il suffit de décomposer le problème Pi en sous-problèmes plus simples Pi1 **et** Pi2 **et** . . .Pim ("étape Et"), le même procédé est appliqué au sous-problèmes Pij.".

On distingue deux types de noeuds formant ces arbres; les premiers, correspondant à une étape Ou, sont appelés **noeuds Ou**, les autres sont appelés **noeuds Et**.

**Un noeud ET dans Prolog signifie :**

" Pour prouver un but B à l'aide d'une clause C ( H :- $L_1, L_2, \ldots, L_n$ ), il faut d'abord unifier les arguments du but B et ceux du littéral H, et ensuite prouver successivement les sous-buts $L_i$ figurant dans le corps de la clause C".

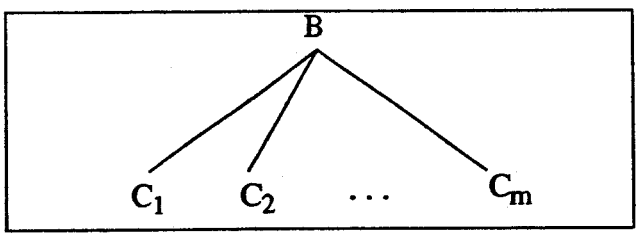
Un tel noeud est représenté :




**Un noeud Ou dans Prolog signifie :**

" Pour résoudre un but B en utilisant les alternatives C1, C2, . . .,Cm, il suffit de démontrer le corps d' une clause Ci (n >= i >= 1), après avoir effectué les unifications des arguments but B avec ceux des têtes de clauses".

Un tel noeud est représenté :



**Exemple** : Considérons l'exemple de la figure 1.
Traçons l'arbre *Et-Ou* associé à la question *chemin(a, e)? :*

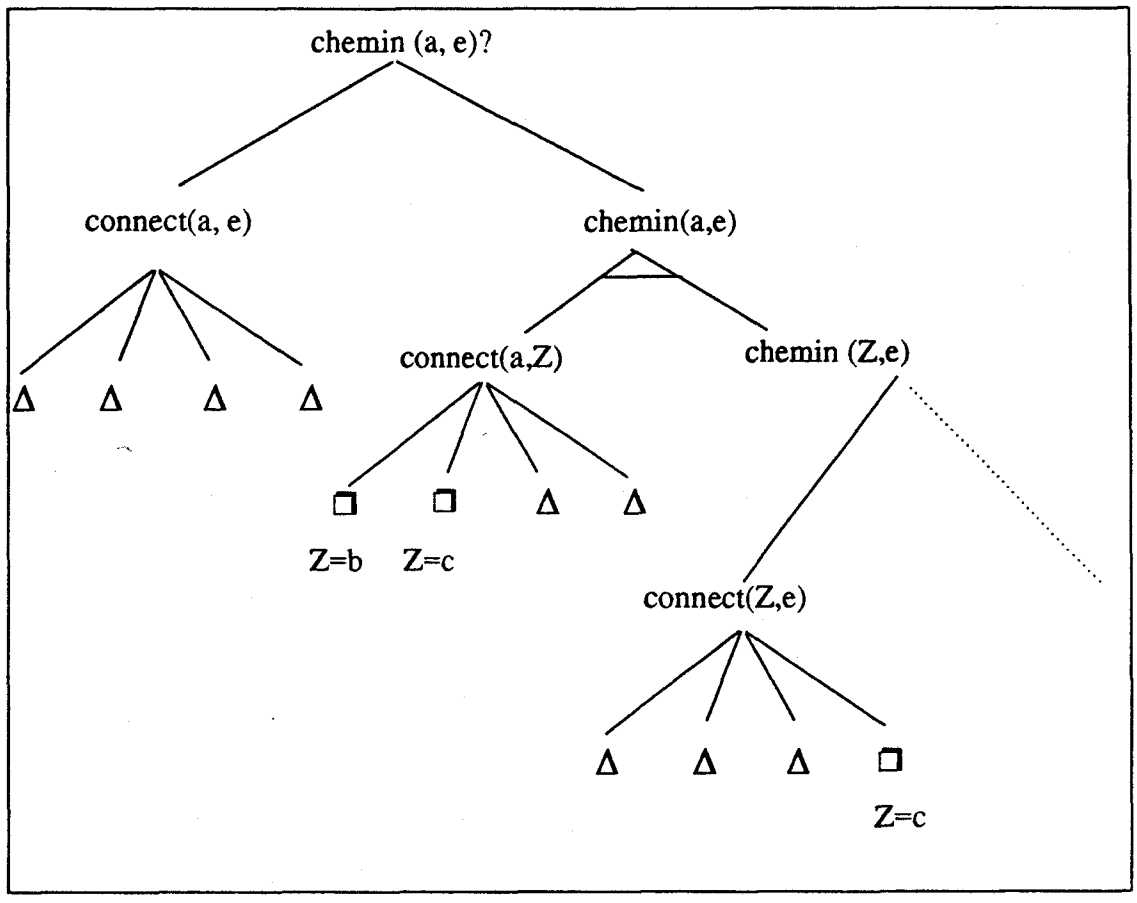



figure 1.2 : Arbre *Et-Ou*

Il existe deux alternatives pouvant résoudre le but *chemin(a, e)?*, d'où le premier noeud Ou.

Après unification, la première alternative consiste à prouver le but *connect(a, e)*, les quatre faits associés au prédicat *connect* donnant un échec (symbolisé par Δ).

L'exécution de la deuxième alternative consiste à démontrer les sous-buts connect(a, Z) et chemin(Z, e), d'où le noeud Et.

Il existe deux faits donnant un succès (d'où les deux clause vides ❐ ) au premier sous-but, mais seul le résultat obtenu par le premier fait (Z = c) peut satisfaire le deuxième sous-but.

### 1.6.2 - Arbre *SLD* :

La notion d'arbre *SLD* ou arbre de dérivation *SLD* est fondée sur le principe de résolution de Robinson.

Dans le cas d'un interpréteur Prolog, la preuve d'une liste de buts Lb = B1, . . ., Bn, peut être décrite par un arbre défini de la manière suivante [Del 87] :

**(1)** la racine de l'arbre est la liste Lb;

**(2)** chaque noeud est une liste d'atomes;

**(3)** si le noeud L1 est un fils d'un noeud L2 alors L1 est obtenu en dérivant l'atome le plus à gauche dans L2 par une règle (clause) du programme,
(l'arc ainsi créé peut être étiqueté par la substitution obtenue par unification);

**(4)** pour toute règle permettant de dériver un atome d' un noeud L, ce dernier a un fils correspondant;

**(5)** les noeuds fils d'un noeud L sont ordonnés en fonction de l'ordre d'apparition dans le programme, des clauses permettant de dériver l'atome le plus à gauche dans L;

**(6)** un noeud qui ne contient aucun atome, est représenté par □.

Un noeud feuille de l'arbre est soit une liste d'atomes, soit la liste vide.

Dans le premier cas, l'atome le plus à gauche n'est pas dérivable et conduit ainsi à l'échec.

Dans le second cas, un succés a été obtenu : la question posée a au moins une réponse, cette dernière étant constituée de la liste des liaisons des variables apparaissant dans la question.

**Exemple :**

Arbre SLD associé à la question chemin(a,e)? dans la base de connaissances de la figure 1.1 :

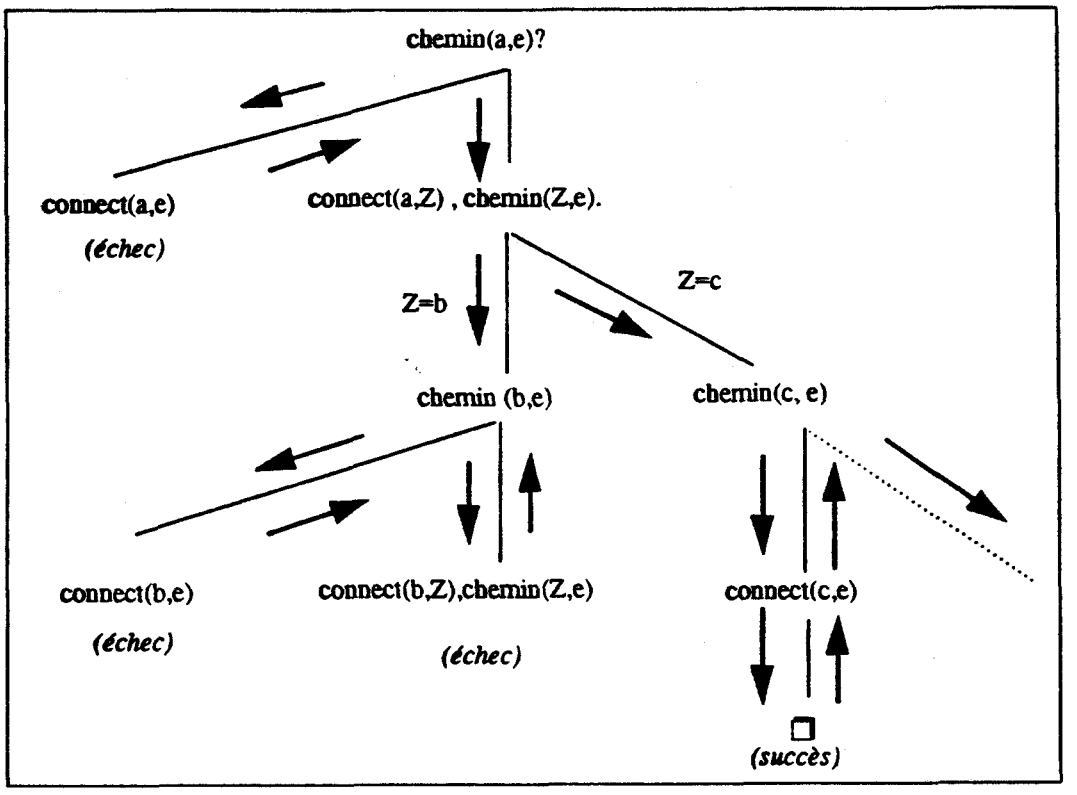



figure 1.3 : Arbre *SLD*

Le parcours indiqué ci-dessus correspond à la recherche de la première solution.

Prolog ne s'arrête pas à la rencontre de cette solution, même si la question est "existe-t-il un chemin liant a et e ?".

En effet, compte-tenu de la stratégie de résolution de Prolog, le parcours des branches s'effectue en profondeur d'abord, avec retour-arrière systématique sur le dernier point de choix.

L'arbre de recherche, dans cet exemple, est infini.

Dans le paragraphe suivant, nous montrerons comment Prolog peut contrôler l'explosion de l'arbre de recherche.

Dans la suite de cet exposé, nous emploierons le nom *arbre de recherche* au lieu de *arbre SLD*, et, lorsque l'arbre *Et-Ou* sera considéré, nous le préciserons.

## 1.7 - Contrôle :

### 1.7.1 - Contrôle explicite :

Comme nous l'avons vu, le contrôle Prolog peut être mis en oeuvre par deux opérations qui sont l'invocation des prédicats (procédures) et le retour-arrière. La première opération est statique, tandis que la deuxième est complètement dynamique.

Il est cependant possible, dans certains cas, de savoir à quelle condition la clause qui conduit à la solution est déterminée.

Si, alors, des clauses alternatives restent à essayer, elles sont en général sans utilité et peuvent conduire à des calculs redondants lorsque le retour-arrière est effectué, comme le montre l'exemple de la figure 1.3.

Il a donc été nécessaire, dès les premières implantations de Prolog, de fournir au programmeur un moyen d'écarter les alternatives inutiles. Ce moyen est l'utilisation du prédicat coupe-choix (cut), noté "!" ou "/". Son action consiste, lorsqu'il est invoqué, à éliminer toutes les alternatives rencontrées depuis l'invocation de la procédure parente. La sémantique du coupe-choix ne peut être définie qu'en terme de contrôle, ce qui fait perdre la nature déclarative du programme.

D'autres approches ont été proposées pour améliorer la stratégie classique (*en profondeur d'abord*) de contrôle en Prolog. Parmi ces approches, l'on trouve celles basées sur la suspension de la résolution d'un sous-but tant qu'il ne satisfait pas un certain nombre de conditions sur l'unification de ses arguments. La résolution est reprise une fois qu'une unification ultérieure satisfait ces conditions.

Plusieurs implantations de Prolog proposent de telles possibilités : les déclarations *wait* de MU-prolog, subordonnant l'exécution d'un but, à des conditions sur l'unification de ses arguments, et le prédicat *geler* de Prolog-II qui permet de retarder l'évaluation d'un but tant qu'une variable demeure libre.

### 1.7.2 - Contrôle implicite (retour-arrière intelligent) :

Le retour-arrière classique s'effectue systématiquement au dernier point de choix. Cette façon de procéder s'avère dans certains cas inefficace par sa remise en cause "aveugle" des choix effectués jusqu'alors.

Considérons l'exemple suivant :

```
p(0). p(1). ... p(1000).
q(0,1). q(1,2). q(2,3).
:-p(X), q(X,X).
```

Lors de l'échec du sous-but q(X,X), le retour-arrière sur le sous-but p(X) est inutile.

Le but du retour-arrière intelligent est d'éviter de parcourir inutilement des sous-arbres ne pouvant produire que des échecs. Pour cela, il se doit d'analyser les causes d'échec d'une unification afin de déterminer le responsable de la, ou des liaisons, ayant provoqué l'échec.

Les propositions faites dans ce domaine n'ont pas encore influencé les implantations actuelles. En effet, les différents algorithmes proposés jusque là restent relativement complexes et coûteux.

## 2 - Implantation :

L'idée d'utiliser la logique des prédicats comme langage de programmation est née, au début des années 70, grâce aux travaux de A. Clomerauer et R. Kowalski. Le premier interprète prolog a été développé en 1971 à l'Université d'Aix-Marseille par P. Roussel. Très vite, la puissance et la simplicité du langage ont donné naissance, dans le courant des années 70, à de nombreuses recherches visant à trouver des techniques efficaces d'implantation.

Dans les premières mises en oeuvre de Prolog, les deux problèmes habituels aux langages interprétés se posent : ils concernent d'une part le temps d'exécution et d'autre part la consommation d'espace mémoire.

En 1977, Warren [War 80] a montré par la réalisation d'un premier compilateur (Prolog-Dec10), qu'il était possible de compiler Prolog et d'obtenir des performances cinq à dix fois meilleures qu'un simple interpréteur. Actuellement, l'immense majorité des systèmes Prolog commercialisés intègrent un compilateur.

La conception de la plupart des compilateurs Prolog est basée sur la notion de machine abstraite. Cette technique est fréquemment employée en matière de compilation, en vue d'assurer la portabilité. Le compilateur génère un code intermédiaire (programme objet) exécutable par la machine abstraite.

Parmi les machines abstraites proposées, nous avons choisi le modèle défini par Warren [War 83], appelé *Machine Abstraite de Warren* (*WAM*), pour décrire la mise en oeuvre d'un interprète Prolog. Ce choix se justifie par le fait que plusieurs systèmes commerciaux Prolog utilisent la WAM, qui est reconnue comme l'une des techniques les plus efficaces d'implantation de Prolog. Nous avons de plus utilisé cette machine comme base de départ pour la conception d'une machine abstraite parallèle qui est fondée sur la stratégie séquentielle.

## 2.1 - Architecture de la *WAM* :

La stratégie en profondeur de Prolog se prêtant à une implantation par pile, la zone mémoire de la WAM est constituée par trois piles qui sont la pile de restauration, la pile locale et la pile globale. Celles-ci sont organisées selon le schéma suivant :

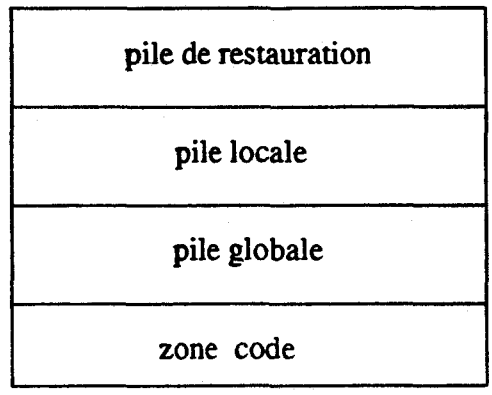



figure 1.4 : Architecture de la *WAM*

La seule contrainte d'organisation réside dans le fait que la pile globale doit figurer en-dessous de la pile locale, ceci afin de simplifier la tâche de l'algorithme de liaisons dans son aspect de chronologie, et de permettre la mise à jour de la pile locale [Boi 88].

D'un point de vue procédural, la pile locale permet de mettre en oeuvre le contrôle Prolog en terme d'appels et de retours de procédures avec passage de paramètres. La gestion par une seule pile, comme dans un langage classique, est insuffisante à cause du retour-arrière. Ce dernier introduit une nouvelle dimension pour une procédure, qui est la notion de succès et de rappel, compte-tenu de la possibilité de définition multiple d'une même procédure. Les autres piles (pile globale et pile de restauration) ont donc été introduites, afin d'étendre le modèle classique appel-retour de procédures et de prendre en compte des particularités de l'unification et du retour-arrière.

## 2.2 - Pile locale :

Le problème de l'utilisation multiple d'une même clause est résolu par génération d'un nouvel exemplaire de clause à chaque utilisation (exemplarisation). Un exemplaire contient un environnement destiné à recevoir les valeurs des variables figurant dans la clause. Un environnement est représenté physiquement par un vecteur comportant $n$ emplacements mémoire, $n$ étant le nombre de variables figurant dans une clause. L'accès à une variable s'effectue par un simple déplacement dans ce vecteur.

A chaque activation d'une clause, un bloc d'activation est créé sur la pile locale. On distingue deux types de blocs d'activation suivant qu'ils sont ou non associés à des points de choix : les blocs déterministes et les blocs de choix.

### 2.2.1 - Les blocs déterministes :

Un bloc déterministe est créé lors de l'activation de la dernière clause envisageable du paquet. Un bloc déterministe est formé d'une partie contrôle destinée à gérer l'avancée dans la résolution, et d'une partie environnement provenant de l'exemplarisation de la clause.

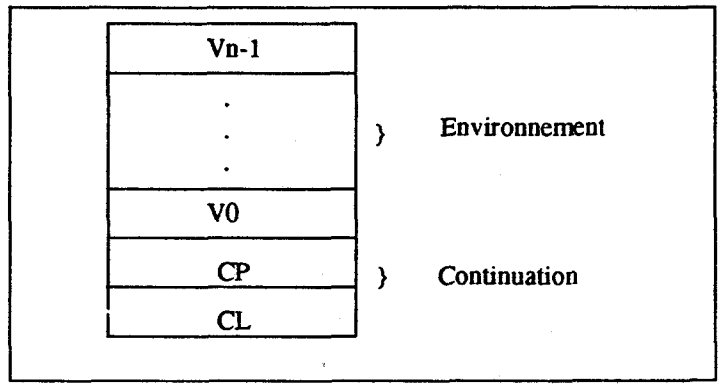



figure 1.5 : Bloc déterministe

La partie continuation qui gère l'avancée comporte deux champs :

**CP** : il permet d'accéder à LB, prochaine suite de buts à prouver, une fois le corps de la clause démontrée;

**CL** : il indique le bloc d'activation portant les informations nécessaires à l'exécution de LB.

### 2.2.2 - Les blocs de choix :

Un bloc de choix est créé sur la pile locale lorsque la clause activée n'est pas la dernière clause du paquet. Un bloc de choix contient une troisième partie, appelée partie reprise, qui permet de mettre en oeuvre le retour arrière. Elle est composée de trois champs:

**BL** : il indique le bloc de choix précédent sur la pile de contrôle.

**BP** : il désigne les choix restants sous la forme de paquet de clauses.

**TR** : (voir pile de restauration).

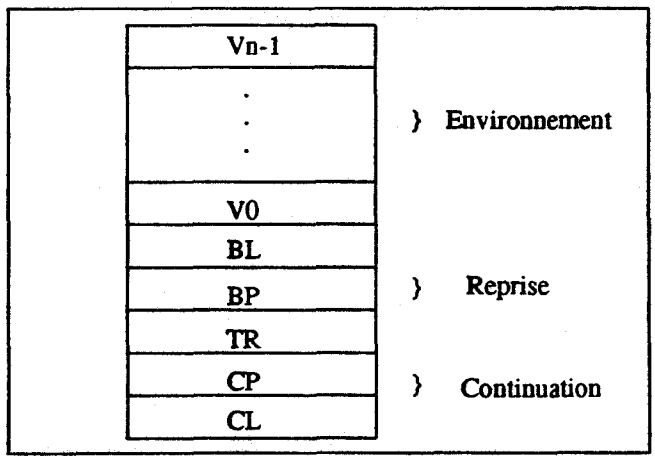



figure 1.6 : Bloc de choix

**Exemple :** Considérons la question p(a)? dans la base suivante:

p(X) :- q(X,Y),r(Y).
q(a,b). q(Z,c).
r(c).

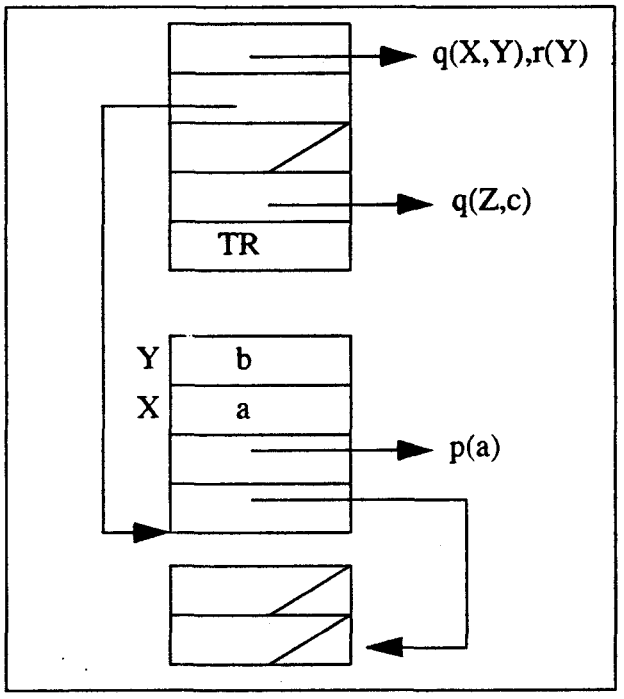



figure 1.7 : Pile locale

La figure 1.7 montre l'état de la pile locale avant l'appel du but r(Y). Trois blocs ont été créés. Le premier est associé à la question. Le second, associé à l'appel de p(a), est d'environnement de taille 2. Le troisième, provenant de l'appel q(X,Y), génère un point de choix avec un environnement vide.

## 2.3 - La pile de restauration :

Cette pile a été introduite afin de mettre en oeuvre le retour-arrière. En effet, lors du retour-arrière sur le dernier point de choix, l'état de la résolvante avant la création de ce point de choix doit être restauré.

Le rôle de cette pile est donc de mémoriser, en cours de démonstration, toutes les modifications effectuées sur les environnements antérieurs au dernier point de choix. Ces modifications concernent toutes les liaisons de variables liées antérieurement au dernier point de choix.

Dans l'exemple précédent, la liaison "Y = b" doit être défaite lors du retour-arrière pour traiter la deuxième clause associée à q.

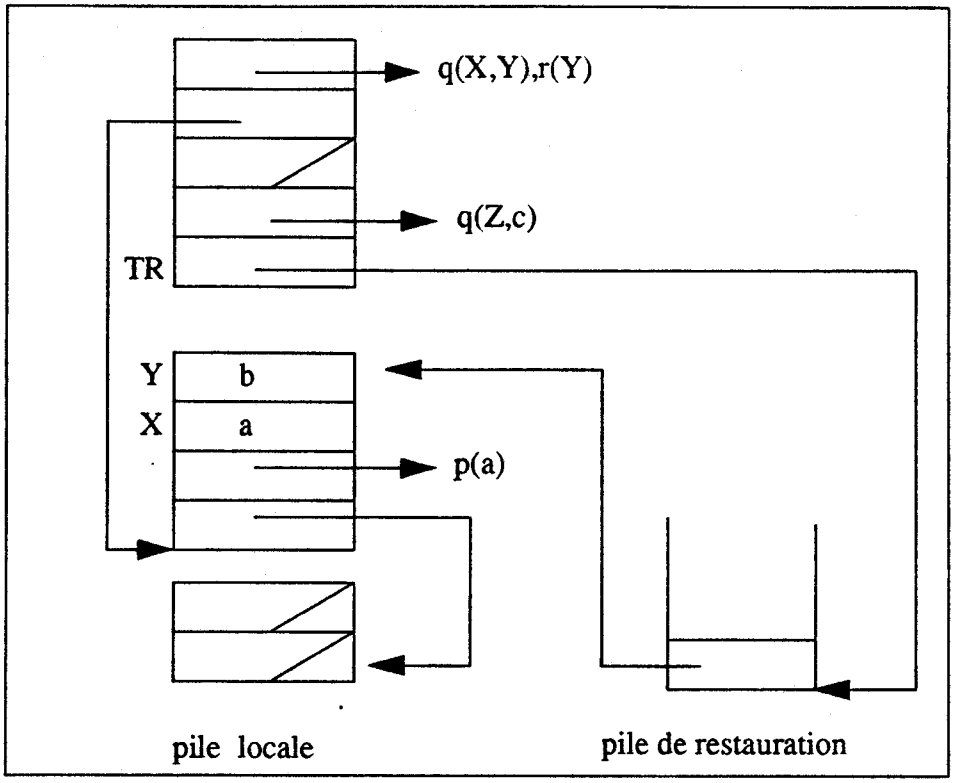



figure 1.8 : Pile de restauration

Le champ TR d'un point de choix indique le sommet courant de la pile de restauration avant la création de ce point de choix. Dans l'exemple ci-dessus, le champ TR indique que la pile de restauration était vide avant la création du point de choix associé à l'appel de q(X,Y).

## 2.4 - La pile de recopie :

Cette pile est utilisée pour stocker les termes complexes tels que les termes structurés et les listes. Elle joue un rôle différent si la technique de **partage de structure** est utilisée pour représenter les termes complexes [Mel 80]. Dans le cas de la *WAM*, la **recopie de structure** est utilisée pour représenter les termes complexes.

Dans un système utilisant le partage de structures, lorsqu'une variable est liée à un terme structuré, ce dernier est représenté par un couple de pointeurs, le premier pointant vers le squelette de la structure qui se trouve dans la zone du code d'une clause, le second vers un environnement d'invocation (il peut en exister plusieurs) de cette clause.

Dans la recopie de structure, l'unification d'une variable à un terme structuré entraîne la création (ou copie) explicite de ce terme, en créant les cellules de variables qui peuvent être liées à leur tour. Une telle liaison est donc constituée d'un seul pointeur.

**Exemple :** Soit la question p(a)? et la base : (1) p(X) :- q(g(X),Y). (3) r(U):-s(U).
(2) q(Z,T) :- r(f(Z)). (4) s(f(g(a))).

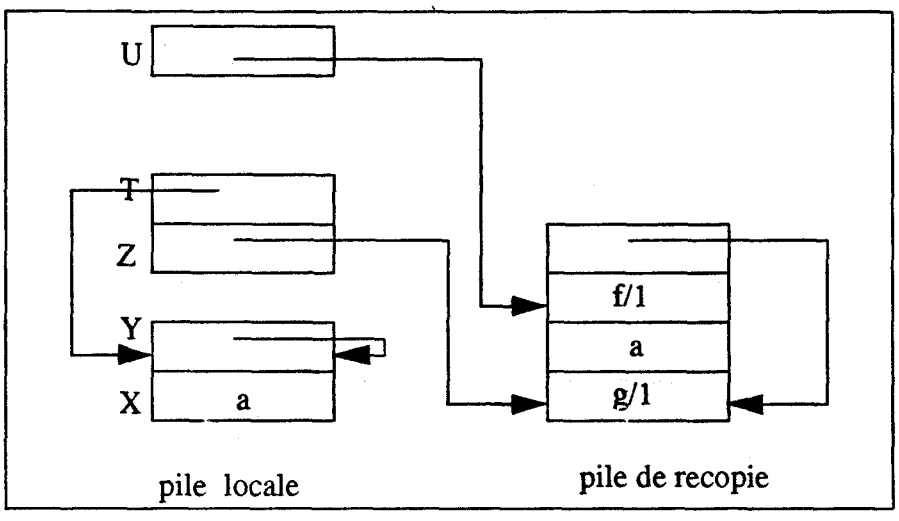



figure 1.9 : Recopie de structures

La figure 1.9 montre l'état de liaisons résultant de la démonstration du but p(a) (seuls les environnements sont indiqués). Lorsqu'une variable est liée à un terme structuré, si ce dernier est accessible pour la première fois, une copie de celui-ci est créée et mémorisée dans la pile de recopie ( mode construction). Lors de la liaison "Z = g(X)", une copie du terme g(X) est créée en sommet de pile de recopie. Mais lors de la liaison de la variable U au terme f(g(a)), ce dernier n'est pas entièrement recopié (mode décomposition) puisque le sous-terme "g(a)" est déjà accessible.

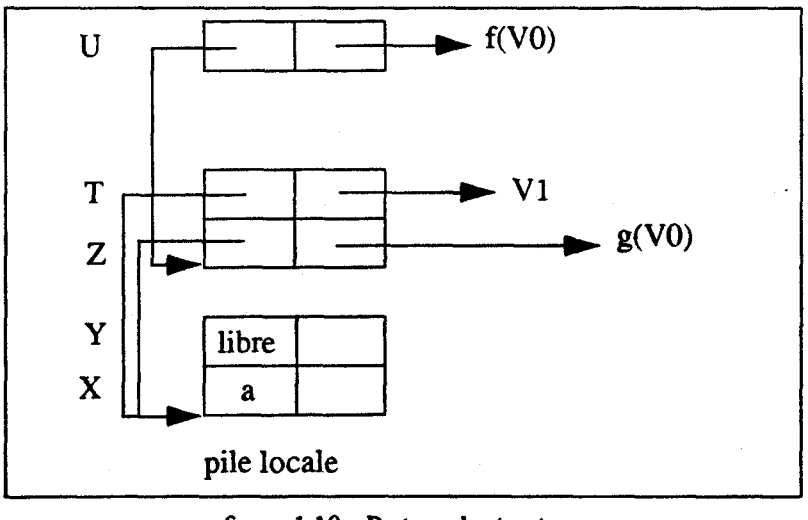



figure 1.10 : Partage de structures

La figure 1.10 montre l'état des liaisons résultant de la démonstration du but *p(a)*. Les variables sont généralement numérotées de 0 à n-1. Dans un environnement, l'ordre des variables est celui de leur apparition dans la clause, soit X = V0 (dans E1), Z = V0 et Y = V1 (dans E2) et T = V1 (dans E3), E1, E2 et E3 étant les environnements d'invocation respectifs des clauses (1), (2) et (3).

La pile globale permet, lorsque le partage de structures est utilisé, de sauvegarder [Boi88] toute variable susceptible de se lier à un terme de chronologie supérieure. Ceci entraîne la création de références du bas vers le haut de la pile locale, interdisant toute mise à jour de celle-ci au retour déterministe. La détection d'une telle variable est statique : toute variable apparaissant dans un terme structuré peut entraîner une référence du bas vers le haut de la pile locale. De ce fait, un environnement est scindé en deux parties: la première est portée par la pile locale, la seconde est allouée dans la pile globale (voir figure 1.11).

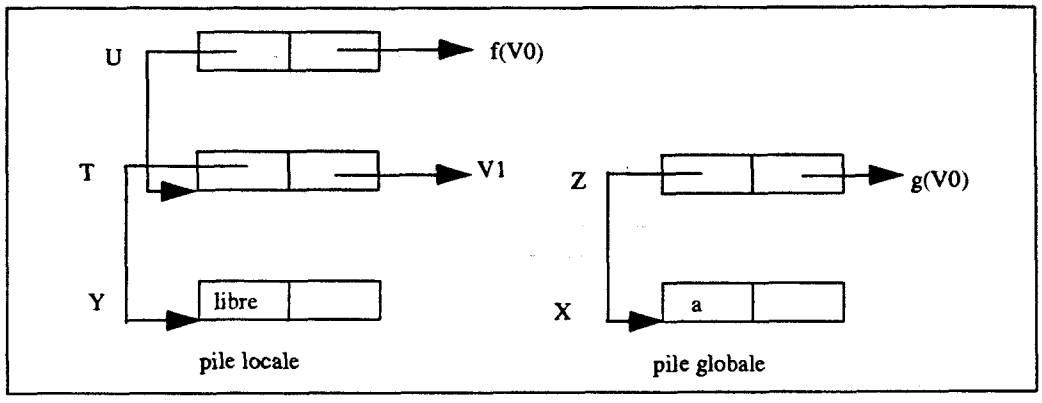



figure 1.11 : Pile globale dans le partage de structures

## 2.5 - Mise à jour des trois piles :

La pile de contrôle est mise à jour au retour des appels déterministes, et lors des retour-arrière, tandis que les deux autres piles sont mises à jour uniquement lors du retour-arrière.

Afin de réaliser ces mises à jours, certains registres de travail ont été introduits :

Registres de sommets de piles : **L** : sommet de pile locale,
**G** : sommet de pile de recopie,
**TR** : sommet de pile de restauration.

Registres de continuation : **CP** : prochaine suite de buts à prouver,
**CL** : bloc local de CP.

Registres de retour-arrière : **BL** : dernier bloc de choix,
**BG** : sommet de pile de recopie associé à B.

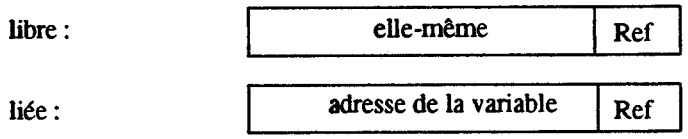
Registres de gestion de l'appel courant : **Ai** : sauvegarde des arguments du but courant.
**PC** : prochain but.

D'autres champs (BCL, BCP) ont été ajoutés pour permettre la transformation des appels terminaux [Boi 88], ces champs étant une copie des champs CL et CP de la partie continuation.

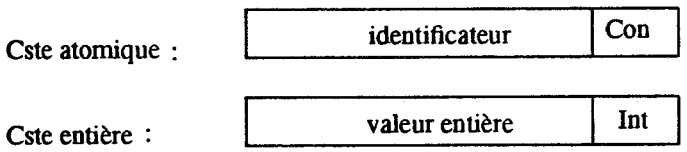
## 2.6 - Représentation des objets :

Le codage des termes s'effectue à l'aide de mots de 32 bits, dont trois, au minimum, servent à désigner le type de l'objet. L'état libre d'une variable est symbolisé par le fait qu'elle est "liée" à elle-même.

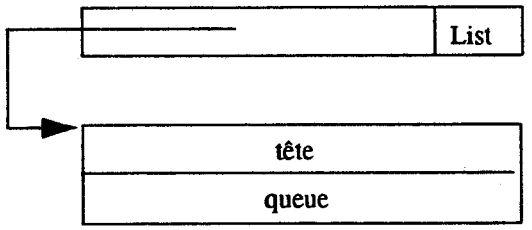
**Variable**



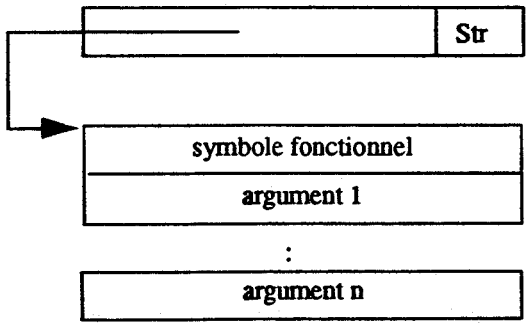

**Constante**




**Liste**




**Terme fonctionnel**



## 2.7 - La zone code :

La zone code contient l'ensemble des instructions représentant les procédures Prolog. On distingue 3 types d'instructions : instructions d'indexation, de contrôle et d'unification.

### 2.7.1 - Instructions d'indexation :

L'indexation des clauses permet de diminuer, à chaque appel d'un but, la taille du paquet de clauses envisageables, évitant ainsi des tentatives inutiles d'unification. Pour cela, à chaque appel d'un but, la nature des arguments (constante, variable ou terme structuré) est analysée, afin de détecter les clauses susceptibles de satisfaire le but. Dans la WAM, seule l'analyse du premier argument permet de sélectionner des clauses. La sélection des clauses s'effectue à deux niveaux.

**Premier niveau :**

Dans le premier niveau, la discrimination se fait par le type de la valeur du premier argument. Si celle-ci est :

(1) une variable, alors tout le paquet de clauses mérite d'être retenu;
(2) une constante, ne doivent être alors envisagées que les clauses dont la tête a pour premier paramètre soit une variable, soit une constante;
(3) un terme fonctionnel (respectivement une liste), seules les clauses dont la tête a pour premier paramètre, un terme fonctionnel (respectivement une liste), peuvent être candidates à l'unification.

Le premier niveau d'indexation est réalisé par les instructions suivantes :

***switch_on_term*** *EtVar, EtConst, EtListe, EtFonct* :
permet de réaliser l'aiguillage (charger PC) vers l'étiquette correspondant à la nature du premier argument (contenu dans le registre A1).

***try_me_else*** *EtiqCL* (précède la première clause) :
crée une partie Reprise, avec comme champ BP, *EtiqCL*;
met à jour les registres BL et BG.

***retry_me_else*** *EtiqCL* (précède une clause au milieu du paquet) :
met à jour le champ BP du dernier bloc de choix à la valeur *EtiqCL*.

***trust_me_elsefail*** (précède la dernière clause) :
dépile le dernier point de choix et met à jour BL et BG.

**Second niveau :**

Dans le second niveau, la discrimination est affinée de la façon suivante :

**(1)** les constantes, sont indexées par elles-mêmes;
**(2)** les termes fonctionnels se distinguent par le symbole fonctionnel;
**(3)** les listes, sont distinguées suivant qu'elles soient vides ou non-vides.

Les instructions permettant de réaliser ce niveau d'indexation sont :

***switch_on_constant*** *N,Table* *et* ***switch_on_structure*** *N,Table* :
Table est une structure qui permet d'associer à une clé (identificateur ou symbole fonctionnel), l'étiquette de début du sous-paquet (sous forme compilée) correspondant. A l'intérieur d'un sous-paquet associé à une même clé, la gestion des point de choix s'effectue par les instructions qui suivent.

***try L*** ( associée à la première clause) :
crée une partie Reprise avec pour champ BP l'instruction qui suit;
met à jour les registres BL et BG;
charge le registre PC avec L.

***retry L*** (associée à une clause de milieu de paquet) :
met à jour le champ BP à la valeur instruction qui suit;
charge le registre PC avec L.

***trust L*** (associée à la dernière clause) :
dépile le dernier point de choix, et met à jour BL et BG;
charge le registre PC avec L.

### 2.7.2 - Instructions de contrôle :

***allocate*** :
elle est déclenchée quand une clause comportant plus d'un sous-but est invoquée. Un bloc local est créé en sommet de pile. Les registres de continuation CP et CL sont sauvegardés.

***deallocate*** :
quand les variables d'un bloc courant ne sont plus nécessaires, celui-ci est désalloué. Les registres de continuation sont restaurés.

***execute*** *Pr* (Pr est le dernier but d'une règle) :
charge PC avec Pr.

***call Proc, N*** (Appel du but Proc, N taille de l'environnement) :

| met à jour CP (instruction suivante);
| charge PC avec Proc.

***proceed*** :

| lorsque la résolution du but courant réussit (donc après un fait), le contrôle est transféré au code pointé par le registre CP en chargeant PC par CP.

### 2.7.3 - Instructions d'unification :

L'unification s'effectue entre les paramètres d'une tête de clause et les n arguments d'un but détenus par les registres Ai. L'unification de la Wam bénéficie directement du fait que l'on puisse déterminer statiquement :

| **(1)** la nature du paramètre mis en jeu : constante, variable, liste ou terme fonctionnel;
| **(2)** la première occurrence ou non d'une variable (une variable est forcément libre dans sa première occurrence);
| **(3)** les variables liées à des valeurs portées par la pile locale ou par la pile globale.

Suivant le type de l'argument, et pour une variable, suivant qu'il s'agisse ou non de sa première occurrence, on utilise les instructions *get* qui permettent d'unifier deux termes, et les instructions *unify* qui permettent de réaliser l'unification de sous-termes.

***get_const C, Ai :***

| réalise l'unification de la constante C qui est en position argument, avec le registre Ai.

***get_variable Vn, Ai :***

| réalise l'unification de la variable Vn (lors de sa première occurrence) avec le registre Ai.

***get_value Vn, Ai :***

| réalise l'unification de la variable Vn (lors de sa ième occurence, avec i > 1) avec le registre Ai.

***get_nil Ai :***

| réalise l'unification de la liste vide avec le registre Ai.

***get_list Ai :***

| réalise l'unification d'une liste non vide avec le registre Ai.

***get_structure F, Ai :***

| réalise l'unification d'une terme fonctionnel référencé par F avec le registre Ai.

Lorsque la valeur du registre a conduit à un terme structuré, les instructions *get_list* et *get_structure* sont complétées par les instructions ***unify*** qui réalisent l'unification des sous-termes. Là encore, l'on distingue trois cas suivant que le sous-terme est une constante, une variable, ou lui-même un terme structuré :

***unify_constant C :***

réalise l'unification d'un sous-terme qui est une constante C.

***unify_variable Vn :***

réalise l'unification de la variable Vn apparaissant pour la première fois dans le terme à unifier.

***unify_value Vn :***

réalise l'unification de la ième occurrence (i > 1) de la variable Vn dans le terme à unifier. Cette instruction est exécutée lorsque l'on est sûr que la première occurrence de Vn a conduit à une liaison avec une valeur portée par la pile globale. Dans le cas contraire, où l'on ne peut rien affirmer sur la nature de la première occurrence de Vn, l'instruction ***unify_locale_value Vn*** est utilisée. Elle teste au préalable si la variable Vn est portée par la pile locale. Dans ce cas, une nouvelle variable libre est empilée sur la pile globale, puis Vn est liée à cette nouvelle variable.

### 2.7.4 - Instructions de chargements de registres :

Dans La WAM, lors de l'appel d'un but, ses arguments sont sauvegardés dans des registres spéciaux, avant unification. Cette sauvegarde permet de mettre en place la transformation des appels terminaux ou le trimming, qui constituent une économie de l'espace en pile locale, nous reviendrons en détail sur ces notions dans le quatrième chapitre.

Comme pour l'unification, l'on distingue plusieurs situations suivant la nature de l'argument mis en jeu.

***put_const C, Ai :***

charge le registre argument Ai avec la constante C.

***put_nil Ai :***

permet de charger le registre Ai avec la liste vide.

***put_list Ai :***

le registre Ai est chargé avec une référence vers la zone de représentation de la liste argument.

***put_structure F, Ai :***

charge le registre Ai avec *F* qui est une référence vers le terme fonctionnel en position d'argument.

Lors des deux dernières instructions, l'argument est sauvegardé systématiquement par une recopie sur la pile globale.

Dans le cas où une variable Yn apparait en position d'argument du but appelé, deux cas doivent être distingués en raison de l'application de la transformation des appels terminaux :

le but appelé n'est pas en position terminale :

***put_variable Yn, Ai :***

charge le registre Ai avec la référence de la variable Yn lors de sa première occurrence.

A partir de la seconde occurrence de Yn, l'instruction ***put_value Yn, Ai*** est utilisée pour charger le registre Ai avec la valeur de liaison de Yn.

le but appelé est en position terminale :

***put_unsafe_value Yn, Ai :***

cette instruction a été introduite afin de contrôler l'application de la transformation des appels terminaux qui consiste à récupérer le bloc associé à une procédure déterministe, non pas à son retour mais dès l'appel de son dernier littéral. L'instruction *put_unsafe_value Yn, Ai* teste, au préalable, si Yn est une variable libre, ou liée à une variable appartenant au même environnement amené à disparaitre. Si oui , alors la variable Yn est dynamiquement ré-allouée sur la pile globale, et le registre Ai est chargé par la référence de la zone ainsi créée sur la pile globale, et, dans le cas contraire, *put_unsafe_value Yn, Ai* se comporte comme *put_value*.

# 3 - Parallélisme en programmation logique :

La programmation logique basée sur les clauses de Horn, offre les principales sources de parallélisme suivantes : parallélisme OU, parallélisme ET, parallélisme d'unification.

## 3.1 - Parallélisme OU :

Cette forme de parallélisme résulte du non déterminisme existant dans la résolution basée sur la forme clausale. Un prédicat peut être défini par plusieurs clauses (paquet de clauses). Chacune de ces clauses constitue une alternative pour résoudre le but courant. Dans la stratégie séquentielle de Prolog, un paquet de clauses est exploré séquentiellement en utilisant le retour-arrière.

Le parallélisme-OU consiste à essayer plusieurs alternatives concurremment pour résoudre le même but. Les résolutions correspondant à ces alternatives sont indépendantes, mais partagent l'état de la résolvante (une résolvante est formée par une conjonction de buts à résoudre et une substitution) à l'appel du but.

C'est ici qu'apparait le problème principal dans l'utilisation de cette forme de parallélisme. En effet, les résolutions parallèles peuvent être amenées à lier une même variable différemment.

**Exemple** : Considérons la résolvante : . . ., p(X,Y), q(Y,Z), . . . (X,Y et Z sont libres)

et la base :

```
p(a,Y) :- . . .
p(b,Y):- . . .
p(c,Y):- . . .
. . .
```

Pour résoudre le but *p(X,Y)*, les trois clauses associées au prédicat *p* peuvent être activées en parallèle. Les trois résolutions produisent trois solutions différentes qui sont :
(X = a, Y= . . .), (X = b, Y= . . .) et (X = c, Y= . . .).

La variable X se trouve donc instanciée par trois valeurs différentes.

De ce fait, le problème majeur à résoudre dans le parallélisme OU est la nécessité de gérer les liaisons multiples.

## 3.2 - Parallélisme ET :

La stratégie de résolution de Prolog choisit un littéral (le plus à gauche) de la résolvante courante et essaie de satisfaire ce but. Le parallélisme ET consiste à prendre plusieurs littéraux et à mener leur réfutation en parallèle.

Le problème qui se pose dans ce type de parallélisme résulte de la dépendance de certains littéraux. Cette dépendance peut provoquer des incompatibilités de résultats comme le montre l'exemple suivant.

**Exemple :** Considérons la résolvante : . . ., p(X), q(X), . . .

et la base :

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

Dans une résolution séquentielle, le but *q(X)* n'est pris en compte que lorsque la réfutation du but *p(X)* a produit une solution, celle-ci étant prise en considération dans la réfutation de *q(X)*. La réfutation en parallèle de p*(X)* et *q(X)* produirait une première solution X=*a* et *X*=*c* qui est incohérente.

Nous verrons dans le chapitre suivant les solutions apportées à ce problème.

## 3.3 - Parallélisme d'unification :

Il consiste à unifier simultanément les arguments de deux littéraux. Pour unifier les deux littéraux p*(a,X,b)* et p*(Z,Y,b)*, les trois unifications peuvent être exécutées en parallèle et indépendamment. Par contre, si le deuxième littéral est *p(Z,Z,b)*, l'unification des arguments *X* et *Z* ne peut avoir lieu que si la première a réussi.

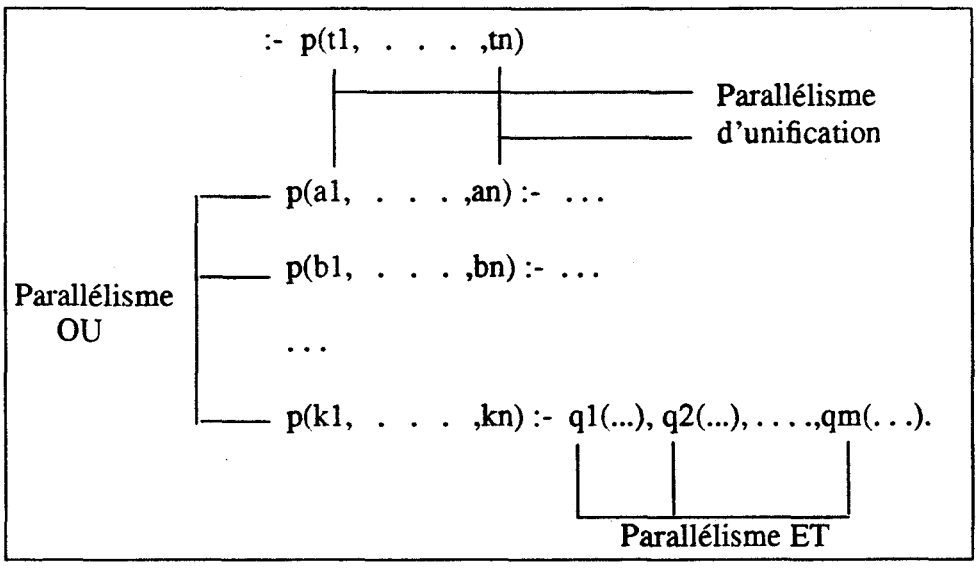



figure 1.12 : Les 3 formes de parallélisme dans Prolog

Le parallélisme d'unification caractérisé par son fin grain, peut exploiter le parallélisme de données présent dans des bases de faits de tailles importantes. Cependant, malgré que ce type de parallélisme concerne une opération importante dans Prolog, des résultats théoriques ont montré que le gain que l'on peut en espérer est très faible à cause de la complexité et de la nature séquentielle de l'unification [Dwo 84].

Le parallélisme ET est difficile à extraire automatiquement des programmes, en raison de la nécessaire cohérence des variables instanciées par les sous-buts d'une même clause. C'est pourquoi certains modèles étudiés, que nous verrons dans le chapitre suivant, donnent la possibilité au programmeur d'exprimer, lui même, le contrôle du parallélisme.

Contrairement au parallélisme ET, le parallélisme OU ne nécessite aucun contrôle pour être validé. Mais le problème majeur de cette forme de parallélisme, est le besoin de gérer les liaisons multiples. Les différentes solutions proposées à ce problème ont donné naissance à une variété de modèles OU-parallèles que nous présenterons dans le chapitre suivant.

D'autres types de parallélisme existent: parallélisme de flot (stream) qui est une variante du parallélisme ET, il consiste à lancer tous les sous-buts en parallèle, mais ceux ci vont se synchroniser automatiquement en fonction des dépendances (entre littéraux) explicitées dans le programme, parallélisme de recherche qui n'offre d'interêt que pour les programmes ayant beaucoup d'assertions, et qui peut être considéré comme une forme de parallélisme OU.

# -=- Chapitre II -=-

*Modèles et Systèmes*
*de Programmation Logique Parallèle*

La programmation logique a connu ces dix dernières années une multitude de recherches visant à exécuter les programmes logiques de façon parallèle.

En 1981, les Japonais lancèrent leur projet d'ordinateur de cinquième génération visant à concevoir des machines basées sur le concept de programmation logique et devant répondre à des besoins multiples en matière de traitement intelligent de l'information : bases de connaissances, résolutions de problèmes, langage naturel.

Ce projet a engendré une réaction internationale qui a donné naissance à une série d' efforts de recherches :

. le programme ESPRIT (European Strategic Programme for Research & development in Information Technology) lancé par la CEE;
. création de MCC (Microelectronics and Computer Technology Corporation);
. le projet Gigalips qui a réuni SICS (Swedish Institute for Computer Science), "Argonne National Laboratory" (USA), l'université de Bristol et plus récemment celle de Manchester;
. le projet PEPsys (Parallel ECRC Prolog Sytem) effectué à l'ECRC (European Computer industry Research Centre) fondé par Bull, ICL et Siemens.

La principale raison de cette envolée de recherches réside d'une part, dans l'intérêt croissant pour les techniques dites d'intelligence artificielle, et d'autre part, dans le fait que la programmation logique offre un pouvoir d'expression élevé et un haut degré de parallélisme intrinsèque qui peut être exploité dans des implantations sur machines multiprocesseurs.

Dans ce chapitre, nous présenterons les deux axes de recherches en programmation logique parallèle, à savoir, les systèmes parallèles logiques gardés et les systèmes non-déterministes.

Pour le premier axe, nous rappellerons les caractéristiques générales des systèmes gardés qui exploitent le parallélisme de flot et fournissent des mécanismes de synchronisation inter-processus. Nous citerons quelques systèmes basés sur ces concepts.

En ce qui concerne le second axe de recherche, nous montrerons les solutions apportées aux problèmes liés à l'utilisation du parallélisme ET ou OU, celui-ci constituant la principale source de parallélisme pour les systèmes non-déterministes. Nous nous concentrerons sur les modèles dits multi-séquentiels dont l'objectif est de fournir des systèmes efficaces pour l'exécution des programmes symboliques écrits en Prolog, sur des machines multiprocesseurs.

# 1 - Les systèmes parallèles logiques gardés :

Les systèmes parallèles logiques gardés fournissent au programmeur des dialectes pour expliciter le contrôle du parallélisme à travers des concepts de flots et de synchronisation.

## 1.1 - Caractéristiques des langages gardés :

. **La garde,** qui consiste en la spécification, pour chaque clause du programme, d'une condition qui doit être satisfaite pour que le corps de la clause puisse être exécuté.

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

. **La synchronisation**, entre processus par indication du mode d'accès (lecture ou écriture) des arguments.

Dans un langage gardé, une clause à la forme suivante :

H :- G1, . . . .,Gn | B1, . . . .,Bp. n, p >= 0

H est la tête de clause.
G1, . . .,Gn la garde de la clause.
| est appelé opérateur d'engagement (*commit*).
B1, . . .,Bp représentent le corps de la clause.

La sémantique déclarative de cette clause est celle d'une clause de Horn si l'on ignore l'opérateur d'engagement, c'est à dire :

H est vrai si G1et . . . et Gn et B1et . . . et Bp.

Mais sa sémantique (procédurale) est plus complexe. La réfutation d'un but s'effectue en lançant simultanément autant de processus qu'il y a de clauses alternatives pouvant satisfaire ce but, chacun des processus étant chargé d'exécuter l'unification avec la tête d'une clause et la garde de celle-ci. Dès qu'une garde est évaluée avec succès, les autres processus sont arrêtés et seule l'exécution de la clause contenant cette garde continue.

Ainsi le parallélisme OU résulte des deux premiers points évoqués, en se limitant à l'exécution simultanée des unifications avec les têtes de clauses et l'évaluation des gardes de celles-ci.

Le troisième point permet de gérer le parallélisme ET 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 (Parlog) ou pour chaque littéral (CP) le mode d'accès à ces arguments (lecture/écriture). Une occurrence de variable en entrée dans un littéral indique que celui-ci est consommateur, aussi son exécution est-elle mise en attente tant que cette variable n'a pas été liée par un littéral producteur (littéral où cette variable figure en sortie). Ainsi le problème de l'incohérence des liaisons dû au parallélisme ET est inexistant.

L'objectif des langages gardés est de devenir des langages de base pour les machines parallèles (langages systèmes) ainsi que pour les machine séquentielles. Plusieurs implantations ont été réalisées sur mono et multi-processeurs.

## 1.2 - Spécifités des langages gardés :

Dans ce paragraphe, nous présentons les traits spécifiques à chacun des principaux langages gardés : Parlog, CP et GHC. Une étude exhaustive est présentée dans [Chas 89a].

**Parlog** (Imperial College, G.B) :

Parlog utilise des déclarations de modes explicites qui spécifient pour chaque prédicat le sens d'accès (entrée ou sortie) à ses arguments. Une déclaration de mode pour un prédicat *p* et d'arité *n* a la forme :

$$\text{mode } p(m_1, \ldots, m_n).$$

Chaque $m_i$ est un symbole "?" ou "^" qui indique que le ième argument est en entrée (symbole ?) ou en sortie (symbole ^).

Lors de l'unification avec une tête de clause, une tentative de liaison d'une variable d'un argument déclaré en entrée, provoque la suspension du processus qui effectue l'unification, jusqu'à ce que cette variable soit liée par un autre processus.

D'autre part, Parlog utilise des annotations pour forcer l'exécution séquentielle : l'opérateur *&* entre deux littéraux indique que l'évaluation du deuxième littéral n'a lieu que si le premier d'entre eux a été évalué avec succès. La séparation des clauses d'un même paquet par "." ou ";" indique que les gardes des clauses se terminant par "." sont évaluées en parallèle, et une clause se terminant par ";" est essayée seulement si toutes les gardes des clauses précédentes ont échoué.

CP Concurrent Prolog (Weizmann Institute, Israel) :

CP utilise des annotations sur les variables qui indiquent que l'accès à celles-ci ne peut se faire qu'en lecture. On peut annoter une variable apparaissant en tête ou dans le corps d'une clause. Ainsi dans une clause *p(X,Y):-q(X,Z),r(Z?,Y)*, le processus chargé de résoudre *r* ne peut lier la variable *Z* que lorsque celle-ci est instanciée par le processus démontrant le but *q*.

L'annotation de CP exige un algorithme d'unification capable, d'une part, de suspendre un processus qui est sur le point d'instancier une variable annotée, et d'autre part, de gérer les annotations à cause de leur propagation dynamique; si dans l'exemple précédent, la variable Z est liée par *p* au terme *f(X)*, *X* devient une variable annotée (*X?*) pour *r*.

La sémantique de cette annotation n'est pas bien définie et pose plusieurs problèmes. Dans [Sar 87] une autre annotation (↓) est définie clairement pour lever les ambiguités sémantiques posées par une telle annotation.

GHC (Guarded Horn Clauses) (ICOT , Japon) :

GHC n'utilise ni déclarations de modes ni annotations. La synchronisation sur les variables partagées est mise en oeuvre en appliquant la règle suivante :

"Ni l'unification, ni l'évaluation de la garde ne peuvent conduire à l'instanciation d'une variable de l'appelant". Si une telle liaison est tentée, l'exécution est suspendue. Les liaisons des variables de l'appelant sont effectuées explicitement dans le corps de la clause en utilisant le prédicat "=".

Exemple : Considérons l'évaluation du but <- p(X), q(X),
et les clauses : p(a) <- true | . . .
q(Z) <- true | Z=a.

L'unification de p(X) avec p(a) entraîne la suspension du processus chargé de résoudre *p* car il y a tentative d'instanciation de la variable *X* du but. Par contre, la résolution de *q* peut unifier *X* à *Z*, puis effectuer la liaison *Z=a*, ce qui permet d'instancier *X*.

## 1.3 - Problèmes posés par les langages gardés :

Le principal problème des langages gardés est leur sémantique qui,à ce jour, n'a pas encore fait l'objet d'un consensus. L'opérateur d'engagement et le *don't care non-déterminism* introduisent des problèmes d'incomplétudes et des phénomènes de blocage.

En effet, le concept d'engagement, qui écarte définitivement les clauses alternatives même si celles-ci peuvent conduire à un succès, risque de donner une réponse négative à une question qui possède une solution au vu des clauses de Horn du programme. Les mécanismes de synchronisation peuvent mener à des interblocages (*dead-lock*) entrainant la non-terminaison de la résolution..

Une autre catégorie de problèmes concerne l'efficacité de leurs implantations qui mettent en jeu un nombre important de processus, et en particulier des processus suspendus, ce qui engendre des changements de contexte très fréquents. Il en résulte une fine granularité.

# 2 - Les systèmes non déterministes :

Contrairement aux systèmes logiques parallèles basés sur le concept de garde, les modèles non-déterministes exploitent uniquement le parallélisme implicite à Prolog en essayant de garder une totale compatibilité avec ce dernier. Ainsi la principale caractéristique "multi-solutions" de Prolog séquentiel réalisée par le retour-arrière est offerte par les systèmes parallèles non-déterministes.

On distingue trois modèles (ET, OU et ET-OU) selon la forme de parallélisme qu'ils exploitent.

## 2.1 - Les modèles ET-OU :

Les modèles ET-OU [Conery 83] combinent les deux sources de parallélisme intrinsèque à Prolog. Chaque étape de l'exécution d'un programme donne naissance à deux types de processus : processus-ET et processus-OU, un processus-ET étant chargé de résoudre le corps d'une clause alors qu'un processus-OU est chargé de résoudre un littéral d'une clause.

**Exemple :** p(X,Y):-q(X,Z),r(Z,Y).
q(. . .):- . . .
q(. . .):- . . .
r(. . .):- . . .

L'exécution d'un but initial p(. . .)? crée un processus-ET pour résoudre la clause *p*; ce processus crée à son tour deux processus-OU pour résoudre *q* et *r*, qui eux mêmes créent respectivement deux et un processus-ET, etc ... (*fig 2.1*).

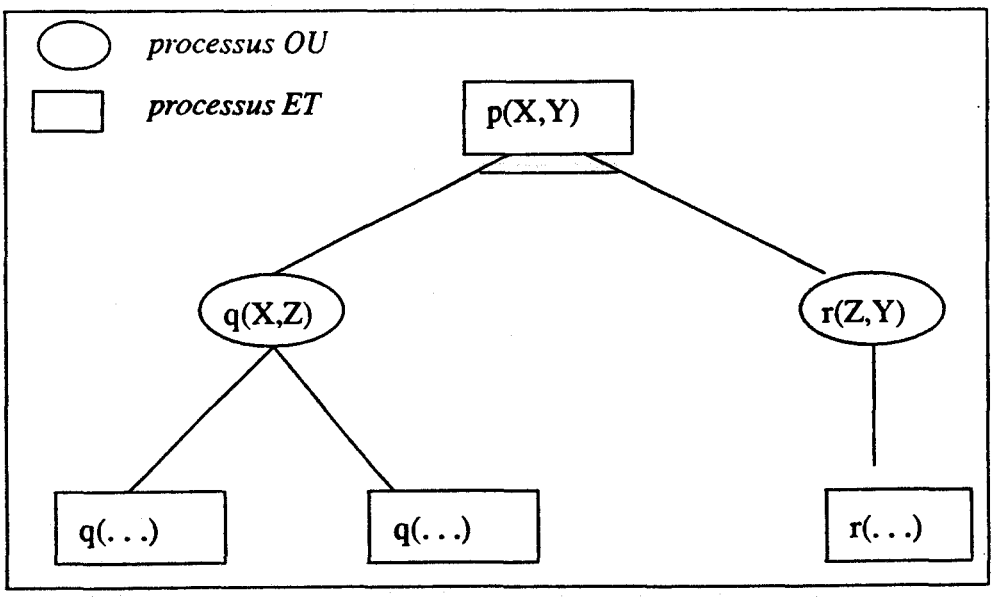



figure 2.1 : Arbre ET/OU des processus.

Les liens de parenté entre les processus définissent un arbre ET/OU. Chaque processus est capable d'échanger des messages avec ses fils et avec son processus père. Les messages diffusés vers les fils permettent de commencer l'exécution (*start*), d'obtenir une solution (*redo*) et d'arrêter une exécution (*cancel*) devenue inutile. Les messages envoyés au processus père permettent de véhiculer les solutions lors d'un succès (*success*), ou d'annoncer un échec (*fail*).

Le parallélisme-OU est lié à l'exécution simultanée des corps des clauses dont les têtes s'unifient avec le même but; ceci correspond à l'exécution de plusieurs processus-ET fils d'un même processus-OU. Le parallélisme ET a lieu lorsque plusieurs processus-OU ayant le même noeud père (noeud ET), sont exécutés simultanément.

L'une des principales caractéristiques des modèles ET-OU est l'utilisation d'algorithmes complexes pour extraire le parallélisme ET.

La deuxième caractéristique de ces modèles est leur fin grain de parallélisme. En effet, ils mettent en jeu un grand nombre de processus ayant en général besoin de communiquer de manière importante et de se synchroniser.

Les modèles présentés ci-après, appliquent la même philosophie, dans la mesure où l'objectif visé est l'obtention d'un parallélisme massif

**Le modèle COALA** :

Dans le schéma d'exécution COALA (Calculateur Orienté Acteur pour la Logique et ses Applications) [Per 86] [Dur 86], un programme est précompilé en un graphe dont les noeuds sont les clauses du programme, et où un littéral du corps d'une clause est relié avec un littéral en tête d'une autre clause, par un arc indiquant que ces deux littéraux sont unifiables. Cet arc est étiqueté par la substitution correspondante, celle-ci étant appelé environnement d'unification de l'arc.

Dans ce modèle, les arcs sont considérés comme des éléments de base (acteurs), et l'exécution d'un programme consiste à déplier le graphe statique précompilé, en ajoutant et en supprimant des arcs. Le parallélisme OU est obtenu en activant plusieurs arcs issus d'un même littéral; le parallélisme ET correspond à l'unification simultanée de l'environnement choisi avec les environnements des autres arcs issus des clause parentes.

Le parallélisme ET, induit par le parallélisme OU, a été étendu pour développer en parallèle les arcs issus de différents littéraux d'un résolvant.

**Le modèle MIPAP** :

Le modèle MIPAP (Modèle d'Interprétation PArallèle pour Prolog) [Kha 88] s'inspire de [Con 83] et des techniques de détermination statique du parallélisme ET [Deg 84].

Le modèle s'appuie sur des processus (appelés noeuds) qui échangent des messages. On retrouve les mêmes types de processus et de messages introduits dans [Con 83]. L'un des principaux intérêts de ce modèle repose sur la détection statique du parallélisme ET qui permet de diminuer le coût des tests à l'exécution par rapport à [Con 83].

**Le modèle LOG_ARCH** :

Le schéma d'évaluation proposé dans le cadre du projet LOG_ARCH [Han 89], a pour objectif de permettre l'exécution des programmes écrits dans le langage Prolog, sur des architectures distribuées, et ce afin de pouvoir utiliser un grand nombre de processeurs.

Le modèle repose sur une exécution basée sur des processus communiquants. L'accent est mis principalement sur le contrôle de l'explosion de l'arbre des processus, et sur la gestion des environnements [Bou 90]. Pour cela, le modèle introduit un mécanisme d'activation "restreinte" du parallélisme OU permettant de contrôler la création des processus. Une forme de parallélisme ET appelé "pipeline" est exploitée afin d'éviter les liaisons conflictuelles provoquées par les variables partagées.

## 2.2 - Les modèles ET-parallèles :

Rappelons que la parallélisme ET est obtenu par l'exécution simultanée de plusieurs sous-buts d'une même clause. Contrairement au parallélisme OU qui n'apparait que dans les programmes non-déterministes, le parallélisme ET n'est pas spécifique à une catégorie de programmes logiques, et il est présent dans de nombreux programmes en particulier dans les programmes déterministes.

Cependant le parallélisme ET est difficile à extraire à cause des dépendances de littéraux. La solution la plus naturelle pour pallier ce problème est de lancer indépendamment tous les sous-buts d'une même clause et d'effectuer ensuite une jointure des solutions retournées, pour assurer la cohérence des liaisons.

Mais le gros inconvénient de cette solution, est le coût de réalisation d'une jointure, qui peut être très élevé. Considérons la clause *p(X,Y) :- q(X,Z),r(Z,T),s(T,X)*, et supposons qu'il existe respectivement $10^5$, $10^5$ et $10^9$ solutions à *q*, *r* et *s* (ce qui est possible dans le cas d'une base de données), le nombre de tests pour contrôler la cohérence des solutions est égal au produit des nombres de solutions de chaque sous-but, i.e $10^5 * 10^5 * 10^9 = 10^{19}$. Si l'on suppose que chaque test prend une nano-seconde, le contrôle de la compatibilité des liaisons prendrait 317 ans.

D'autre part, cette solution semble peu réaliste étant donné qu'elle risque de mettre en jeu un nombre important de processus communiquants, qui nécessiterait une gestion d'un grand nombre d'environnements d'unification.

Les solutions qui sont proposées pour résoudre le problème des liaisons conflictuelles ont donné plusieurs formes au parallélisme ET.

### 2.2.1 - Parallélisme de flot :

Le parallélisme de flot est défini par l'expression des dépendances entre les littéraux d'une clause. Ces dépendances sont exprimées dans le programme par des déclarations de mode d'accès aux arguments d'un prédicat, ou bien en spécifiant de quelles variables un littéral est producteur ou consommateur. C'est l'approche utilisée dans les langages gardés. Cette explicitation permet de lancer tous les buts en parallèle, mais ceux-ci vont se synchroniser automatiquement en fonction des dépendances énoncées dans le programme.

### 2.2.2 - Parallélisme pipeline :

Le parallélisme pipeline repose sur la possibilité de décomposer, en *n* étapes, la résolution d'un but <-q1, q2, . . . .,qn : la ième étape consiste à chercher toutes les solutions du sous-but qi et l'étape suivante est la validation de ces solutions par le but qi+1, qui lui même est capable de produire des solutions.

Ces étapes peuvent être exécutées par un algorithme pipeline. La résolution opère de gauche à droite comme dans un interprète séquentiel : lorsque le processus chargé de résoudre le sous-but qi (processus qi) produit une solution, celle-ci est communiquée au processus qi+1, pendant que le processus qi cherche d'autres solutions.

La mise en oeuvre d'un tel modèle d'exécution nécessite l'utilisation d'un tampon de messages entre chaque deux étages du pipeline, pour mémoriser les solutions produites par l'un et consommées par l'autre. Le parallélisme ET est une conséquence directe du pipeline, mais il est étroitement lié à la taille du tampon : lorsque celui-ci est plein, l'exécution est équivalente à une exécution séquentielle.

Le parallélisme pipeline nécessite l'existence de plusieurs alternatives pour un même sous-but. Dans les programmes déterministes, l'exécution reste séquentielle.

### 2.2.3 - Parallélisme ET indépendant:

Le parallélisme ET indépendant consiste à considérer que seuls les sous-buts ne risquant pas de provoquer un conflit de liaisons peuvent être exécutés de manière parallèle. Dans ce cas, les résolutions parallèles de sous-buts sont indépendantes et ne se synchronisent que rarement, contrairement au parallélisme ET dépendant qui existe dans les systèmes gardés. En contrepartie, le parallélisme ET disponible est plus faible que celui offert par les langages gardés.

**Exemple :** Considérons l'exemple suivant donné dans [Cha 89b]:
Soit le but à résoudre : :- p(X),q(X,Y).
et les clauses suivantes définissant p et q: p([a|Y]) :- p'(Y)
q([a|Y],Z) :- q'(Z)
q([b|Y],Z) :- q''(Z)

En parallélisme ET indépendant, l'évaluation des sous-buts *p(X)* et *q(X,Y)* s'effectue de manière séquentielle puisque ceux-ci partagent la variable Y.

L'exécution de *q* ne commence que lorsque celle de *p* (donc de *p'*) est terminée, alors qu'à priori *q* peut être exécuté dès que la variable X est liée à [a|Y], sans attendre que p' soit terminé.

Une telle exécution est possible dans un langage gardé.

Dans le parallélisme ET indépendant, le problème majeur à résoudre est la détection de l'indépendance entre littéraux, qui est difficile à réaliser de manière statique. L'activation de la clause *p(X,Y):-q(X),r(Y)* avec le but *p(U,U)?* lie la variable *Y* à *X*, ce qui rend les deux littéraux *q* et *r* dépendants.

La première solution a été proposée par J.Conery [Con 83]: il s'agit d'un mécanisme de détection dynamique qui détermine la possibilité du parallélisme lors de l'activation d'une clause. L'intérêt majeur de cette méthode est qu'elle permet d'extraire tout le parallélisme possible, mais son principal défaut est un coût élevé.

Pour diminuer le coût de détection de l'indépendance de littéraux, D. DeGroot [DeG 84] propose une méthode, appelée **parallélisme ET restreint**, qui repose sur une analyse statique du programme, celle-ci produisant des tests qui permettront, à l'exécution, de déterminer si des sous-buts peuvent être exécutés en parallèle. Les tests produits sont des conditions sur la nature des valeurs de liaisons des variables.

Le parallélisme ET restreint, même s'il ne permet pas de détecter tout le parallélisme possible, semble apporter la meilleure solution au problème des dépendances de littéraux.

## 2.3 - Les modèles OU-parallèles :

Les modèles OU-parallèles exploitent le parallélisme (parallélisme OU) existant dans le non-déterminisme offert par la programmation logique. En effet, l'ordre des clauses définissant un même prédicat n'existe que pour des raisons d'efficacité dans l'implantation de la stratégie de résolution *en profondeur d'abord*, ou lorsque les effets de bord sont à considérer. Ainsi pour résoudre un but donné, plusieurs clauses peuvent être essayées concurremment.

**Exemple :** Considérons le programme suivant :

```
but :   p(X), q(X,Y) . . .
base :  p(a).
        p(b).
        p(X) :- . . . . , X = c, . . . .
```

--> pour résoudre le but *p(X)*, les trois clauses définissant le prédicat *p* peuvent être activées en parallèle.

Le parallélisme OU n'est présent que dans les programmes non déterministes tant dans les programmes fournissant beaucoup de solutions que dans ceux qui n'en produisent que peu. En effet, si le but d'un programme est de trouver une seule solution, l'exploration de l'arbre de recherche tire avantage de l'utilisation de ce type de parallélisme.

Contrairement au parallélisme ET, le parallélisme OU peut offrir une grosse granularité. En effet, il est possible de distinguer deux situations lorsqu'un sous-but est résolu avec succès :

- Le processus (processus OU), chargé de résoudre ce sous-but, meurt après avoir communiqué les solutions au processus père (processus ET). Ce dernier a la charge d'activer le sous-but suivant. Lorsque les trois résolutions de *p* se terminent, elles renvoient leurs solutions au processus père qui active le processus q avec chacune des solutions.

C'est l'approche utilisée dans les modèles ET-OU (voir le paragraphe 2.1). Le grain de parallélisme est très fin, puisque les processus communiquent fréquemment, et ont généralement une durée de vie courte.

- Lorsqu'un processus résoud un sous-but avec succès, au lieu de renvoyer les solutions au processus père, il utilise le mécanisme de continuation séquentiel pour invoquer le sous-but suivant.

De ce fait, les messages véhiculant les solutions sont supprimés et seules les communications nécessaires à l'initialisation des processus sont à supporter. Le parallélisme peut être exploité à grosse granularité.

En effet, dans l'exemple ci-dessus, même si la résolution de *p* ne nécessite pas beaucoup de calcul, le grain de parallélisme peut être gros si l'exécution du but *q* prend beaucoup de temps.

La majorité des modèles OU-parallèles étudiés sont basés sur cette deuxième approche, qui permet d'utiliser les techniques efficaces d'implantation séquentielle. C'est pourquoi ces modèles sont parfois appelés modèles multi-séquentiels.

### 2.3.1 - La stratégie multi-séquentielle :

Le but des modèles multi-séquentiels est de contrôler la création des processus en fonction des ressources disponibles. Un processus est considéré comme étant une instance de la machine abstraite de Warren et peut être implanté comme un processeur physique dans une architecture spécialisée, ou comme un simple processus du type processus unix. L'arbre de recherche qui est développé en parallèle peut être calqué sur l'arbre constitué par les piles manipulées par les processus.

Le nombre des processus actifs à un instant donné est borné. De ce fait, l'évaluation d'un programme doit se faire en combinant la stratégie parallèle (en largeur) et la stratégie séquentielle (retour arrière) pour parcourir l'arbre de recherche.

**Exemple :** Considérons la question p(U,V)? dans la base suivante :

```
p(X,Y) :- q(X,Z),r(Z,Y).
q(a,a).   q(b,b).
r(a,b).   r(b,a).
```

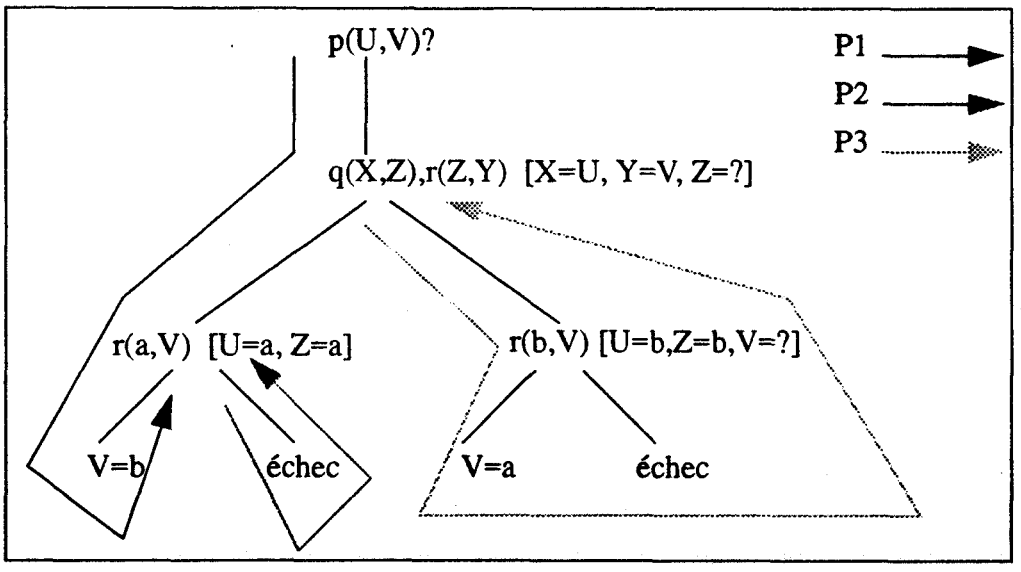



figure 2.2 : Parcours de l'arbre de recherche par trois processus *P1*,*P2* et *P3*.

Le processus *P1*, commençant la résolution, crée un point de choix sur *q*. La deuxième alternative est traitée par le processus *P3* pendant que *P1* poursuit la résolution avec la première alternative associée à *q*. Le processus *P1* crée un deuxième point de choix sur *r*, dont la deuxième alternative est prise en charge par le processus *P2*. Le processus *P3* crée à son tour un point de choix sur *r*, mais cette fois, les alternatives de ce point de choix sont traitées séquentiellement en utilisant le retour-arrière, étant donné qu'il n'existe pas de processus disponible pour prendre en charge la deuxième alternative. On suppose que le nombre de processus est limité à trois.

Un système parallèle logique opérant selon la stratégie multiséquentielle est généralement vu comme un ensemble de processus où :

(1) Un processus actif poursuit une résolution selon le modèle séquentiel et indépendamment des autres processus. Il peut produire des points de choix (noeuds "OU" rencontrés) qui sont traités par des processus inactifs si le nombre de ressources disponibles le permet. Si le nombre de ressources est insuffisant, les points de choix sont considérés lors du retour arrière.

(2) Un processus inactif acquiert du travail auprès d'un processus actif possédant un ou plusieurs points de choix en attente. La stratégie multiséquentielle a l'avantage de ne nécessiter aucun contrôle centralisé, qui pourrait constituer un goulot d' étranglement, pour la distribution des différentes tâches sur les processeurs. Seule une règle de choix (scheduling) pour "voler" [Bri 90] une résolvante en attente chez un processus actif peut être utilisée pour prendre en compte les propriétés non logiques du langage (par exemple, les effets de bord) ou pour assurer l'efficacité du parallélisme.

(3) L'acquisition d'un travail (installation d'une tâche) est la seule communication entre le processus demandeur du travail et le processus fournisseur du travail.

(4) Les processus partagent nécessairement une partie de leur résolvante qui est la partie commune de l'arbre de recherche.

Le dernier point est le principal problème étudié dans le modèles OU-parallèles. En effet, une partie commune de l'arbre de recherche peut être accédée en lecture comme en écriture par les processus actifs qui la partagent. Cette partie est constituée par l'état de la résolvante avant le dernier point de choix créé dans cette partie. La résolvante est constituée par les sous-buts restant à résoudre et la substitution obtenue par les unifications. Mais l'existence de variables libres dans la résolvante peut provoquer des écritures multiples : l'activation de plusieurs alternatives pour résoudre un même but peut conduire à lier différemment une variable figurant dans la partie commune.

Pour pallier ce problème, plusieurs solutions ont été proposées et ont donné naissance à une variété de modèles OU parallèles. Ces modèles sont généralement basés sur deux approches différentes [Ali 88] : la recopie ou le partage de l'environnement de calcul.

### 2.3.2 - La recopie de l'environnement de calcul :

Dans les modèles basés sur la recopie de l'environnement, chaque processus possède une copie de la partie de l'environnement qui est commun avec les autres processus. L'initialisation d'un processus consiste à créer une copie de la partie commune. La création de cette copie peut être effectuée par duplication de calcul ou par recopie.

#### 2.3.2.1 - La recopie :

Dans les modèles basés sur la recopie (Kabu Wake [Sho 85], Muse [Ali 90], OPERA [Bri 90]), lorsqu'un processus acquiert du travail (alternative en attente), il recopie tout l'historique de l'exécution précédant la création de ce travail. Ce n'est qu'à ce moment que le coût de l'utilisation du parallélisme est supporté.

Les processus actifs évoluent en totale indépendance, mais le gros inconvénient de ces modèles est le taux important de déperditions dues à l'initialisation des processus.

Cette initialisation nécessite la recopie de l'état des piles du processus fournissant le travail, entre "leur bases et leurs sommets telles qu'elles étaient lors de la création du point de choix donnant le travail." .

Parmi les techniques qui ont été utilisées pour réduire le taux de déperditions, on peut citer : la recopie incrémentale dans Muse[ Ali 90] et l'utilisation, dans la BC-machine [Ali 88], d'un réseau d'interconnexion qui permet à un processeur (maître) de diffuser simultanément les liaisons qu'il effectue dans les mémoires de processeurs (esclaves) inactifs, ceux ci devenant actifs dès que le travail créé par le processeur maître est jugé important (dépassant un certain seuil).

#### 2.3.2.2 - La duplication du calcul :

Cette technique a été proposée pour la première fois par Clocksin [Clo 88]. Afin d'éviter toute recopie, la partie commune est calculée par chaque processus qui y accède. Si l'on considère l'arbre de recherche de la figure 2.2, parcouru par les 3 processus *P1*,*P2* et *P3*, dans une approche basée sur la duplication de calcul, l'exploration de l'arbre de recherche peut être réalisée comme dans la figure 2.3.

Le travail attribué à chaque processus est déterminé statiquement. Clocksin propose un codage des branches de l'arbre de recherche (oracles), qui permet d'identifier pour chaque processus, les chemins à parcourir.

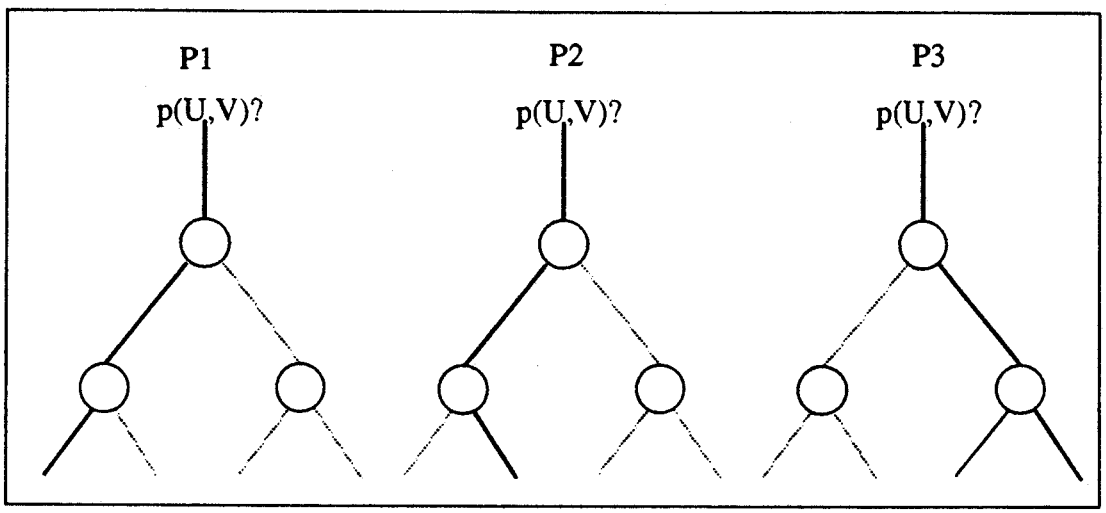



figure 2.3 : Branches parcourues par les processus *P1,P2* et*P3 dans le modèle Delphia*

Les avantages de cette technique sont, d'une part, l'absence de toute communication entre processus et, d'autre part, un accès aux variables réalisé de la même façon que dans le modèle séquentiel. Mais l'inconvénient majeur de ces modèles est la perte du parallélisme due au calcul redondant effectué, et la difficulté d'intégrer les effets de bord.

Dans les modèles basés sur la recopie ou la duplication du calcul, un espace d'adressage local est alloué à chaque processus. C'est pourquoi ces modèles font généralement l'objet d'implantation sur des architectures à mémoires distribuées.

### 2.3.3 - Le partage de l'environnement :

Contrairement aux modèles précédents, dans lesquels chaque processus dispose d'une copie de la partie de l'arbre de recherche qui est partagée avec les autres processus, dans les modèles effectuant le partage (PEPsys [Bar 88], SRI [War 87], ANL-WAM [But 86], Versions-Vectors [Hau 87], la partie commune de l'arbre de recherche existe en un seul exemplaire et est accédée par l'ensemble des processus qui la partagent.

Le seul problème à résoudre est celui des liaisons multiples : une variable libre qui apparait dans une partie commune à plusieurs processus peut être liée différemment, ce qui peut entraîner des écritures multiples.

Il s'agit donc de trouver un mécanisme qui permette de représenter les différentes valeurs de liaison d'une même variable et de contrôler la validité de chacune des liaisons pour chaque processus.

### 2.3.3.1 - Représentation des liaisons :

Dans Prolog séquentiel, une liaison est effectuée en écrivant la valeur de cette liaison dans la cellule qui représente la variable, l'adresse de la cellule étant généralement le nom de la variable.

Ce mécanisme de liaison (liaison superficielle) n'est pas suffisant quand le parallélisme OU est utilisé à cause des liaisons multiples qui peuvent coexister.

Un deuxième type de liaison (liaison profonde) a été introduit pour représenter les différentes liaisons d'une même variable : il consiste à stocker la paire (variable, valeur) dans une zone qui est généralement une table de hachage.

### 2.3.3.2 - Validation des liaisons :

On peut distinguer deux types de liaison de variables : liaison conditionnelle ou universelle. Une liaison conditionnelle a lieu lorsque un ou plusieurs points de choix ont été créés entre la création de la variable et sa liaison. Si, à l'inverse, une variable est liée avant qu'une alternative ne soit rencontrée, la liaison est dite universelle et la variable ne peut être liée une deuxième fois.

Dans une exécution OU-parallèle, toutes les liaisons universelles apparaissant dans une partie commune à plusieurs processus sont valides pour ceux-ci. Mais ce n'est pas toujours le cas pour les liaisons conditionnelles, comme le montre l'exemple de la figure 2.4 :

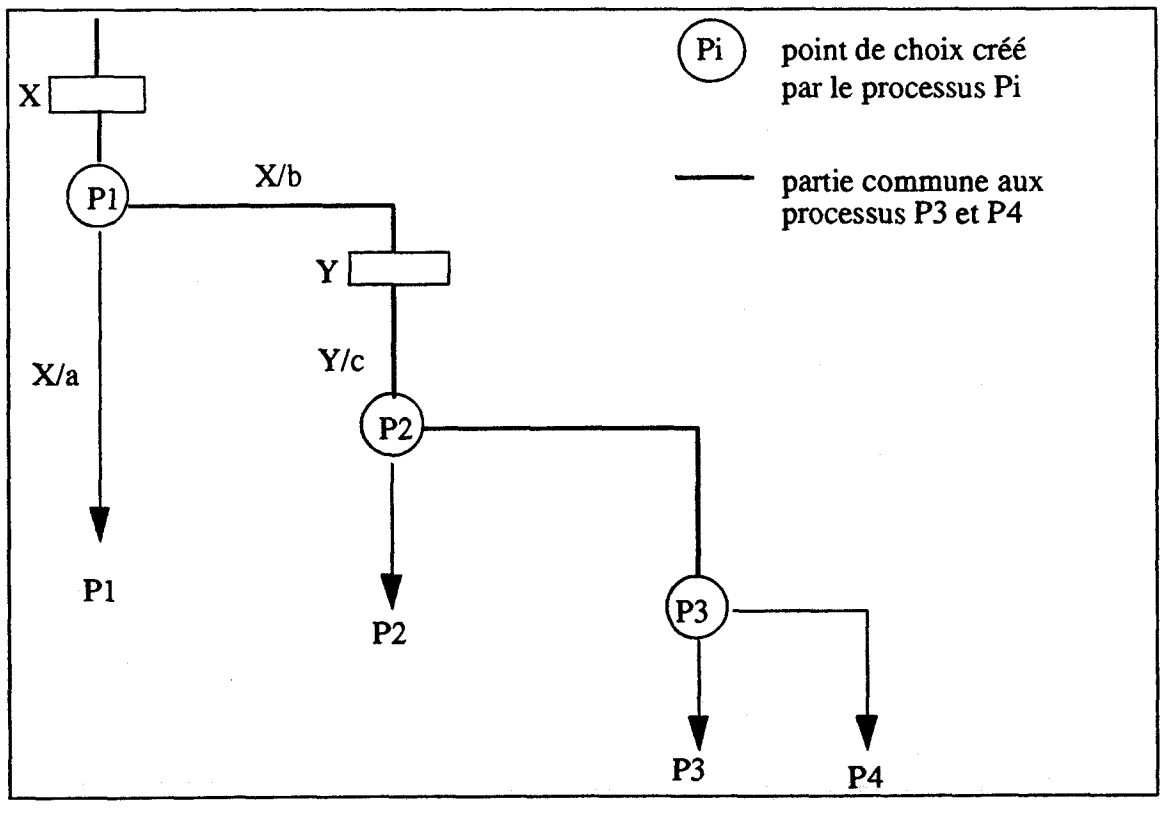



figure 2.4 : Arbre de recherche exploré par 4 processus

Le processus *P1*, commençant la résolution, crée la variable *X* (i.e *P1* active une clause qui contient la variable *X*) qu'il lie à *a* après avoir rencontré un point de choix : il s'agit donc d'une liaison conditionnelle.

Le processus *P2* prend en charge la deuxième alternative du point de choix créé par *P1*; après avoir lié conditionnellement la variable *X* à b ( la liaison X/*a* effectuée par *P1* n'est pas valide pour *P2*), *P2* crée la variable *Y* qu'il lie à c avant de rencontrer le premier point de choix.

Dans la partie commune au processus *P3* et *P4*, la liaison *Y/c* est valide pour les deux processus (la liaison est universelle), et la liaison conditionnelle *X/c* est également valide pour *P2* et *P3* (car elle a lieu avant la création du point de choix qui a donné naissance au deux processus).

Le partage de l'environnement nécessite donc une technique pour représenter plusieurs liaisons d'une même variable, et un mécanisme qui permette à un processus de valider une liaison effectuée par un processus ancêtre. Toute liaison universelle est valide, mais lorsqu'il s'agit de liaison conditionnelle, chaque processus doit être capable de détecter la position de celle-ci dans l'arbre de recherche afin de déterminer sa validité.

Les modèles que nous allons présenter par la suite proposent différentes méthodes pour résoudre ces problèmes, et présentent tous un compromis entre le coût de la gestion des liaisons multiples et le coût de l'initialisation des processus (installation des tâches).

### 2.3.3.3 - Le modèle PEPsys :

Le modèle Pepsys utilise une technique de marquage qui permet de valider les liaisons. Toute liaison est étiquetée par un numéro appelé OBL (Or Branch Level) qui correspond au nombre de points de choix créés par le processus effectuant la liaison.

Lorsqu'il s'agit d'une variable locale (variable créée par le processus), le mode de liaison superficielle est choisi en étiquetant cette liaison par l'OBL courant.

Quand il s'agit d'une variable non locale (variable créée par un processus ancêtre), le triplet (variable , valeur , OBLcourant) est stocké dans une table de hachage, chaque processus disposant d'une telle table pour stocker les liaisons profondes effectuées.

L'accès à la valeur d'une variable (déréférencement) locale est identique au mécanisme utilisé dans le modèle séquentiel, et qui consiste à parcourir la chaîne de liaison. Mais le déréférencement d'une variable non locale est plus complexe.

Il s'agit d'abord de valider une éventuelle liaison effectuée par le processus qui a créé cette variable (processus ancêtre), en comparant le champs *OBL de* cette liaison avec l'âge du point de choix qui a été créé par le processus ancêtre et qui a donné naissance au processus effectuant le déréférencement (l'âge d'un point de choix étant l' *OBL* courant du processus l'ayant créé).

Si cette comparaison ne valide pas une telle liaison, une première recherche est effectuée dans la table de hachage du processus courant. Si cette recherche échoue, une seconde recherche est réalisée donnant lieu à un parcours de la chaîne des tables de hachage des processus ancêtres susceptibles d'avoir lié la variable.

**Exemple :** La figure 2.5 illustre le déréférencement de la variable X par le processus *P3*.

La cellule représentant la variable est d'abord accédée. La liaison superficielle X/a n'est pas valide puisque l'OBL sous lequel elle est enregistrée (= 5) est supérieur à l'âge du point de choix (= 4) qui a donné naissance au processus *P3*. Aucune liaison conditionnelle de X ne se trouve dans la table de hachage de *P3*. La table de hachage de *P2* ne contient aucune liaison valide pour *P3*. La recherche se termine donc puisque la chaîne des tables de hachage des processus ancêtres est réduite à un élément. Le résultat du déréférencement est donc : la variable X est libre pour le processus *P3*.

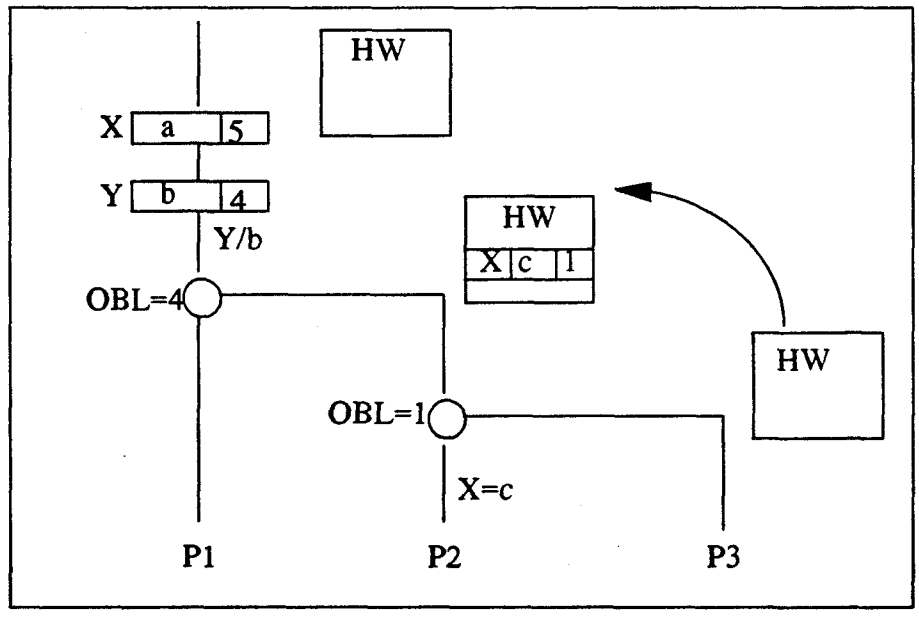



figure 2.5 : Mécanisme de liaison dans le modèle PEPsys

Cette technique, n'entraînant aucune recopie, permet l'installation d'une tâche parallèle à un coût très réduit. Elle présente toutefois un gros inconvénient : l'accès aux variables liées conditionnellement nécessite dans certains cas le parcours d'une chaîne de tables de hachage. La longueur de cette chaîne peut croître arbitrairement.

### 2.3.3.4 - Le modèle ANL-WAM :

Dans le modèle ANL-WAM, développé à "Argonne National Laboratory", chaque processus distingue trois types de liaisons, suivant la section dans laquelle se trouve la variable : section privée, favorisée ou partagée.

La section privée d'un processus contient toutes les variables créées entre le sommet de pile et la plus jeune alternative utilisée par un autre processus.

La section favorisée d'un processus contient toutes les variables créées par le processus et qui ne sont pas dans la partie privée.

La section partagée contient toutes les variables qui ont été créées entre la racine de l'arbre de recherche et le point de choix duquel est issue la branche traitée par le processus.

La figure 2.6 illustre les différentes sections :

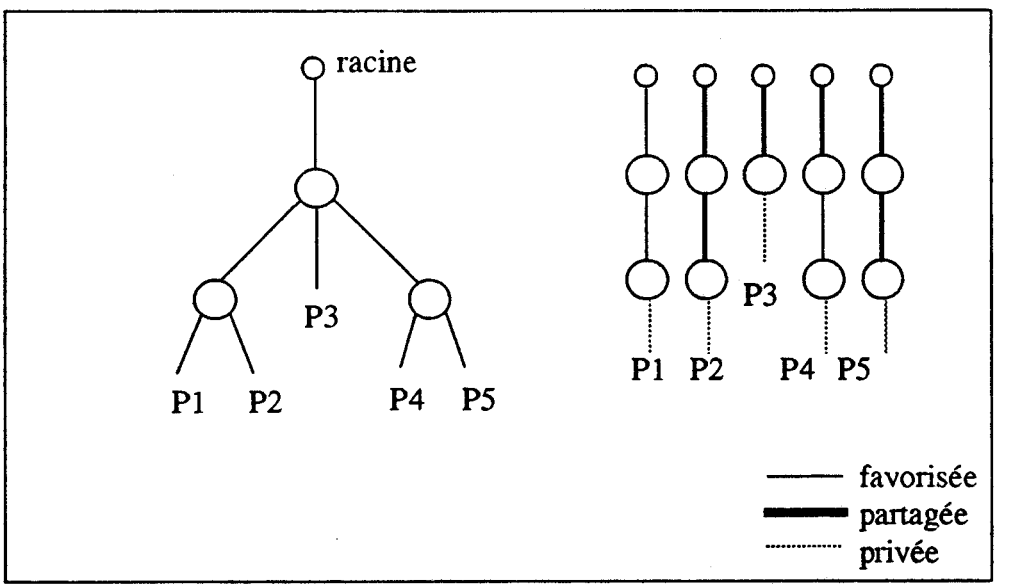



figure 2.6 : Classification des variables dans ANL-WAM

Suivant la section dans laquelle se trouve la cellule d'une variable, les opérations de liaison et de déréférencement qui lui sont appliquées utilisent le mode de liaison superficielle ou le mode de liaison profonde.

Chaque cellule de variable contient un bit dit "bit de faveur". La liaison d'une variable se trouvant dans la section privée ou favorisée est effectuée en utilisant le mode de liaison superficielle en positionnant le bit de faveur à zéro pour la section privée et à un pour la section favorisée.

Un processus liant une variable dans la section partagée utilise le mode de liaison profonde en stockant le couple (variable, valeur) dans une table de hachage dont les entrées peuvent contenir une liste de liaisons profondes.

Lors du déréférencement, si la variable superficielle de la variable est une liaison privée, la valeur de cette liaison doit être prise comme résultat. Dans le cas contraire, si cette liaison superficielle est favorisée (i.e le bit de faveur est positionné) et si la cellule de variable se trouve dans la section favorisée, la valeur de liaison est valide. Dans les autres cas, que ce soit une liaison favorisée dans la zone partagée ou une variable sans liaison superficielle, il faut chercher si une liaison profonde n'aurait pas été enregistrée. Cette recherche est similaire à celle utilisée dans le modèle PEPsys.

### 2.3.3.5 - Le modèle SRI :

Dans le modèle SRI, chaque processus (worker) possède, en plus des données habituelles (piles Wam), un tableau de liaisons (*Processor Binding Array*) où sont mémorisées toutes les liaisons conditionnelles. Une variable liée de façon conditionnelle contient un index du tableau de liaisons. L'index se trouvant dans la cellule d'une telle variable est utilisé, lors d'une liaison, pour stocker la valeur de liaison dans le tableau.

Les processus qui explorent l'arbre de recherche défont les liaisons conditionnelles enregistrées dans leurs tableaux de liaisons, lorsqu'ils vont vers la racine de l'arbre de recherche et les réinstallent lorsqu'ils descendent vers les feuilles. La réinstallation consiste à recopier le tableau de liaison et la pile de restauration de la branche de l'arbre de recherche sur laquelle une alternative a été trouvée. La pile de restauration contient non seulement les noms de variables liées conditionnellement mais aussi leurs valeurs de liaison pour faciliter la recopie.

Dans la figure 2.7, les processus parallèles *P1* et *P2* lient conditionnellement la variable X respectivement à *b* et *c* (la liaison *Y/a* est une liaison universelle). Lors de la déréférenciation de la variable *X*, l'index *i* contenu dans la cellule qui représente la variable *X* est utilisé comme entrée du tableau de liaisons du processus effectuant le déréférencement.

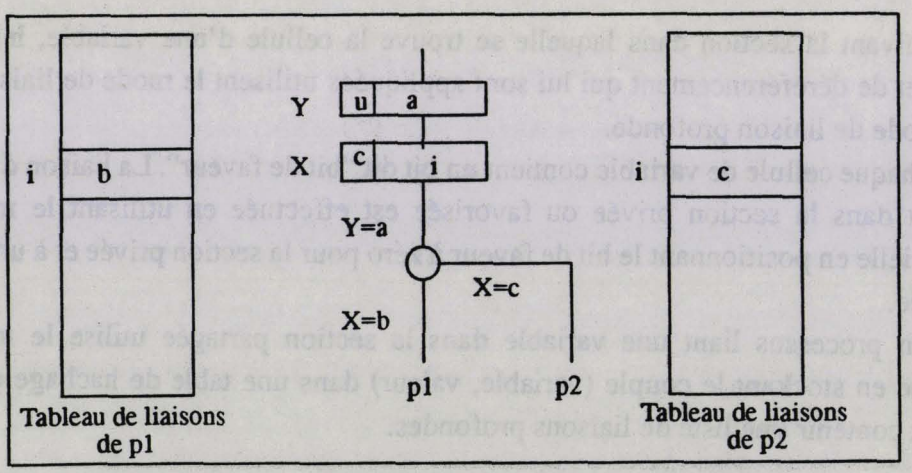



figure 2.7 : Mécanisme de liaison dans le modèle SRI

Le temps d'accès aux variables est constant mais le gros inconvénient de ce modèle est le coût de l'installation d'une tâche parallèle. Afin de diminuer la granularité du parallélisme, une heuristique a été utilisée. Elle consiste à attribuer à un processeur (worker) libre une nouvelle tâche située à proximité de la racine de l'arbre de recherche. Ce dernier est divisé en deux parties dites publique et privée. La partie publique contient tous les noeuds dont sont issues les branches traitées en parallèle. Un processeur qui travaille dans une partie privée explore entièrement celle-ci en utilisant le retour arrière. Les déperditions apparaissent donc seulement dans la partie publique.

Le système Prolog parallèle Aurora est une implantation du modèle SRI, qui a été réalisée dans le cadre du groupe de travail Gigalips. Les performances du système Aurora en font l'une des plus efficaces implantations du parallélisme OU existant actuellement.

### 2.3.3.6 - Versions-Vectors :

La technique utilisée dans le modèle Versions-Vectors pour gérer les liaisons multiples est une variante du modèle SRI. Les tableaux de liaisons sont remplacés par l'utilisation de vecteurs (figure 2.8).

Une variable liée conditionnellement est liée à un vecteur dont la taille est le nombre de processeurs, et dont chaque cellule peut contenir une valeur de liaison.

La recopie du tableau de liaisons dans le modèle Versions-Vectors se traduit par la mise à jour des vecteurs des liaisons effectuées dans la branche sur laquelle une alternative est trouvée. Lors de l'installation d'une branche parallèle, chaque liaison conditionnelle effectuée antérieurement à la création du noeud dont est issue cette branche, est installée dans la cellule correspondant au processeur qui est chargé de traiter la branche.

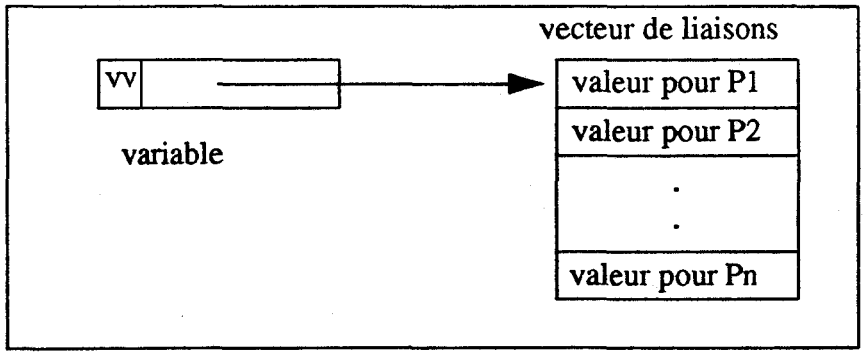



figure 2.8 : Mécanisme de liaison dans le modèle Versions-Vectors

Les caractéristiques de ce modèle semblent comparables à celles du modèle SRI. Cependant, il faut noter que cette technique exige une synchronisation à la création des vecteurs. En effet, chaque vecteur est créé dynamiquement par le processus qui lie conditionnellement la variable pour la première fois, ce qui nécessite la synchronisation des processus liant simultanément une même variable non liée auparavant.

# 3 - Conclusion :

De très nombreuses recherches ont été menées depuis le début des années 80 dans le domaine de la programmation logique parallèle. Deux principales approches peuvent être distinguées : les langages gardés ayant pour but la programmation système, et les langages logiques parallèles proches de Prolog, qui visent à implanter efficacement les applications symboliques.

Les langages gardés exploitent le parallélisme de flot, et fournissent des mécanismes explicites de synchronisation inter-processus. L'exécution d'un prédicat ne fournit qu'une seule solution. Cette restriction a pour but d'éviter l'explosion combinatoire de processus provoqués par un parallélisme OU. Par ailleurs, le contrôle de parallélisme ET est explicité par le programmeur en spécifiant le mode d'accès aux variables.

Des implantations efficaces de ces langages ont été réalisées sur mono et multi-processeurs. La machine PIM (Parallel Inference Machine) de l'ICOT est une machine parallèle spécialisée pour l'exécution du langage gardé GHC.

Malgré les résultats obtenus, les langages gardés ne peuvent remplacer complètement Prolog, à cause de l'abandon du non-déterminisme et l'introduction explicite du contrôle du parallélisme qui détruit ainsi la simplicité de la programmation et la sémantique déclarative de la programmation logique.

Les recherches basées sur la deuxième approche, dont l'étude est plus récente que celle des systèmes gardés, visent à fournir des systèmes efficaces permettant l'implantation des programmes écrits en Prolog. Seules les deux formes de parallélisme ET et OU existant dans les programmes logiques sont utilisées comme sources de parallélisme.

L'utilisation du parallélisme ET est plus délicate, en raison de la gestion nécessaire des variables partagées par plusieurs sous-buts d'une même clause. C'est pourquoi la majorité des modèles proposés utilisent la parallélisme OU.

L'implantation multiséquentielle du parallélisme OU présente un grain plus gros et de meilleurs résultats. Cependant, les recherches sur le parallélisme OU ne connaissent pas encore toutes les limites de l'efficacité de ce type de parallélisme. Tous les modèles proposés présentent un trait commun, à savoir le compromis entre le coût de l'installation des tâches et celui du mécanisme de liaison utilisé pour gérer les liaisons multiples.

Dans les modèles basés sur la recopie de l'environnement (ou recopie partielle dans SRI), le déréférencement (accès aux valeurs des liaisons) est facile à mettre en oeuvre mais toujours au prix d'un taux important de déperditions, dû à l'installation des tâches, et qui pénalise les performances du système.

Dualement, dans les modèles utilisant le partage de l'environnement, l'installation des tâches n'introduit que peu de déperditions, mais l'opération de déréférencement est plus coûteuse et plus complexe à réaliser.

Partant de cette constatation, le modèle OU parallèle multi-séquentiel que nous proposons répond à deux objectifs.

Le premier est de démontrer que la recopie n'est pas toujours nécessaire pour éviter un coût d'accès aux variables qui risque de pénaliser les performances du système. D'une part, l'accès aux valeurs de liaisons se fait de manière très simple et ne présente pas beaucoup de déperditions, et d'autre part, l'installation des tâches parallèles ne nécessite aucune recopie d'environnement.

Le deuxième objectif du modèle est l'implantation des effets de bord, qui jusqu'à présent n'a connu que peu d'efforts, les solutions proposées jusqu'ici n'étant que partielles. Nous montrerons qu'une implantation efficace des effets de bord dépend de la stratégie de recherche de travail (*scheduler*), adoptée par les processus libres pour explorer l'arbre de recherche.

# -=- Chapitre III -=-

*Présentation*
*du modèle de calcul*
*OU-parallèle*

***Pourquoi le parallélisme OU ?***

Le parallélisme OU a fait l'objet de nombreuses recherches dans le domaine de la programmation logique parallèle. Parmi les principaux avantages qui expliquent cette attirance pour le parallélisme OU, l'on trouve les suivants :

- Il est possible d'exploiter le parallélisme OU sans faire appel à des annotations de la part du programmeur, ou à une analyse à la compilation pour détecter le parallélisme.

- L'utilisation du parallélisme OU préserve la puissance de la programmation logique et en particulier elle permet à Prolog de générer toutes les solutions à un but donné.

- Le parallélisme OU apparait à une grande échelle dans beaucoup d'applications, en particulier dans : les systèmes experts, les problèmes de recherche (ex : "générer et tester"), la démonstration des théorèmes, la compilation, le traitement du langage naturel, et la recherche en bases des données.

***Pourquoi la stratégie multi-séquentiele ?***

Comme nous l'avons vu dans le chapitre précédent, les modèles d'exécution parallèle proposés pour Prolog reposent sur deux approches différentes : L'une a pour objectif de maximiser le potentiel de parallélisme en élaborant des modèles à fin grain de parallélisme, l'autre a pour objetctif d'exploiter efficacement un potentiel de parallélisme.

Dans la première, où l'on vise un parallélisme massif, on cherche à produire le maximum d'entités élémentaires d'exécution ou processus lors du parcours ET/OU et l'on procède par construction dynamique d'un réseau de processus communiquants. Il s'ensuit que le placement de ce réseau dans la machine physique reste le problème majeur à résoudre.

Dans la seconde, on ne s'intéresse qu'à trouver des processus " réellement" exécutables en parallèle et leur activation est contrainte par la disponibilité des ressources de calcul. Le degré de parallélisme, c'est à dire le nombre de processus actifs, à un instant donné, est borné. L'exécution d'un programme Prolog est alors vue comme un mixage d'exécution séquentielle et d'exécution parallèle, d'où le nom de "multi-séquentiels" pour les modéles basées sur cette approche. De ce fait, la mise en oeuvre de ces modèles peut bénéficier, d'une part des techniques efficaces utilisées dans les implantations séquentielles, et de l'autre part de l'existence sur le marché des machines multiprocesseurs à mémoire commune. Actuellement, de telles machines ne présentent qu'un degré de parallélisme limité (quelque dizaines de processeurs) du fait du goulot d'étranglemet constitué par l'accés à la mémoire. A leur avantage, elles permettent, grâce à cette mémoire commune), une grande facilité de programmation (partage des données et code), et d'implantation des algorithmes parallèles qui sont des extensions simples da langages algorithmiques séquentiels. Cette facilité se traduit par l'existence d'environnements de programmtion et de systèmes "classiques" (tels UNIX).

Si plusieurs projets ambitieux ont mis à jour des modèles d'exécution visant un parallélisme massif, de réelles implantations n'ont pas encore vu le jour. Certains modèles sont restés qu'au stade de simulation. A ce jour, les implantations des modèles multi-séquentiels sur des multiprocesseurs à mémoire commune, sont les seules à montrer un accroissement réel de performances dû au parallélisme. Les tableaux suivants montrent certains résultats obtenus dans les systèmes Aurora [Ali 91], Muse [Ali 91] et PEPSys [ Cha 89b].

| Programmes testés | 1 processeur | 10 processeurs | 20 proceseurs | 30 processeurs | 32 processeurs | SICtus0.6 TC2000 |
|---|---|---|---|---|---|---|
| semigroup | 1699.78 | 169.62 | 87.03 | 58.90 | 54.68 | 990.89 |
| 11-reines | 369.14 | 36.92 | 18.54 | 12.47 | 11.70 | 190.87 |
| 8-reines | 2.85 | 0.33 | 0.21 | 0.22 | 0.23 | 1.48 |
| tina | 6.33 | 0.83 | 0.65 | 0.88 | 1.16 | 3.07 |
| salt-mustard | 1.47 | 0.19 | 0.12 | 0.12 | 0.14 | 0.44 |
| parse2*20 | 2.46 | 1.30 | 1.50 | 1.74 | 1.87 | 1.43 |
| house*20 | 1.55 | 0.79 | 1.12 | 1.99 | 2.63 | 0.94 |
| farmer*100 | 1.14 | 1.80 | 2.14 | 2.33 | 2.37 | 0.66 |

table 3.1 : Performances (en secondes) de Aurora sur BBN Butterfly TC2000

| Programmes testés | 1 processeur | 10 processeurs | 30 proceseurs | 50 processeurs | 64 processeurs | SICtus0.6 TC2000 |
|---|---|---|---|---|---|---|
| semigroup | 1178.63 | 118.56 | 60.06 | 40.72 | 38.35 | 990.89 |
| 11-reines | 225.23 | 22.78 | 11.58 | 7.88 | 7.41 | 190.87 |
| 8-reines | 1.79 | 0.21 | 0.41 | 0.13 | 0.13 | 1.48 |
| tina | 4.28 | 0.58 | 0.39 | 0.34 | 0.34 | 3.07 |
| salt-mustard | 0.71 | 0.10 | 0.08 | 0.09 | 0.10 | 0.44 |
| parse2*20 | 1.82 | 0.80 | 0.84 | 0.85 | 0.87 | 1.43 |
| house*20 | 0.98 | 0.58 | 0.61 | 0.63 | 0.63 | 0.94 |
| farmer*100 | 0.83 | 1.01 | 1.03 | 1.07 | 1.05 | 0.66 |

table 3.2 : Performances (en secondes) de Muse sur BBN Butterfly TC2000

| Programmes testés | PEPSys(1) | PEPSys(4) | PEPSys(8) | Quintus Sun 3/50 |
|---|---|---|---|---|
| Hamiltonien | 274 | 68.8 | 38.3 | 21.1 |
| Mandel | 42.3 | 12.1 | 6.4 | 13.2 |
| 8-reines | 67 | 17.9 | 9.2 | 3.5 |
| salt-mustard | 7.4 | 2.7 | 1.5 | 0.5 |
| Tina | 107.9 | 29.7 | 15.9 | 9.5 |

table 3.3 : Performances (en seconde) de l'implantation de PEPSys sur MX500

# 1 - Caractéristiques générales du modèle de calcul :

Le modèle OU parallèle qui est ici présenté, a profité de l'état de l'art des travaux réalisés dans la programmation logique *OU-parallèle*. Il propose des solutions aux problèmes connus par les modèles qui sont basés sur le partage de l'environnement (PEPsys, ANL-WAM) et qui utilisent des mécanismes de liaison introduisant beaucoup de déperditions lors de l'accès aux valeurs de liaisons. Les solutions apportées permettent de réduire ces déperditions en évitant toute recopie d'environnement. La nécessité de contrôler la granularité de l'exécution, le besoin de gérer efficacement les effets de bords, expliquent cette volonté d'éviter la recopie de l'environnement.

**Contrôle de la granularité :**

Les modèles OU-parallèles multi-séquentiels offrent une grosse granularité par rapport aux autres modèles parallèles. Mais envisager que toutes les alternatives du programme sont des sources potentielles de parallélisme peut rendre, dans certains cas, ce dernier inefficace.

La détection des points de choix offrant un parallélisme significatif n'est pas facile à réaliser.

Le coût de l'utilisation du parallélisme OU, lors de l'invocation du but *r(X)* (dans le programme 1) dans la clause (1), est certainement supérieur au gain que l'on peut en espérer.

Par contre, l'exécution en parallèle des deux faits définissant le prédicat *r*, peut présenter un gain significatif lors de l'exécution du corps de la clause (2). En effet, chaque processus qui exécute un fait avec succès, poursuit la résolution (stratégie multi-séquentielle) avec le but *grostache* (celui-ci est supposé nécessiter beaucoup de calcul).

```
(1) p(X) :- grostache, r(X).
(2) q(X) :- r(X),grostache.
(3) grostache:- . . .
(4) r(b).
(5) r(a).
```

figure 3.1 : Programme 1

Certains modèles recommandent une discipline de programmation, basée sur des annotations qui permettent le contrôle du parallélisme. Le programmeur a la charge d'indiquer les prédicats dont les clauses peuvent être exécutées en parallèle; les clauses définissant un prédicat non qualifié de *parallèle*, sont exécutées en utilisant le retour-arrière.

Cette technique est indispensable lorsque l'ordre des clauses d'un même paquet a une signification pour le programmeur, ou lorsque les prédicats à effets de bord sont utilisés dans le programme. Elle est insuffisante pour garantir l'efficacité du parallélisme. En effet, l'invocation du prédicat *r* (figure 3.1) implique un parallélisme significatif lors de la résolution de q, et un parallélisme à très fin grain lors de l'exécution du corps de la clause associée à p; il est difficile de ce fait de qualifier le prédicat *r* de *parallèle* ou de *séquentiel.*

En définitive, il n'est pas facile de déterminer la taille d'une branche de l'arbre de recherche qu'un processus parcourt. C'est dans un souci de réduire les déperditions dans les programmes à fin grain de parallélisme, que nous avons voulu éviter la recopie. Si le partage de l'environnement est utilisé, le coût de l'utilisation du parallélisme OU dans la clause (1) est certainement inférieur à celui résultant de l'utilisation du parallélisme dans une approche basée sur la recopie, puisqu'alors, l'exécution en parallèle des deux alternatives associées au but *r(X)* dans (1) nécessite la recopie de l'historique de la résolution du but *grostache.*

**Gestion des effets de bord :**

L'avantage de l'utilisation du partage de l'environnement (sans aucune recopie) est qu'elle offre un coût très réduit d'installation des tâches parallèles. Ce coût est constant quelque soit la position, dans l'arbre de recherche, du point de choix donnant lieu au parallélisme. Ainsi, aucune règle de choix n'est imposée aux processus libres cherchant du travail, contrairement aux modèles basés sur la recopie. Dans ces modèles, la règle de choix consiste à concentrer la recherche de travail par les processus oisifs, à proximité de la racine de l'arbre de recherche, afin de diminuer la taille des données à recopier.

Le coût constant d'installation des tâches parallèles nous a permis d'étudier plusieurs règles de choix afin de gérer efficacement les effets de bord. Nous montrerons par la suite quelles sont les règles de choix que nous avons étudiées et comment elles peuvent résoudre les problèmes de l'utilisation des effets de bord dans une exécution OU-parallèle.

Dans les deux paragraphes suivants, nous présentons les deux principaux traits du modèle de calcul, à savoir le contrôle du parallélisme et la gestion des liaisons multiples. Dans le premier, nous montrons comment le parallélisme OU est mis en oeuvre. Dans le second, nous décrivons un mécanisme de liaison qui peut être implanté de trois façons différentes, et indiquons les avantages et les inconvénients de chacune de ces méthodes.

## 1.1 - Contrôle :

Les principes du modèle de calcul sont d'une part, l'utilisation de la stratégie multi-séquentielle, et d'autre part, la limitation du nombre de liaisons d'une même variable au nombre de processeurs. Nous verrons que cette limitation ne réduit pas l'efficacité du parallélisme, mais qu'elle permet au contraire, de borner le temps d'accès aux valeurs de liaisons.

Comme nous l'avons vu dans la section 2.2 du second chapitre, la stratégie multi-séquentielle a pour but de contrôler la création des processus en fonction des ressources disponibles. Le nombre de processus actifs, à un instant donné, est toujours inférieur ou égal au nombre de processeurs. Chaque processus actif parcourt une branche de l'arbre de recherche.

De ce fait, un processus actif ne peut lier une variable plus d'une fois. Il semblerait que le nombre de liaisons d'une même variable ne puisse excéder le nombre de processus actifs et par conséquent le nombre de processeurs. Il n'en est rien. Si le nombre de processus actifs est toujours borné, le nombre de processus créés peut dépasser le nombre de processeurs.

En effet, lorsque des mécanismes de suspension sont utilisés, certains processus sont amenés à devenir inactifs et les processeurs qui les exécutent sont alloués à d'autres tâches pour explorer de nouvelles branches; ceci ne permet pas de garantir l'existence d'un nombre limité de liaisons d'une même variable.

## 1.2 - Mise en oeuvre du parallélisme OU :

### 1.2.1 - Un processus est une instance de la WAM :

Un processus prenant en charge une branche de l'arbre de recherche poursuit la résolution de celle-ci selon la stratégie *en profondeur d'abord* et indépendamment des autres processus. Chaque processus actif gère les trois piles utilisées dans le modèle séquentiel (pile locale, de restauration (ou traînée) et pile globale). L'exécution parallèle peut être représentée par un arbre de piles, qui contient les informations de contrôle et les données manipulées et peut être calqué sur l'arbre de recherche.

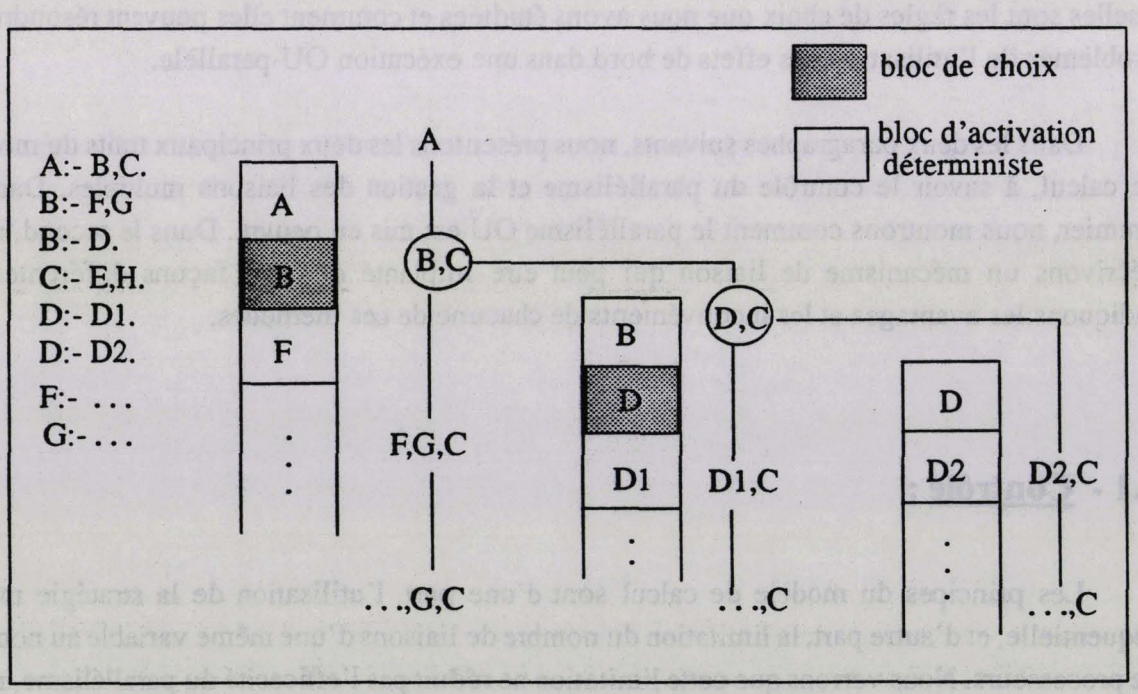



figure 3.2 : Arbre de piles calqué sur l'arbre de recherche exploré par 3 processus.

### 1.2.2 - Une répartition de la charge par vol d'alternatives :

Un processus inactif acquiert du travail auprès d'un processus actif possédant une ou plusieurs alternatives en attente. Dans les systèmes multi-séquentiels, l'accès au travail disponible par un processus oisif, peut être réalisé de trois manières différentes :

(1) La première consiste en un contrôle centralisé : un seul processus de contrôle collecte toutes les alternatives en attente chez les processus actifs, et distribue celles-ci sur les processus cherchant du travail. Son inconvénient est le goulot d'étranglement entraîné par la communication de l'ensemble des processus oisifs avec le processus de contrôle.

(2) La deuxième possibilité qu'offre la stratégie multi-séquentielle pour répartir la charge entre processus, est de laisser la responsabilité au processus actif de distribuer lui-même le travail qui peut être réalisé en parallèle.

(3) La troisième façon de faire coopérer les processus est celle que nous avons choisie. Elle consiste à donner la permission à un processus oisif de voler une alternative en attente chez un processus actif. Contrairement à la deuxième solution, le coût de la recherche et du choix de la nouvelle branche à explorer est supporté par la ressource disponible et non par la ressource active comme dans la solution précédente. Si aucune ressource n'est disponible pour traiter une alternative en attente, celle-ci est traitée, lors du retour-arrière, par le processus l'ayant créée. Seules, donc, les déperditions introduites par le partage des données (exclusion mutuelle, gestion des liaisons multiples,...), sont à supporter par un processus actif.

La seule contrainte à respecter pour mettre en oeuvre une répartition des tâches par vol d'alternatives, est de rendre visibles tous les points de choix créés par les processus actifs, ce qui nécessite l'utilisation d'une mémoire accessible par l'ensemble des processus. Cette mémoire permet d'exhiber tous les points de choix qui peuvent être traités en parallèle.

Un technique similaire a été utilisée dans le modèle RAP-WAM [Her 86] qui exploite le parallélisme ET restreint. Lorsque l'indépendance de sous-bus est vérifiée, des structures de données correspondant à ces sous-buts sont rangées sur la pile de buts (goal-stack), à l'exception du premier qui est résolu par le processeur produisant les sous-buts. Tout processeur à la recherche de travail peut *voler* un but sur cette pile.

Le modèle introduit pour chaque processus une structure de données supplémentaire, en plus des trois piles déjà citées. Cette structure, appelée pile de choix, permet à chaque processus actif de publier les points de choix pouvant susciter du parallélisme.

Lorsqu'un processus actif crée un point de choix, un pointeur vers celui-ci est empilé sur la pile de choix (figure 3.3). Ce pointeur facilite l'accès à toutes les informations nécessaires à un processus oisif désirant prendre en charge l'une des alternatives du point de choix.

Un processus oisif peut de ce fait explorer toutes les piles de choix des processus actifs à la recherche d'une tâche à prendre en charge.

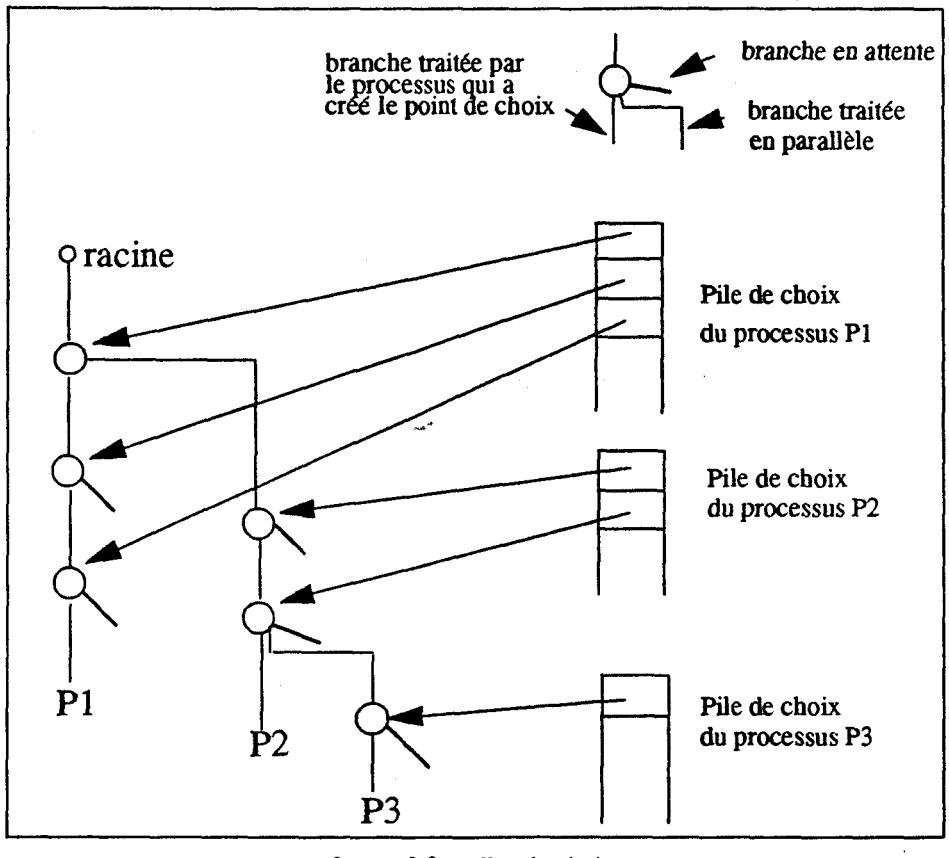



figure 3.3 : Piles de choix

En définitive, ce n'est qu'au moment où un processus trouve un point de choix possédant une alternative non encore traitée, que le parallélisme a lieu.

La pile locale d'un processus actif peut jouer le rôle de la pile de choix introduite précédemment, étant donné qu'elle mémorise tous les point de choix créés. L'accès à tous les point de choix peut être effectué en connaissant le dernier point de choix créé et en utilisant le champ BL de chaque bloc de choix, champ qui, comme nous l'avons vu au paragraphe 2.2 du premier chapitre, permet de chaîner tous les points de choix, et ainsi, d'accéder aux point de choix créés antérieurement.

L'utilisation de la pile de choix va de plus permettre à un processus de décider s'il publie ou non un point de choix. Un point de choix non publié est inaccessible aux processus oisifs, et est donc traité séquentiellement par le processus qui l'a créé.

Cette technique permet d'implanter des mécanismes qui contrôlent l'efficacité du parallélisme, comme l'utilisation des annotations, qui désignent explicitement les prédicats dont la résolution peut être ou non parallèle, ou l'utilisation des techniques de seuil (comme dans la BC-machine) permettant à un processus atteignant un certain seuil de ne plus (ou de commencer à) publier ses points de choix. Ce seuil peut être un nombre de point de choix créés qui serait fixé au départ.

## 1.3 - Contrôle du retour-arrière :

Dans la stratégie multi-séquentielle, le parallélisme OU est toujours combiné avec le retour-arrière afin d'explorer l'arbre de recherche d'un programme. Le retour-arrière finit toujours par se produire, qu'il soit consécutif à l'échec d'une unification, ou à la production d'une solution du but initial. Toutes les branches de l'arbre de recherche sont donc parcourues, soit parallèlement si les ressources sont suffisantes, soit séquentiellement par le mécanisme de retour-arrière.

La mise en oeuvre du retour-arrière pose un problème dans les systèmes multi-séquentiels basés sur le partage de l'environnement. Dans un système Prolog séquentiel, le retour-arrière s'effectue toujours au dernier point choix créé qui contient nécessairement une alternative en attente. En effet, lorsque la dernière alternative détenue par un point de choix est considérée, l'espace alloué à ce point de choix est immédiatement récupéré.

Dans une exécution parallèle, trois situations distinctes peuvent se produire lors du retour-arrière effectué par un processus :

- il reste encore des alternatives non traitées : le retour-arrière s'effectue donc comme dans la stratégie séquentielle;

- toutes les alternatives ont été traitées, soit en parallèle soit en séquentiel : dans ce cas, le point de choix est amené à disparaître, et le retour-arrière s'effectue sur le plus jeune point de choix créé sur la branche traitée;

- toutes les alternatives sont épuisées, mais certains des processus qui ont pris en charge des branches issues du point de choix n'ont pas encore fini la résolution de celles-ci : le retour-arrière ne peut alors être déplacé à un point de choix précédent, car une récupération prématurée de l'espace des piles entraînerait la destruction de certaines données encore utiles aux processus qui parcourent des branches issues du point de choix considéré.

## Exemple :

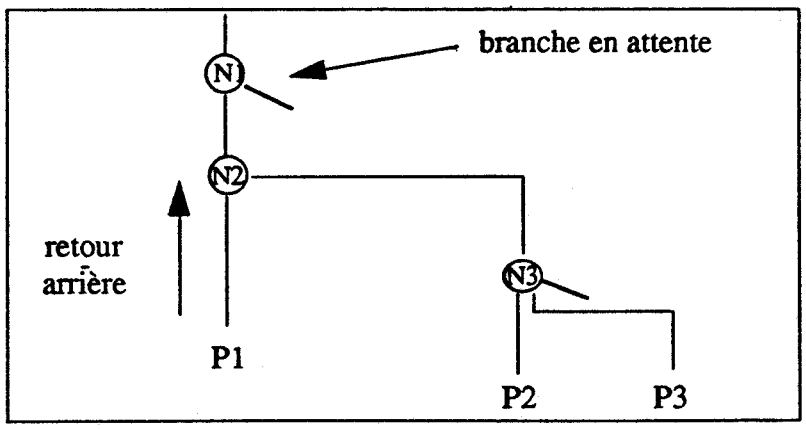



figure 3.4 : Contrôle du retour-arrière

Considérons la figure 3.4.

Lorsque le processus *P1* effectue le retour-arrière vers le point de choix *N2*, il trouve toutes les alternatives épuisées. Le retour-arrière vers le noeud N*1* ne peut cependant avoir lieu, car il risque d'entraîner la destruction des données créées entre les noeuds *N1* et *N2*, qui peuvent être sollicitées par les processus *P2* et *P3*.

Dans ce cas, le retour-arrière par *P1* vers *N1* n'est effectif, que lorsque la branche parallèle issue de *N2 a* été entièrement explorée *: P1* est donc suspendu et se met en attente. Par contre, le processus *P2* est autorisé à effectuer le retour arrière pour traiter l'alternative en attente dans *N3*.

Nous avons considéré uniquement le retour-arrière par un processus vers un point de choix qu'il a lui même créé. Le retour-arrière d'un processus vers un point de choix qui ne lui appartient pas est considéré comme une terminaison de tâche. Dans notre exemple, le processus devient alors libre et peut prendre en charge n'importe quelle branche. Lorsque le processus *P3* termine l'exploration de la deuxième branche issue de *N3*, il peut choisir de traiter la troisième branche issue de *N3* ou l'alternative encore en attente au point de choix *N1*.

Afin de réduire les déperditions introduites par l'éventuelle synchronisation lors du retour-arrière, la ressource de calcul allouée au processus suspendu peut être récupérée pour réaliser une autre tâche. Dans ce cas, le nombre de processus créés (nombre de branches parcourues) peut croître indéfiniment, entraînant, par là même, la non limitation du nombre de liaisons d'une même variable.

Pour répondre aux objectifs initiaux, nous avons adopté la solution utilisée dans le modèle *PEPsys*, qui consiste à allouer au processus suspendu une nouvelle tâche, cette nouvelle tâche étant située dans le sous-arbre dont la racine est le point de choix provoquant le besoin de synchronisation. Dans l'exemple de la figure 3.4, le processus *P1* ne pouvant effectuer le retour-arrière vers le noeud *N1*, peut aller aider le processus *P2 en* résolvant l'alternative mise en attente par celui-ci.

Cette technique d'allocation de nouvelles tâches aux processus suspendus suite à un retour-arrière impossible à effectuer, permet d'éviter de retarder la réactivation du processus suspendu. En effet, il est évident que, lorsque la tâche attendue est accomplie, une éventuelle tâche l'ayant aidée est, elle aussi, terminée, et le retour-arrière peut s'effectuer, sans réactivation du processus suspendu.

Avant de montrer comment les spécifications du modèle permettent de borner le nombre de liaisons d'une même variable, nous définissons d'abord la notion de *processeur virtuel.*

Un **processeur virtuel** est une instance de la machine abstraite de Warren augmentée de la pile des choix introduite précédemment. Plusieurs processus peuvent exister sur un même processeur virtuel, mais seul le processus le plus jeune est actif, les autres sont suspendus. La réactivation d'un processus suspendu n'est possible que lorsque les processus les plus jeunes sont terminés.

De ce fait, les processus peuvent se partager les piles du processeur virtuel. Un processus est lié à un seul processeur virtuel : il est créé lorsqu'une nouvelle branche parallèle est prise en charge, et est détruit après l'exploration complète de la branche.

Par la suite, nous utiliserons "processeur" comme abréviation de "processeur virtuel", et "tâche" pour désigner un "processus".

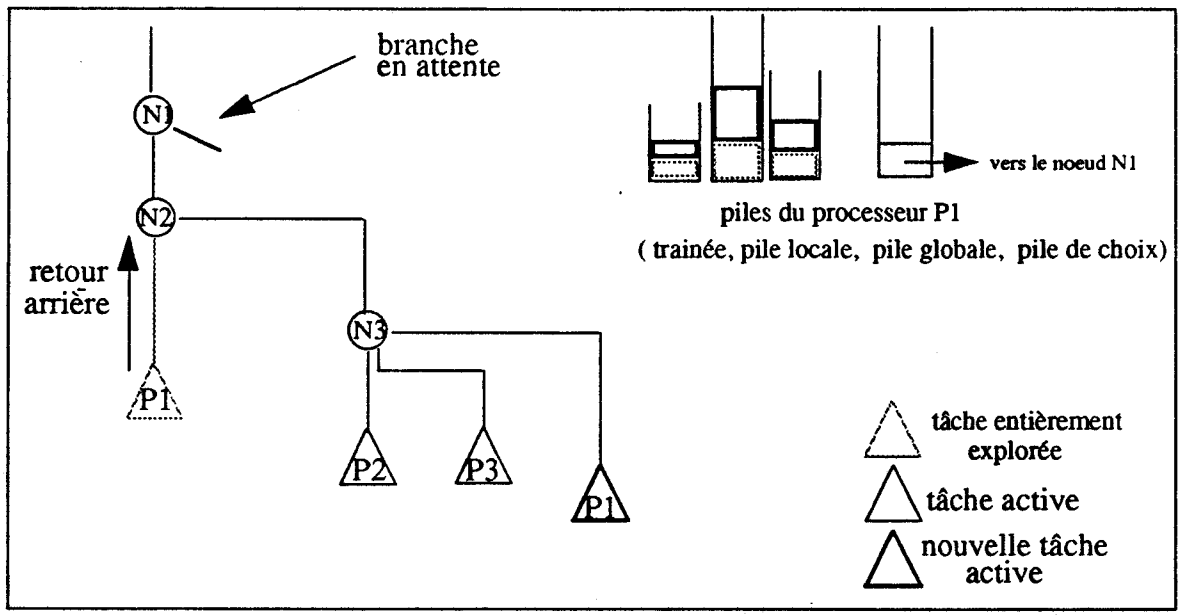



figure 3.5 : Changement de tâche par un processeur

La figure 3.5 montre l'état des piles du processeur *P1* après qu'il ait pris en charge une nouvelle tâche pour aider le processeur *P2*.

L'espace alloué sur les différentes piles, après la création du point de choix *N2*, est récupéré pour être réutilisé lors de l'exécution de la nouvelle tâche. La récupération de l'espace dans la pile de restauration s'effectue en mettant à jour le sommet de pile, après avoir défait toutes les liaisons conditionnelles qui sont effectuées par le processeur *P1* après la création du point de choix N2.

De ce fait, le nombre de liaisons d'une même variable, qui coexistent sur un même processeur à un instant donné, est au plus égal à un. Toutes les liaisons effectuées par les tâches suspendues sont valides pour le processus actif et ne peuvent être remises en cause par celui-ci.

Il faut noter qu'une tâche est toujours attachée à un processeur et un seul, et ne peut migrer. Sans cette hypothèse, des liaisons multiples risquent de coexister sur un même processeur.

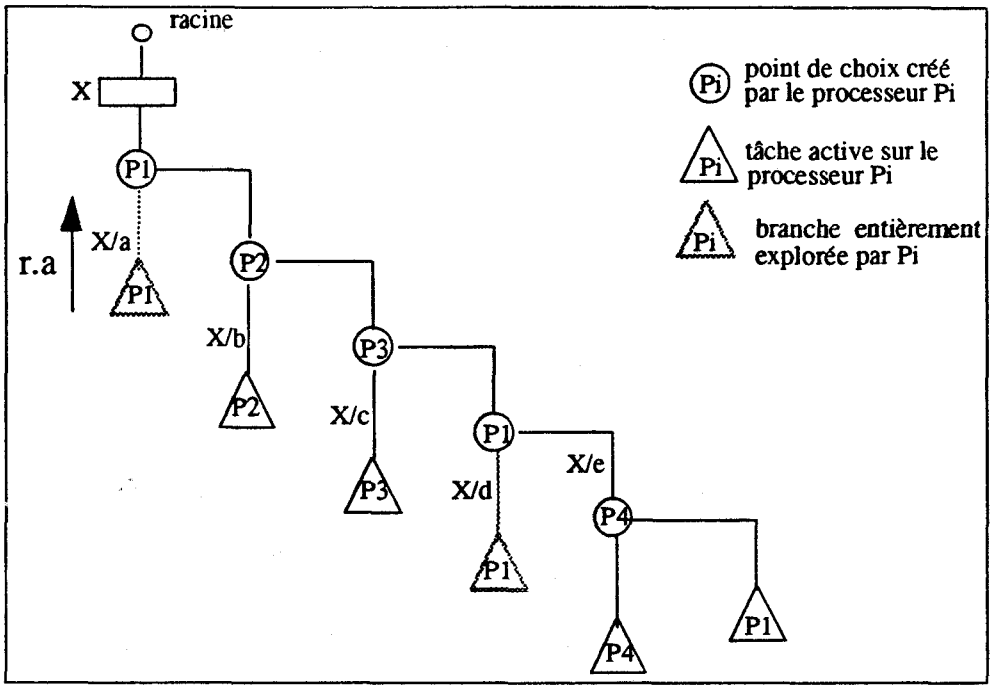



figure 3.6 : Contrôle du nombre de liaisons d'une variable par un processeur.

Dans la figure 3.6, le processeur *P1* commence la résolution et crée la variable *X*, celle-ci demeurant libre lorsque le premier point de choix est rencontré. Elle peut donc faire l'objet de liaisons multiples. La deuxième alternative créée par *P1* est prise en charge immédiatement par le processeur *P2*.

*P2* crée à son tour un point de choix qui donne lieu à une branche parallèle traitée par *P3*. Le processeur *P1*, après avoir lié la variable *X* (*X/a*), décide d'effectuer le retour-arrière. Or ceci est impossible car l'alternative traitée par *P2* n'est pas entièrement explorée.

*P1* s'engage à prendre une nouvelle tâche après avoir défait la première liaison conditionnelle *X/a* (celle-ci n'étant plus utile par la suite); *P1* peut alors lier une deuxième fois la variable *X* (X/d) lors de la tâche entreprise pour aider P3.

Cette deuxième liaison est défaite à son tour lorsque *P1* prend en charge l'autre alternative créée par *P4*. Au cours de cette tâche, le processeur *P1* ne peut lier une autre fois la variable *X*, puisqu'il partage la liaison (X/e) effectuée par *P4*.

C'est là qu'apparait le problème majeur dans le partage de l'environnement, à savoir, la détection par un processeur des liaisons valides qui sont effectuées par les autres processeurs.

## 2 - Gestion des liaisons multiples :

Nous avons vu dans le paragraphe II.3 les différentes solutions apportées au problème des liaisons multiples introduit par l'utilisation du parallélisme OU. Ces solutions sont basées sur deux approches différentes.

La première, qui consiste à dupliquer l'environnement partagé par plusieurs processus (la duplication étant réalisée par recopie ou par duplication du calcul), évite tout conflit de liaison.

Le principe de la seconde approche est de faire partager l'environnement par l'ensemble des processus, chacun des processus étant capable d'atteindre n'importe quelle cellule de variable. Seul un contrôle pour détecter les liaisons conditionnelles valides est nécessaire. Ce problème est résolu de deux façons différentes. La première consiste à recopier (SRI) ou installer (Version-Vectors) les liaisons conditionnelles valides lors de l'installation d'un processus. La deuxième façon utilise une technique de marquage des liaisons (PEPsys, ANL-WAM) qui permet à un processus de détecter les liaisons valides.

Nous rappelons sommairement les principes de ces approches dans l'exemple décrit ci-dessous :

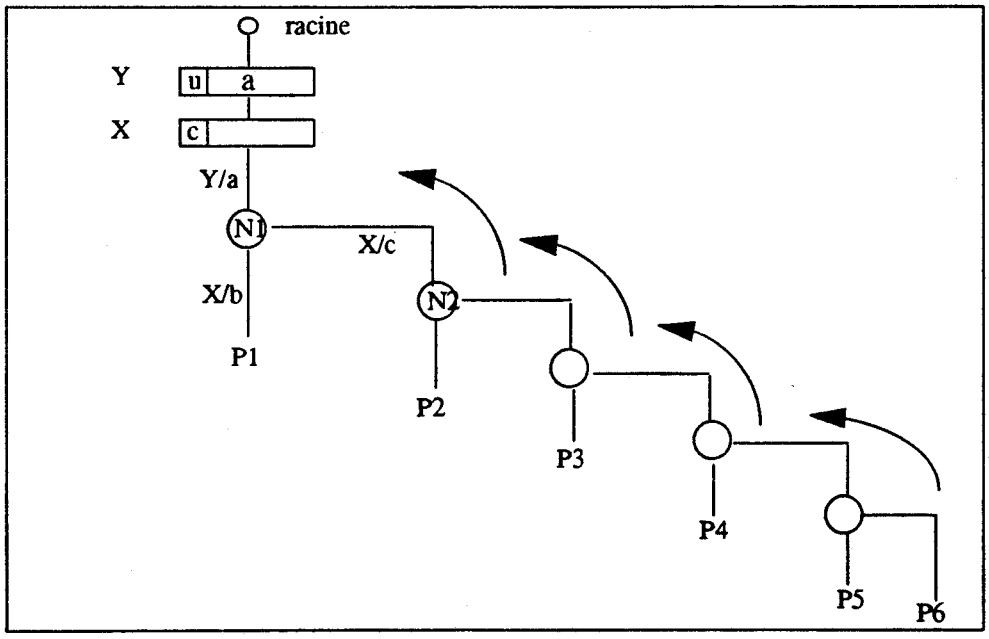



figure 3.7 : Méthodes de gestion des liaisons multiples

Le processus *P1*, commençant la résolution, crée deux variables *X* et *Y* (i.e il invoque une clause contenant X et Y). *P1* lie la variable *Y* avant de rencontrer le noeud *N1*; cette liaison est donc une liaison universelle. La deuxième alternative est prise en charge par le processus *P2*, qui lie conditionnellement la variable X (X/c) avant de créer le noeud *N2*. Cette liaison est valide pour tous les processus travaillant pour l'exploration du sous-arbre de racine *N2*.

Dans une approche basée sur la duplication de l'environnement, chaque processus possède une copie de l'état des deux cellules représentant les variables *X* et *Y*, avant la création du noeud *N1*.

Lorsque le partage de l'environnement est utilisé, les deux cellules existent en un seul exemplaire et sont accessibles par les différents processus. Le mécanisme de liaison profonde (voir paragraphe 2.3.3.1 du chapitre II) est utilisé pour représenter les liaisons multiples.

Lorsque les liaisons conditionnelles valides sont recopiées (SRI, Versions-Vectors), à chaque installation de tâche, la liaison *X/c* est copiée par chacun des processus qui la partagent, c'est à dire *P3, P4, P5* et *P6*. Mais lorsqu'une technique de marquage est utilisée (PEPsys), la liaison *X/c* est datée par le processus l'ayant effectuée, et est rangée dans une structures de données (généralement des tables de hachages) associée au processus. Cette date permettra à un processus de valider une telle liaison en comparant sa chronologie avec celle du noeud *N2*. Lorsque le processus *P6* décide de connaître la valeur de la variable X, il doit parcourir toute la chaîne des processus P5, P4, P3 et P2, indiquée dans la figure 3.7, à la recherche d'une liaison valide.

C'est dans ce parcours qu'apparait le gros défaut des modèles basés sur la technique de marquage.

La technique de liaisons que nous présenterons dans les paragraphes suivants est basée sur le partage de l'environnement et utilise une technique de marquage pour valider les liaisons conditionnelles. Contrairement aux modèles basés sur une telle approche, cette technique a permis d'une part de borner la chaîne des processus parcourue par un processus à la recherche d'une liaison valide, et d'autre part, d'utiliser des structures de données facilitant l'accès aux liaisons conditionnelles.

Dans un mécanisme de liaison basé sur le partage de l'environnement qui évite toute recopie, deux problèmes doivent être résolus. Le premier consiste à trouver une technique qui permet à un processus de valider des liaisons conditionnelles. Le second problème concerne la façon de représenter les liaisons multiples afin de permettre leur création et leurs accès.

## 2.1 - Technique de validation :

Pour valider une liaison conditionnelle, chaque processeur *Pi* dispose d'un jeu de clés [K1, . . .,Kn] (n étant le nombre de processeurs). Une liaison conditionnelle créée par un processeur Pj, est valide pour le processeur Pi si la clé Kj de ce dernier permet de valider une telle liaison. Une clé Kj représente pour un processeur *Pi*, le nombre de point de choix créés par le processeur *Pj*, qui peuvent être atteints par *Pi*. En termes de noeud-OUs, Kj représente le nombre de noeuds créés par *Pj*, ancêtres du noeud dont est issue la branche active sur le processeur *Pi*. L'exemple de la figure 3.8 illustre cette définition.

**Exemple :**

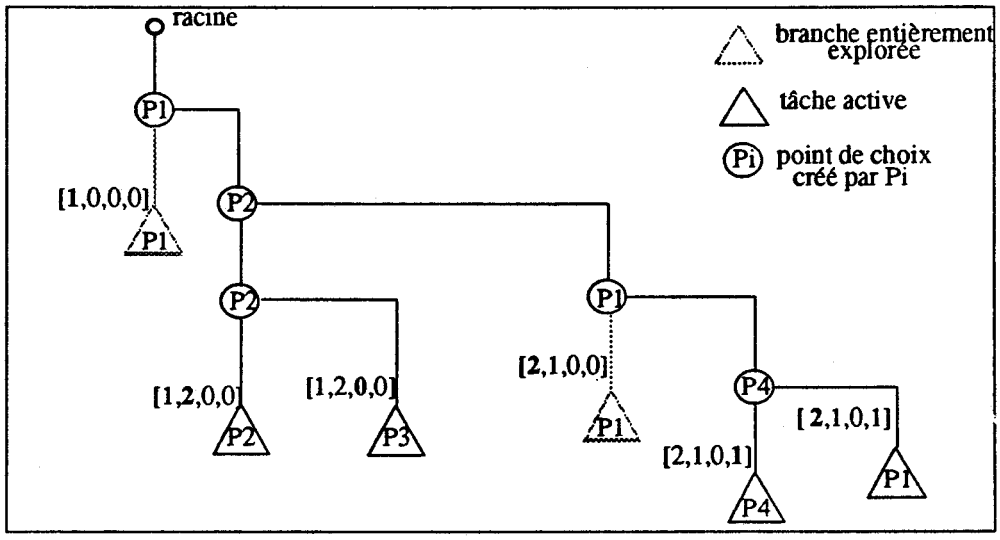



figure 3.8 : Jeu de clés des processeurs

La construction des clés s'effectue de la manière suivante :

**1** - Au départ toutes les clés du processeur commençant la résolution sont initialisées à 0.
**2** - Un processeur *Pi* incrémente (respectivement décrémente) sa clé privée *Ki* à chaque création (respectivement disparition) de point de choix. Seule la clé privée peut être modifiée par le processeur.
**3** - Lorsqu'un processeur *Pk* prend en charge une alternative d'un point de choix créé par un processeur *Pi*, il hérite du jeu de clés de *Pi* associé à ce point de choix.
**4** - Lorsqu'un processeur reprend une tâche suspendue, il récupère le jeu de clés associé à celle-ci.

Considérons l'exemple ci-dessus. La figure 3.9 illustre la construction du jeu de clés de chacun des processeurs. Le processeur P0 commençe la résolution avec un jeu de clés toutes égales à zéro. La création du premier point de choix a entrainé l'incrémentation de la clé privée de P1. La deuxième alternative de ce point de choix, ayant été prise en charge par le processeur P2, celui-ci a hérité de jeu de clés de P1. La clé privé de P2 a été incrémentée à chaque création de point de choix par P2. Le processeur P3, ayant pris en charge la deuxième alternative du deuxième point de choix créé par P2, a hérité du jeu de clés de P2, qui est courant à ce point de choix.

Le processeur P1, ayant terminé l'exploration de la première alternative issue du premier point de choix, a pris une nouvelle alternative créée par P4, ce qui explique l'état du jeu de clés de P1.

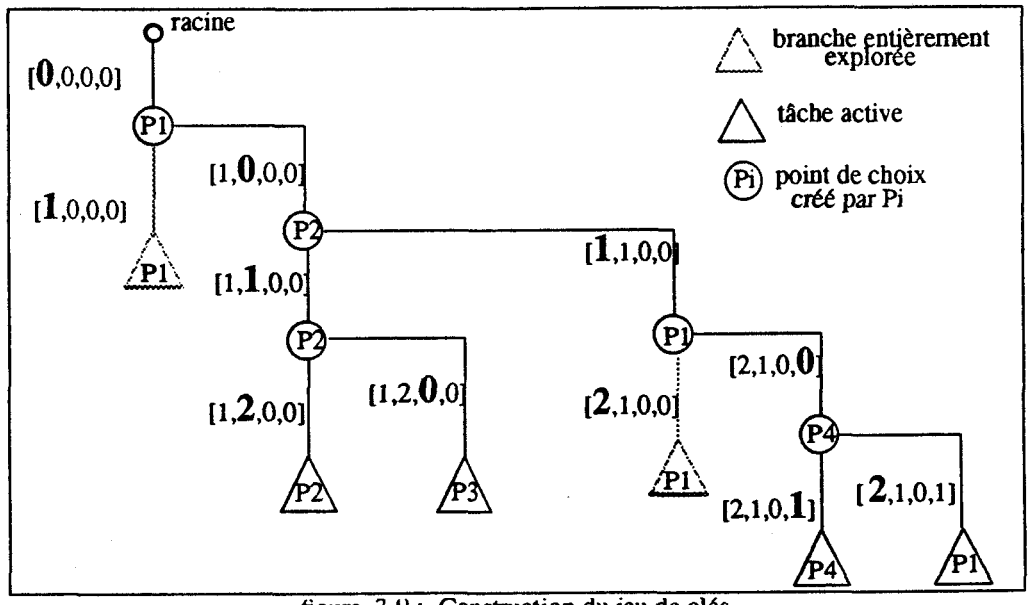



figure 3.9 : Construction du jeu de clés

## 2.2 - Mécanisme de liaison :

Lorsqu'un processeur est amené à lier une variable, deux cas peuvent se produire :

• La liaison est universelle :

Il s'agit nécessairement d'une variable créée par le processeur lors de la tâche courante; sa cellule est donc allouée sur la pile du processeur. La liaison d'une telle variable s'effectue en utilisant le mécanisme de liaison superficielle qui consiste à écrire la valeur de la liaison dans la cellule de la variable à lier. Une liaison universelle (non conditionnelle) est identifiée par une marque "u".

• La liaison est conditionnelle :

Il s'agit d'une variable qui a été créée antérieurement au point de choix dont est issue la branche traitée par le processeur voulant effectuer la liaison. Une telle variable est liée en utilisant le mécanisme de liaison profonde en étiquetant la liaison par la clé privée du processeur, ce qui permet aux autres processeurs de valider cette liaison.

La cellule d'une variable liée conditionnellement contient pour l'instant uniquement une marque "c" afin de l'identifier. Cette marque est positionnée lorsque la variable subit la première liaison profonde. Nous verrons, dans le prochain paragraphe, qu'elle peut contenir une valeur de liaison ou une information qui permet d'accéder au différentes liaisons profondes.

**Accès aux liaisons des variables :**

Lorsqu'un processeur veut connaître la valeur de liaison d'une variable, il effectue un accès direct à la cellule de la variable. Si celle-ci contient une liaison (i.e une liaison superficielle), elle est nécessairement valide puisqu'il s'agit d'une liaison universelle d'après le mécanisme de liaison décrit précédemment.

Dans le cas contraire, si la cellule de la variable ne contient pas la marque "c", la variable est libre; il faut alors d'abord vérifier si le processeur n'a pas lui-même lié la variable, auquel cas la liaison est nécessairement valide. Si le processeur ne possède aucune liaison profonde de la variable et que celle-ci a été créée lors de la tâche courante, la variable est libre; sinon, la recherche d'une éventuelle liaison valide a lieu chez les autres processeurs en commençant par le processeur qui a créé la variable. Pour valider une liaison profonde, le processeur compare la clé associée au processeur possédant la liaison à l'étiquette de celle-ci, cette étiquette représentant l'âge de la liaison. Si la valeur de cette dernière est strictement inférieure à la valeur de la clé, alors la liaison est valide. Si aucune liaison n'a été validée, la variable est considérée comme libre et peut être liée par le processeur.

Dans la figure 3.10, le processeur *P1* commence la résolution en créant les variables *X* et *Y*. Il lie la variable *Y* à la liste vide, avant de créer le premier point de choix; il s'agit donc d'une liaison universelle, expliquant l'utilisation de la liaison superficielle. La variable *X* étant liée conditionnellement par les processeurs *P1*, *P2* et *P3*, trois liaisons profondes ont été créées. Lorsque le processeur *P4* est amené à déréférencer la variable *X*, il effectue d'abord un accès direct à la cellule de *X*, qui contient la marque "c"; une première tentative de validation a lieu chez le processeur *P1* (créateur de la variable). La liaison *X/a*, effectuée par le processeur *P1*, n'est pas valide puisque son étiquette est égale à la clé associée à *P1* dans le jeu de clés de *P4*. Il en est de même pour la liaison *X/b* effectuée par *P2*. Par contre, la liaison *X/c* effectuée par *P3* est valide puisque son étiquette est strictement inférieure à la clé de *P3* héritée par *P4*.

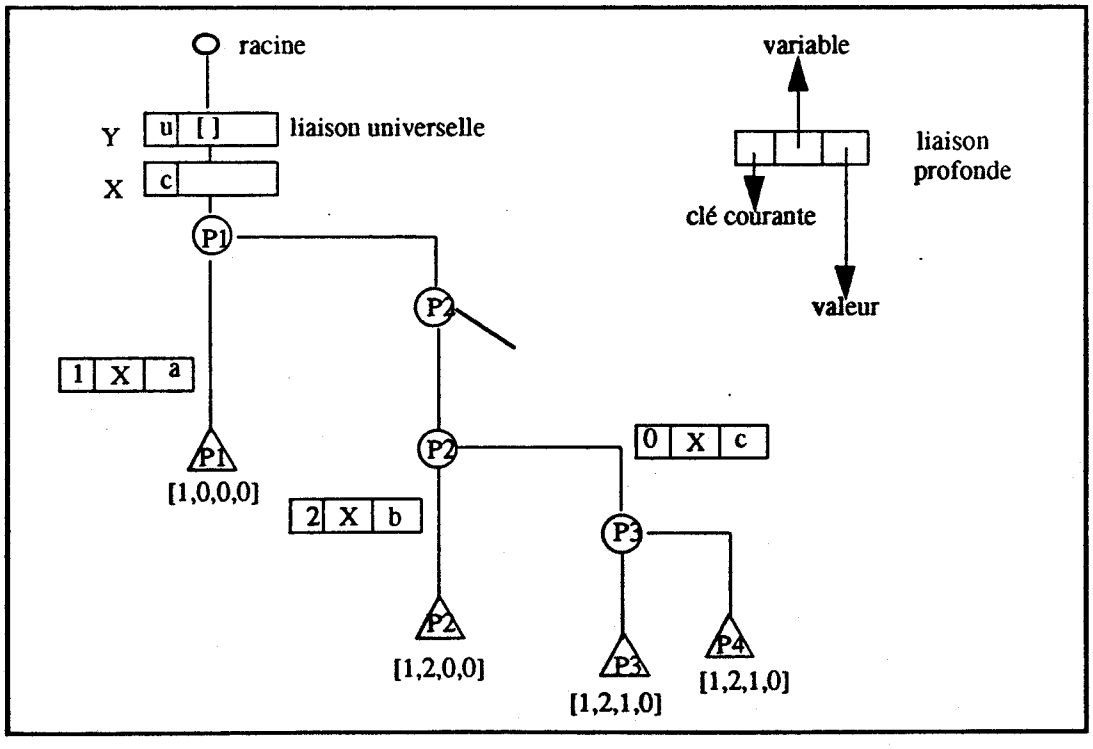



figure 3.10 : Mécanisme de liaison

Une clé héritée de valeur zéro indique que le processeur auquel est associée cette clé ne contient aucun point de choix pouvant être atteint[1] par le processeur ayant hérité de cette clé. De ce fait, il est inutile de vérifier, lors d'une recherche de liaison valide, qu'une telle clé peut valider une liaison. Dans notre exemple, avant d'instancier la variable *X*, le processeur *P3* doit d'abord effectuer une recherche de liaison valide, mais seuls les processeurs *P1* et *P2* sont susceptibles de posséder une telle liaison.

Le déréférencement d'une variable liée de façon conditionnelle, nécessite donc au maximum *n+1* accès, *n* étant le nombre de processeurs. Le premier accès a lieu pour tester si la variable est liée conditionnellement, tandis que les n autres accès proviennent de la recherche d'une liaison valide effectuée par l'un des processeurs.

Des techniques similaires de marquage de liaisons ont été utilisées dans les modèles Kabu-Wake et PEPsys. Dans le modèle Kabu-Wake, basé sur la recopie de l'environnement, le marquage des liaisons est utilisé uniquement pour éviter la pile de restauration. Dans le modèle PEPsys, un compteur du nombre de niveaux de point de choix, créés depuis le début de la branche (OBL), est associé à chaque processus explorant la branche en question; notre modèle utilise un compteur de points de choix par processeur, ce qui permet de borner l'accès aux liaisons profondes.

## 2.3 - Techniques de stockage des liaisons profondes :

Dans le mécanisme de liaison que nous venons de décrire, l'accès aux liaisons profondes n'a pas été considéré. L'accès aux liaisons superficielles est direct puisque la valeur de la liaison est écrite dans la cellule qui représente la variable, l'adresse de la cellule étant le nom de la variable. Mais le mécanisme de liaison profonde ne fournit pas un tel accès. En effet, la variable qui permet, dans le mécanisme de liaison superficielle, d'accéder directement à la valeur, est rangée avec la valeur dans une zone de données du processeur qui a effectué la liaison.

Pour permettre l'accès aux liaisons profondes, la restriction, assurée par le modèle de calcul, et qui consiste à limiter à un instant donné le nombre de liaisons d'une variable au nombre de processeurs, offre, pour gérer les liaisons profondes, trois opportunités décrites ci-après.

1. Un point de choix est atteint par un processeur s'il se trouve sur la branche liant la racine de l'arbre de recherche au noeud dont est issue la branche active sur le processeur.

### 2.3.1 - Utilisation des tables de hachage :

Cette technique consiste à associer à chaque processeur une table de hachage (hash-window) dont chaque entrée peut contenir une liaison profonde. L'accès à une liaison profonde s'effectue en appliquant une fonction de hachage sur le nom de la variable, le résultat de la fonction étant la position dans la table d'une éventuelle liaison de la variable.

Cette technique à été largement utilisée dans les modèles Pepsys et Argonne. A la différence de ces modèles, notre modèle permet l'utilisation d'une table de hachage par processeur au lieu d'une table par processus, ce qui permet une allocation statique des tables aux processeurs.

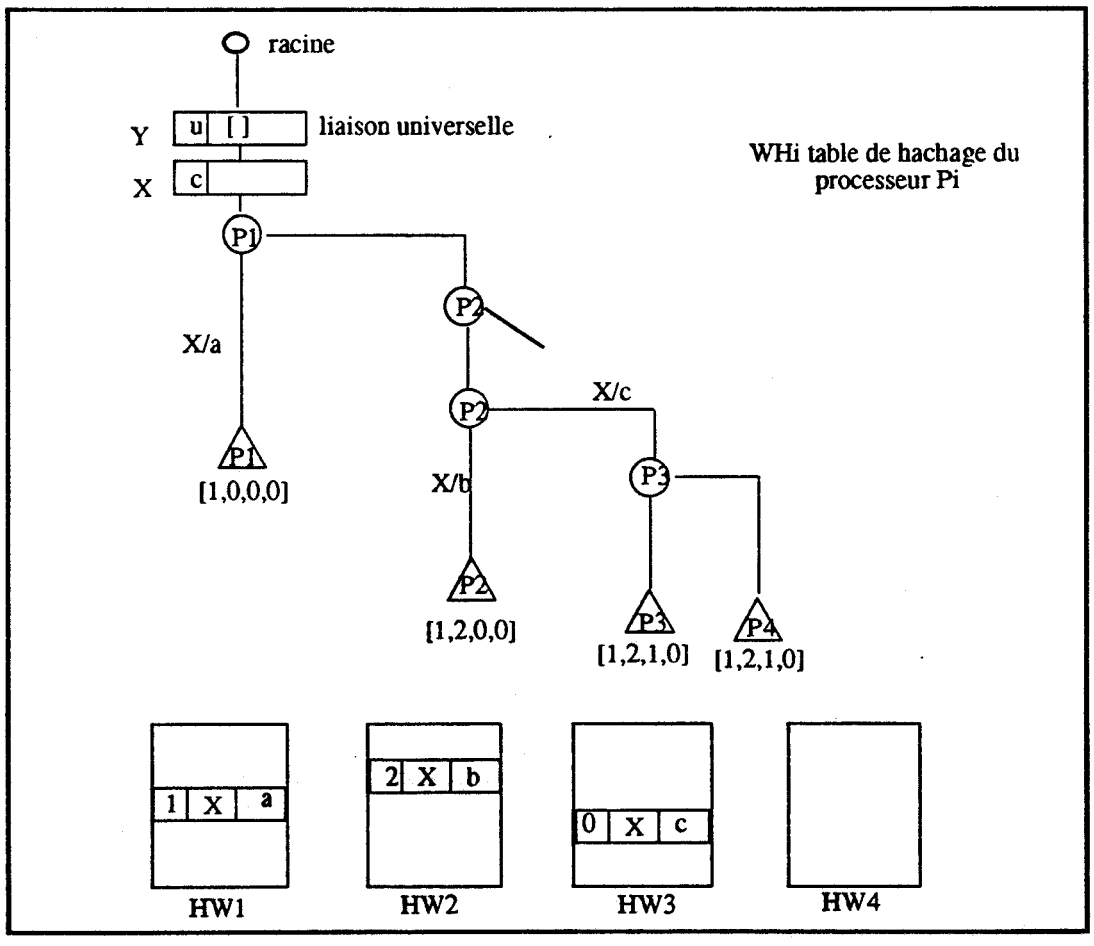



figure 3.11 : Mémorisation des liaisons profondes dans des tables de hachage

Chaque processeur, *P1*, *P2* et *P3*, liant la variable *X*, crée une liaison profonde qu'il mémorise dans sa table de hachage. Le processeur *P4* ne peut lier la variable *X* puisque la liaison X/*c* effectuée par *P3* est valide. La recherche d'une liaison conditionnelle valide consiste à explorer les différentes tables.

L'avantage de la technique de hachage pour stocker les liaisons profondes est qu'elle autorise à un processeur l'utilisation de la liaison superficielle pour lier une variable locale[1]. En effet, dans le mécanisme de liaison décrit précédemment, la cellule d'une variable liée conditionnellement ne contient qu'une marque qui peut être représentée par un seul bit, le reste des bits pouvant être utilisé par le processeur pour stocker la valeur de liaison. Ainsi, dans l'exemple de la figure 3.11 le processeur *P1* peut-il utiliser le mécanisme de liaison superficielle pour lier la variable *X* (voir figure 3.12).

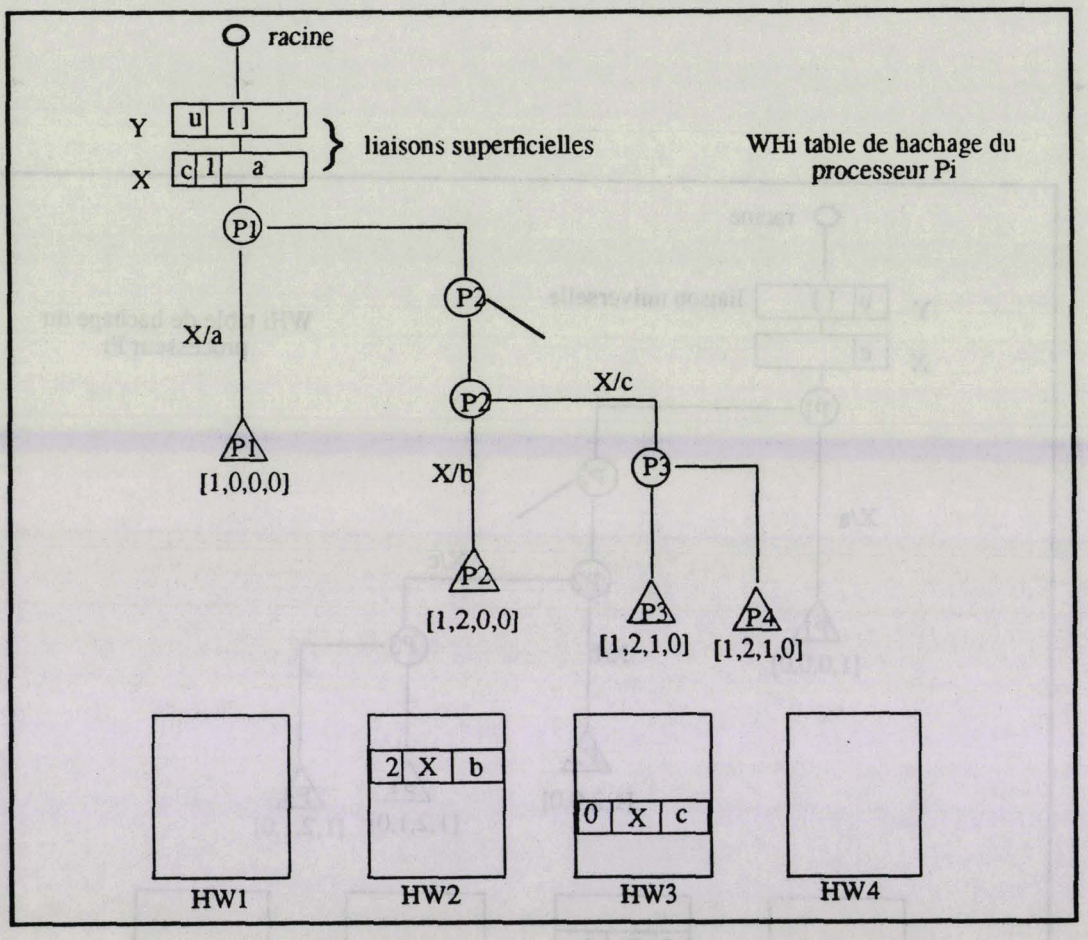



figure 3.12 : Utilisation de la liaison superficielle lors des liaisons locales

Si la liaison superficielle est utilisée systématiquement à chaque fois qu'il s'agit d'une liaison locale, l'accès aux liaisons des variables locales n'est pas toujours direct. En effet, si la variable est créée lors de la tâche active, l'accès peut être direct (lecture de la cellule). Par contre, si la variable a été créée lors d'une tâche suspendue, elle peut être liée par un autre processeur en utilisant la liaison profonde, dans ce cas le processeur effectuant l'accès doit valider une telle liaison.

1. Une variable est dite locale si elle a été créée par le processeur en question, et pas nécessairement lors de la tâche active.

Si le processeur *P1* prend en charge l'alternative en attente chez le processeur *P2* pour l'aider, la liaison *X/a* est défaite avant de commencer la nouvelle tâche. Si au cours de celle-ci, *P1* est amené à lier la variable X, il doit s'assurer que la liaison X/b effectuée par *P2* n'est pas valide.

Le temps d'accès aux tables de hachage peut se révéler pénalisant. En effet, chaque accès à une table nécessite un calcul d'une fonction de hachage, qui dans certains cas (lors des collisions) ne suffit pas pour déterminer la position dans la table de l'information recherchée.

### 2.3.2 - Utilisation des vecteurs de liaisons :

Cette technique a été utilisée pour la première fois dans le modèle Versions-Vectors présenté dans le chapitre précédent (paragraphe 2.3.3.6). Elle consiste à associer à chaque variable liée conditionnellement, un vecteur dont la taille est le nombre de processeurs. Chaque cellule du vecteur peut être utilisée par un seul processeur pour lier la variable. Dans la figure 3.13, l'exemple précédent (fig 3.12) est repris en utilisant des vecteurs :

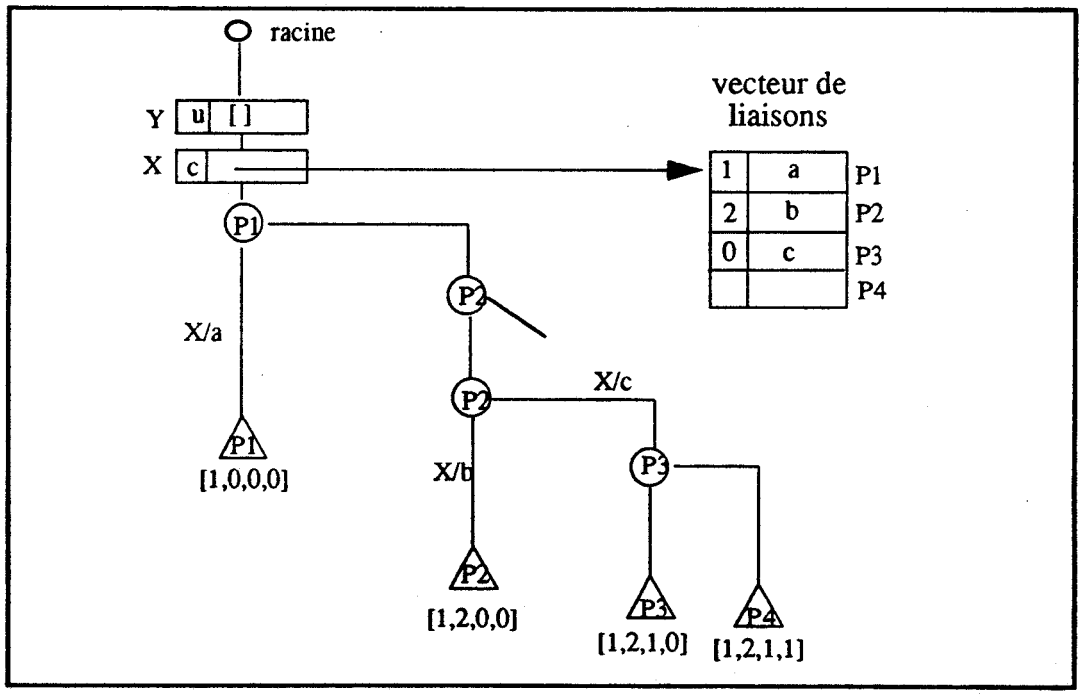



figure 3.13 : Utilisation des vecteurs de liaisons pour stocker les liaisons profondes

Contrairement à la technique précédente, le mécanisme de liaison superficielle n'est utilisé que lors d'une liaison universelle (*Y/[]*). La cellule d'une variable liée de façon conditionnelle contient un pointeur vers le vecteur mémorisant les différentes liaisons. La recherche d'une liaison valide consiste à explorer le vecteur de liaisons associé à la variable concernée.

Le temps d'accès à une liaison est inférieur à celui de la méthode précédente. A l'inverse, elle nécessite une synchronisation entre les processeurs liant simultanément une même variable. En effet, l'allocation d'un vecteur à une variable liée conditionnellement, a lieu lors de la première liaison de cette variable. Cette première liaison peut être effectuée par un processeur quelconque, plusieurs processeurs pouvant être amenés à lier une même variable au même moment.

### 2.3.3 - Utilisation des tableaux de liaisons :

La technique utilisée dans le modèle SRI, qui consiste à mémoriser les liaisons conditionnelles dans des tableaux de liaisons, peut être adaptée en raison des caractéristiques du modèle. Chaque processeur possède un tableau de liaison appelé *PBA*, qui contient toutes les liaisons conditionnelles effectuées par ce processeur. La cellule d'une variable liée conditionnellement contient un index dans le tableau de liaison. Lorsqu'un processeur lie une telle variable, il utilise l'index pour stocker la liaison dans son tableau. Dans la figure ci-dessous, l'exemple précédent est repris en considérant les tableaux de liaisons :

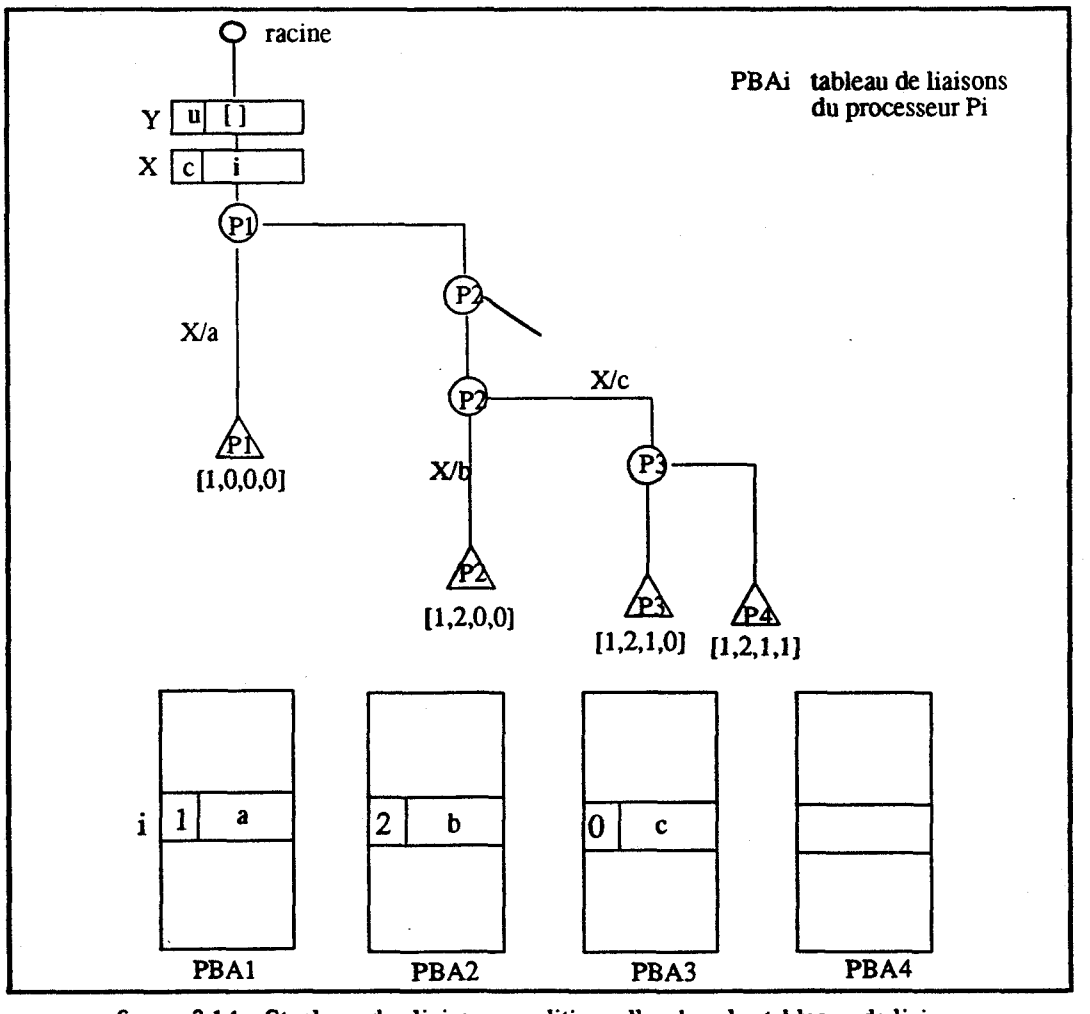



figure 3.14 : Stockage des liaisons conditionnelles dans les tableaux de liaisons

Lorsque le processeur *P4* est amené à déréférencer la variable *X*, il effectue un accès direct pour récupérer l'index *i*, cet index étant utilisé pour accéder aux différentes liaisons effectuées par les processeurs. La liaison superficielle est utilisée uniquement pour lier une variable de façon universelle.

Pour mettre en oeuvre cette technique, chaque processeur dispose d'un compteur de variables. Lorsqu'un processeur crée une variable (i.e invoque une clause qui contient une variable), il initialise la cellule de celle-ci avec la valeur courante du compteur. Un index doit être associé à une seule variable, et doit permettre l'accès aux différentes liaisons effectuées dans les différents tableaux. Pour cela, lorsqu'un processeur prend en charge une alternative dans un point de choix, il hérite de celui ci l'index courant du processeur créant l'alternative. Afin de réaliser ce passage d'index, est mémorisé dans chaque point de choix l'index courant du processeur l'ayant créé.

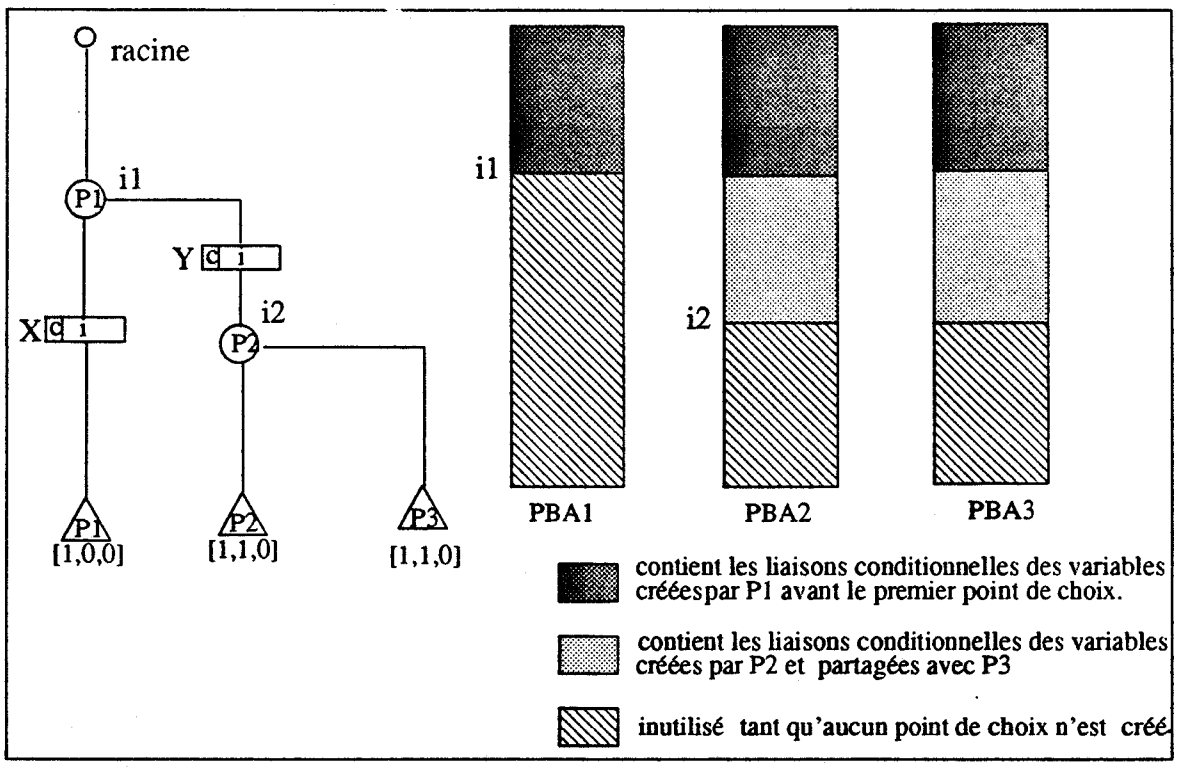



figure 3.15 : Compteur de variables

Il peut se produire des situations entraînant le partage du même index par deux variables différentes. Ainsi, les variables *X* et *Y*, créées respectivement par *P1* et *P2*, peuvent contenir le même index *i* tel que i1< i < i2. Mais une confusion ne peut avoir lieu pour aucun des deux processeurs, car les variables sont locales.

Par contre, lorsque le processeur *P3* est amené à déréférencer la variable *Y*, il doit vérifier que la variable *Y* n'a pas été liée par *P1*, puisque la clé associée à *P1* dans le jeu de clés de *P3* n'est pas nulle. Dans ce cas, une éventuelle liaison se trouvant à l'index *i* dans le tableau de liaisons de *P1* ne correspond pas à une liaison de la variable *Y*. Cette éventuelle liaison ne peut être valide puisqu'elle est nécessairement étiquetée par un numéro supérieur à la valeur de la clé de *P1* héritée par *P3*.

Une solution, pour éviter toute tentative de validation d'une liaison ne correspondant pas à la variable en question, consiste à stocker dans chaque *PBA*, le nom de la variable et sa valeur. Cette solution introduit un test supplémentaire à chaque accès dans un *PBA* pour vérifier si la liaison correspond bien à la variable à déréférencer. Le problème demeure inchangé puisque le test de comparaison d'une étiquette et d'une clé, est remplacé par un test d'égalité d'une adresse et du champ *variable* d'une liaison profonde.

L'utilisation des *PBA*s semble pallier les principaux défauts des deux méthodes précédentes, puisqu'elle permet un accès plus rapide aux liaisons profondes et ne nécessite aucun mécanisme de synchronisation lors des liaisons. Le tableau ci-dessous compare les trois méthodes. Les points de comparaison sont indiqués en gras dans le tableau.

| | table de hachage | vecteurs de liaisons | tableaux de liaisons |
|---|---|---|---|
| **Temps d'accès aux liaisons profondes** | lent - | rapide + | rapide + |
| **Utilisation de la liaison superficielle** | lors des liaisons locales ++ | lors des liaisons universelles + | lors des liaisons universelles + |
| **Synchronisation lors des liaisons** | inutile + | indispensable - | inutile + |
| **Allocation de la structure** | statique + | dynamique - | statique + |
| **Allocation d'une cellule** | dynamique ++ | statique + | statique - |
| **Format de la cellule** | (clé,variable,valeur) - | (clé, valeur) + | (clé, valeur) + |

figure 3.16 : Comparaison des trois méthodes

Le seul inconvénient de l'utilisation des tableaux de liaisons est le gaspillage de l'espace mémoire. En effet, lorsqu'une variable est créée, un index lui est associé. *N* cellules (n étant le nombre de processeurs) sont réservées dans les tableaux pour accueillir les liaisons effectuées par les différents processeurs (même si la variable peut être liée de façon universelle par la suite). Ce problème n'existe pas dans la méthode basée sur l'utilisation des vecteurs de liaisons. En effet, l'allocation d'un vecteur n'a lieu que lorsqu'une variable est liée conditionnellement. En revanche, l'allocation systématique d'un vecteur de taille égale au nombre de processeurs ne garantit pas l'utilisation de l'ensemble des cellules du vecteur. Une utilisation optimisée de l'espace mémoire alloué aux liaisons profondes est assurée par la première méthode, car chaque cellule d'une table de hachage peut être réellement utilisée.

Le mécanisme de liaison que nous venons de décrire permet d'une part l'installation des tâches à un prix très réduit puisque aucun environnement de calcul n'est recopié, et d'autre part, un accès aux liaisons profondes qui est borné. La gestion du retour-arrière, qui est à la base de ce mécanisme de liaison, contrôle le nombre de liaisons d'une même variable. Ce contrôle a permis la mise en place de trois techniques de stockage des liaisons profondes qui jusque là ont toujours été traitées par l'utilisation des tables de hachage dans les modèles OU-parallèles évitant toute recopie d'environnement.

Dans le chapitre suivant, nous montrerons les extensions apportées à une machine abstraite séquentielle que nous avons décrite dans le premier chapitre, afin d'implanter le modèle de calcul parallèle que nous venons de décrire.

# -=- Chapitre IV -=-

## *Machine Abstraite Parallèle*

Dans ce chapitre, nous présentons une méthode pour implanter le modèle de calcul décrit dans le chapitre précédent. Cette méthode est basée sur l'utilisation d'une machine abstraite ou virtuelle.

Une machine abstraite est définie ici comme étant l'ensemble des structures de données et d'instructions, qui permet d'exprimer simplement les programmes écrits dans le langage source, et qui se prête facilement à la génération d'instructions sur les machines cibles. La définition d'une telle machine abstraite est fréquemment employée en compilation. Le compilateur du langage traduit alors les programmes sources en instructions de la machine abstraite.

Comme le constate D.H. Warren [War 83], le code intermédiaire ainsi généré peut être traité de trois façons différentes : par émulation, par génération des instructions de la machine cible, ou directement par un processeur dédié [Tic 83].

La définition précédente est valable pour un modèle de calcul séquentiel. Dans un environnement parallèle, ces spécifications sont insuffisantes. Elles ne peuvent, en effet, que décrire le fonctionnement d'un processeur élémentaire en excluant toute interaction avec les autres processeurs.

Pour avoir une description plus complète d'un système parallèle, il faut décrire les deux notions supplémentaires introduites par le parallélisme, qui sont, d'une part la façon dont les processeurs interagissent et se synchronisent, et d'autre part, le modèle d'exécution (appelé *scheduling* en anglais). Cette deuxième notion fait référence à la façon dont sont gérées les activités des processeurs. Nous distinguons le modèle d'exécution du modèle de calcul qui lui, fait référence au contrôle du parallélisme.

Le chapitre sera organisé en deux parties. Dans la première partie, nous présenterons l'organisation de la mémoire de la machine abstraite parallèle, qui découle des extensions introduites dans la machine abstraite séquentielle (WAM), pour permettre l'implantation du modèle de calcul parallèle. Dans la deuxième nous mettrons l'accent sur la gestion des activités des processeurs, et nous donnerons une description de la façon dont interagissent et se synchronisent les processeurs.

# 1 - Organisation de la mémoire :

Les spécifications du modèle OU-parallèle, qui sont d'une part l'utilisation de la stratégie multiséquentielle, et d'autre part, le partage de l'environnement du calcul, ont permis de distinguer deux parties de la mémoire. La première partie de la mémoire est partagée par les différents processeurs, la deuxième est formée de l'ensemble des mémoires privées des processeurs. La mémoire privée d'un processeur contient toutes les données qui ne peuvent être ni lues ni modifiées par les autres processeurs.

## 1.1 - Mémoire partagée :

La mémoire partagée est constituée (fig. 4.1) d'une zone qui contient les instructions du programme, et des différentes zones de travail des processeurs. Une zone de travail de processeur contient les piles de la machine abstraite séquentielle (excepté la pile de restauration qui se trouve dans la mémoire privée du processeur) ainsi que les deux structures de données introduites, qui sont la pile des choix et le tableau des liaisons.

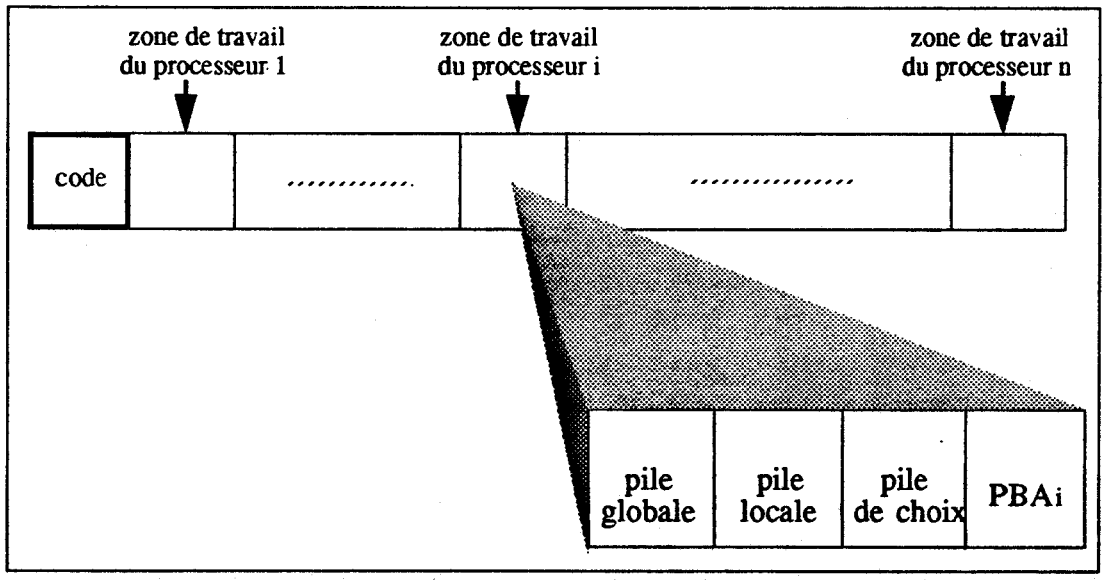



figure 4.1: Organisation de la mémoire partagée

Afin de simplifier la représentation des références et l'accès aux données, nous considérons que le partitionnement de la mémoire partagée se fait de manière statique et ne peut être modifiée au cours de l'exécution.

L'accès aux données par un processeur dans sa zone de travail, s'effectue en utilisant une adresse qui est un index dans l'une des quatre structures de la zone de travail; nous appellerons une telle adresse, une **adresse locale.**

Dans le cas où un processeur veut accéder à une donnée appartenant à un autre processeur, il doit fournir une **adresse non-locale** qui est composée d'une identification du processeur possédant la donnée, et d'une adresse locale.

### 1.1.1 - La pile locale :

Une définition de la notion de processeur virtuel et de processus a été donnée dans le chapitre III. Un processeur virtuel est une instance de la machine abstraite séquentielle, à laquelle sont ajoutées deux structures supplémentaires, à savoir, la pile de choix et le tableau de liaisons. Un processus (ou tâche) est créé lorsque le processeur prend en charge une nouvelle alternative dans l'arbre de recherche, et se termine lorsque le sous-arbre associé à cette alternative est entièrement exploré.

A cause du mécanisme de suspension introduit par le retour-arrière, plusieurs processus peuvent cohabiter sur un même processeur, mais seul le processus le plus jeune est actif, les autres étant suspendus. De ce fait, la gestion des processus peut être mise en oeuvre à l'aide de la pile locale.

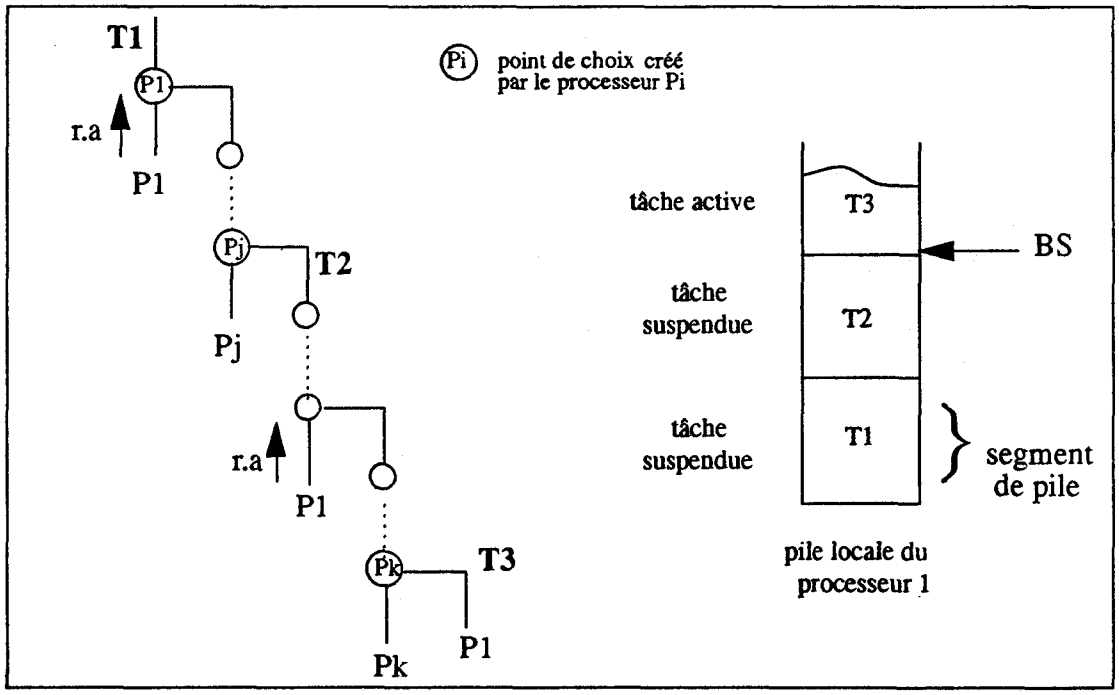



figure 4.2 : Gestion des tâches à l'aide de la pile locale

Dans la figure 4.2, le processeur *P1* ne pouvant effectuer le retour-arrière lors de la tâche T1, entreprend une nouvelle tâche T2; celle-ci est suspendue à son tour, suite au retour-arrière qui est impossible à réaliser. Une troisième tâche T3 est prise en charge par le processeur P1.

Chaque tâche est identifiée par la base du segment de pile contenant l'environnement de cette tâche. Chaque processeur possède un registre **BS** qui contient l'index de la base du segment de pile alloué à la tâche active. Ce registre est mis à jour à chaque changement de tâche.

Le segment de pile alloué à chaque tâche sur la pile locale contient les blocs d'activation décrits dans le premier chapitre. Ces blocs d'activation sont étendus afin de gérer le parallélisme.

### 1.1.1.1 - Les blocs déterministes :

Un bloc déterministe est créé lors de l'activation d'une clause qui constitue la seule alternative restante pour résoudre un but. Il est formé par une partie contrôle qui permet de gérer l'avancée dans la résolution, et une partie environnement contenant l'ensemble des variables figurant dans la clause. Afin d'implanter le mécanisme de liaison basé sur l'utilisation des tableaux de liaisons, chaque partie contrôle d'un bloc déterministe contient un champ **I** qui mémorise la valeur courante du compteur de variables. Ce champ permettra la mise à jour du compteur de variables lors de la disparition du bloc.

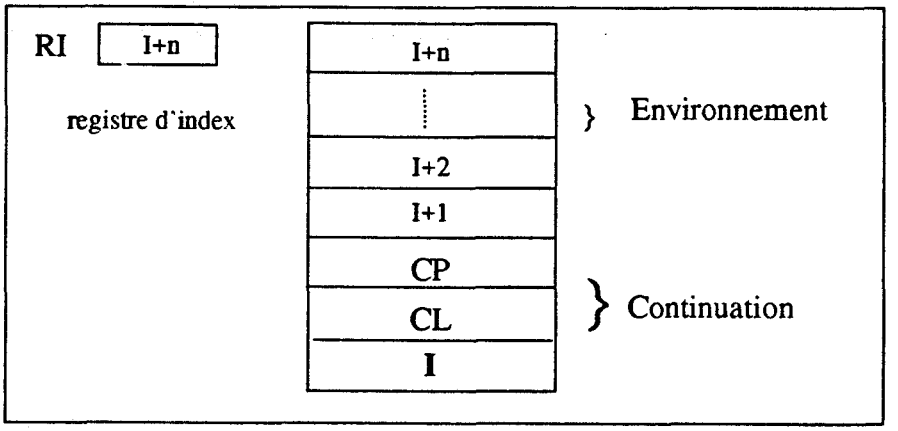



figure 4.3 : Extension du bloc déterministe

La figure ci-dessus représente un bloc d'activation associé à l'appel d'une clause contenant *n* variables. A la création, chaque variable contient un index qui est la valeur courante du compteur de variables. Cette valeur est mémorisée dans un registre **RI** (**R**egistre d'**I**ndex) qui est incrémenté à chaque création de variable. Dans l'exemple de la figure 4.3, le registre RI a été incrémenté *n* fois. La valeur de RI précédant l'appel est mémorisée dans le champ I.

### 1.1.1.2 - Les blocs de choix :

Un bloc de choix est créé lors de l'activation d'une clause qui n'est pas la dernière clause du paquet de clauses à considérer. Il contient, en plus des informations d'un bloc déterministe, une partie *reprise* pour gérer le retour-arrière (voir paragraphe 1.1.2 du chapitre II).

Les blocs de choix constituent les sources de parallélisme dans une exécution OU-parallèle. Pour cela, ils doivent être étendus pour mémoriser les identifications des processeurs traitant des branches issues de ces points de choix, ainsi que d'autres informations pour contrôler le parallélisme.

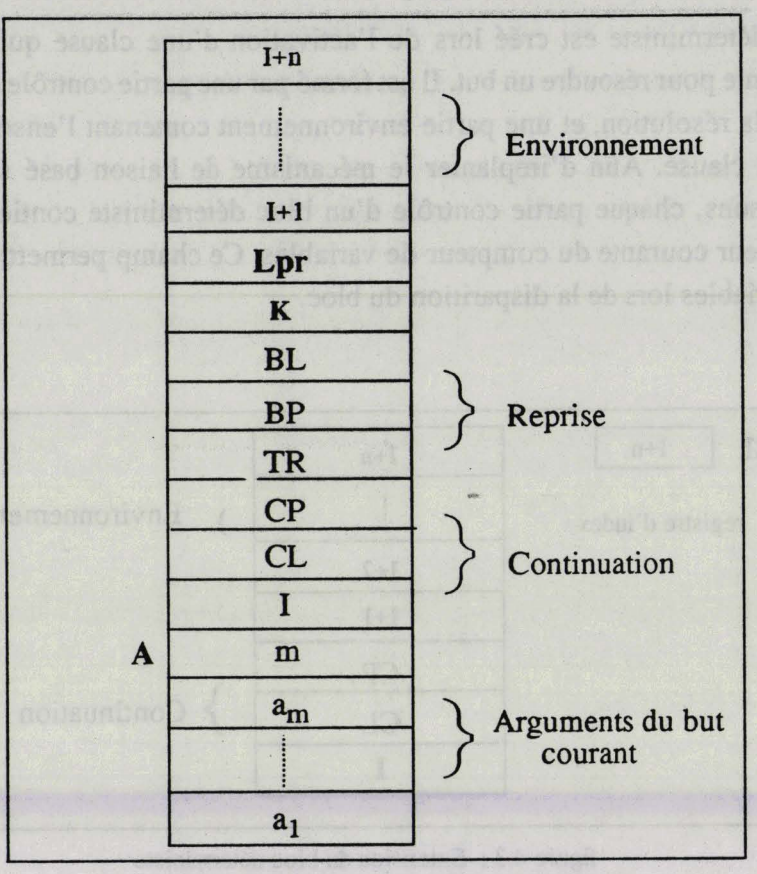



figure 4.4 : Extension d'un bloc de choix

Trois champs supplémentaires **A, Lpr** et **K** (en plus du champ I évoqué précédemment) ont été introduits afin de contrôler le retour-arrière et de mettre en oeuvre le mécanisme d'héritage du jeu de clés lors de l'initialisation des tâches.

**(1)** Le champ A est constitué de m+*1* emplacements. Le premier emplacement indique l'arité m du but, les m autres contiennent les arguments. Le champ A, également utilisé dans une implantation séquentielle [Boi 88], sert à retrouver, lors du retour en arrière, l'instance du but à redémontrer. Dans une implantation OU-parallèle ce champ permet à un processeur prenant en charge une alternative d'un point de choix, de charger les arguments du but courant.

**(2)** Le champ Lpr mémorise la liste des processeurs traitant des alternatives du point de choix. Il peut être implanté à l'aide d'un vecteur de bits, dont la taille est le nombre total des processeurs. Lorsqu'un processeur prend en charge une alternative d'un point de choix, il positionne le bit adéquat à 1. Celui-ci est remis à 0 lorsque l'alternative est entièrement explorée. Un point de choix est amené à disparaître lorsque tous les bits du champ K sont égaux à 1 : ceci permet au processeur ayant créé ce point de choix de contrôler le retour-arrière décrit dans le modèle de calcul.

(3) Le champ K mémorise l'état courant du jeu de clés du processeur, au moment de la création du bloc de choix. Ce champ a été introduit afin de permettre à un processeur oisif prenant en charge une alternative, de recopier le jeu de clés du processeur ayant créé cette alternative. Comme il a été spécifié dans le modèle de calcul, seule la clé privée est modifiée (i.e. incrémentée à chaque création d'un point de choix) par le processeur, les autres clés sont inchangées au cours d'une tâche. Un processeur acquiert un nouveau jeu de clés lors du changement de tâche. Afin d'éviter de recopier entièrement l'état du jeu de clés du processeur, dans chaque bloc de choix, seule la clé privée du processeur ayant créé le point de choix est mémorisée dans un bloc de choix, les autres clés sont mémorisées à la base du segment de pile alloué à la tâche en cours. Le champ K contient désormais la clé privée du processeur.

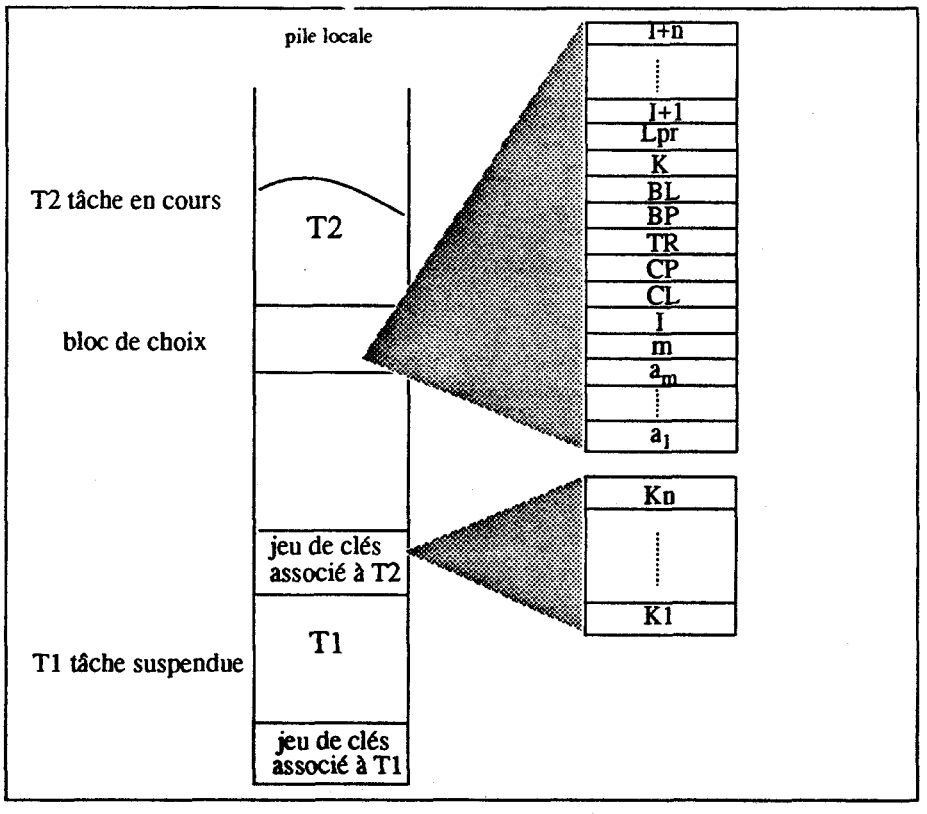



figure 4.5 : Représentation du jeu de clés

L'exemple ci-dessus montre un aperçu de la pile locale d'un processeur possédant une tâche T1 en suspens, et une tâche active T2 au cours de laquelle un point de choix a été créé. A la base du segment de pile alloué à chaque tâche, est mémorisé l'état courant du jeu de clés du processeur lors de cette tâche. Un processeur oisif qui prend une éventuelle alternative non encore traitée et créée lors de l'une de ces deux tâches effectue les deux actions suivantes : dans un premier temps, il recopie le jeu de clés mémorisé dans le segment de pile; dans un deuxième temps il remplace dans cette copie, la clé associée au processeur qui a créé l'alternative, par la valeur du champ K mémorisé dans le point de choix au niveau du quel est considérée l'alternative.

Afin de permettre à un processeur oisif d'accéder à la base de la pile locale d'une tâche offrant du travail, chaque point de choix doit mémoriser le numéro de la tâche l'ayant créé. Ce numéro a été défini comme l'index de la base du segment de pile alloué à la tâche. Nous verrons que cette information sera mémorisée dans la pile de choix.

Lorsqu'un processeur suspend une tâche pour en prendre une nouvelle, il doit sauvegarder l'état des registres de travail. Cette sauvegarde a lieu dans la pile locale, afin d'éviter d'introduire une nouvelle pile. Un segment de pile d'une tâche suspendue est composée de trois parties (figure 4.6). A la base se trouve l'état du jeu de clés du processeur lors de la suspension; en sommet du segment de pile, est sauvegardé l'état des registres de travail; le reste du segment contient l'environnement du calcul.

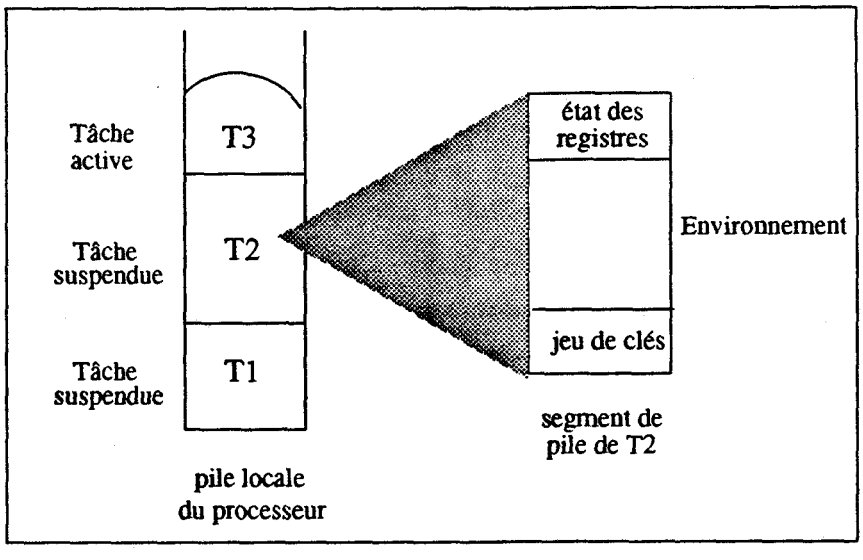



figure 4.6 : Sauvegarde de l'état des registres de travail lors d'un changement de tâche

### 1.1.2 - La pile de choix :

La pile de choix a été introduite afin de permettre à un processeur de publier un travail qui peut être réalisé en parallèle. Chaque processeur gère sa propre pile de choix qui rassemble les pointeurs vers les points de choix créés par le processeur, et qui peuvent être traités en parallèle. Un processeur oisif explore les différentes piles de choix des processeurs actifs, à la recherche d'une branche de l'arbre de recherche non traitée. Lorsqu'un point de choix possédant une alternative en attente, est trouvé par un processeur oisif, la tâche au cours de laquelle ce point de choix a été créé doit être identifiée. Pour cela, lorsqu'un processeur publie un point de choix, il range dans la pile de choix l'adresse du point de choix, ainsi que le numéro de la tâche courante.

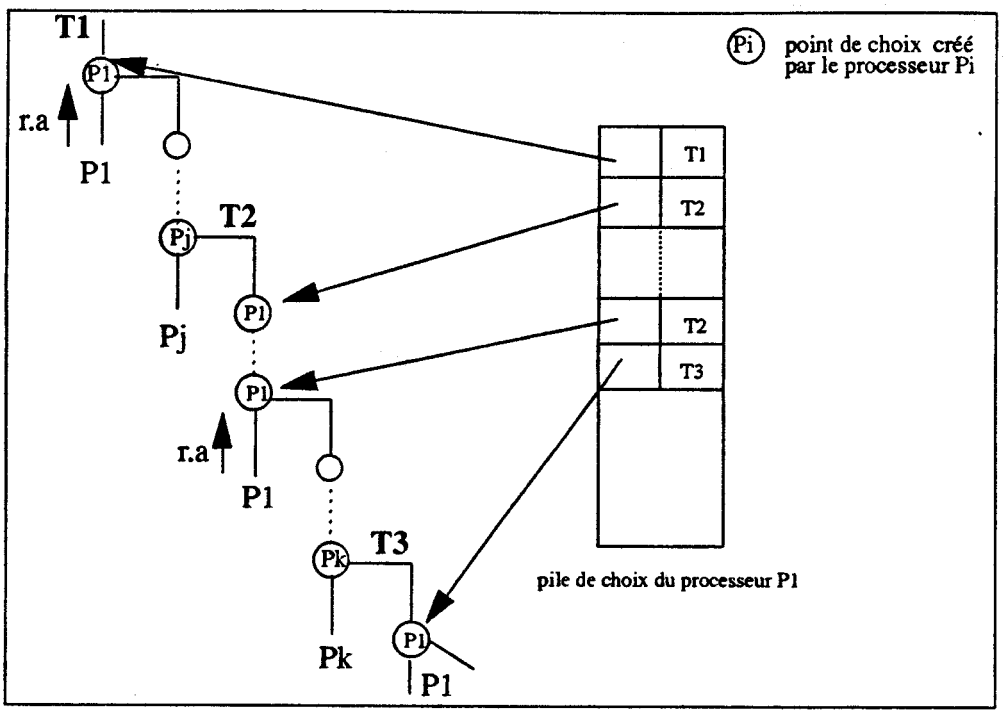



figure 4.7 : Publication des points de choix

La figure 4.7 montre l'état de la pile de choix du processeur P1 exécutant la tâche courante T3, après avoir suspendu successivement les tâches T1 et T2. Chaque élément de la pile de choix contient une information qui permet à un processeur oisif d'accéder à un point de choix créé par P1, et une deuxième information (numéro de la tâche) qui permet d'identifier la tâche courante du processeur P1 lorsque le point de choix a été créé. Cette deuxième information permet à un processeur prenant en charge une alternative issue de ce point de choix, de recopier le jeu de clés qui était celui du processeur lors de la création de cette alternative.

Comme nous l'avons mentionné dans le chapitre III, la pile de choix peut sembler inutile. En effet, les processeurs oisifs peuvent directement explorer les piles locales des processeurs actifs. Le champ BL (bloc de choix précédent) mémorisé dans chaque point de choix, permet d'accéder à tous les points de choix créés. L'inconvénient de cette solution, est qu'elle risque de ralentir la recherche d'une alternative en attente, à cause de l'exploration de certains points de choix qui ne détiennent plus d'alternatives. L'utilisation d'une structure de données supplémentaire pour gérer les travaux disponibles, permet d'éviter des parcours inutiles de points de choix dont les alternatives sont déjà traitées (ou en cours de traitement). En effet, lorsqu'un processeur prend en charge la dernière alternative d'un point de choix, il peut mettre à jour la structure en faisant disparaître, dans celle-ci, l'adresse de ce point de choix. Seuls donc, les points de choix possédant des alternatives en attente, sont représentés dans la structure. A cette fin, la structure mémorisant les adresses des points de choix doit permettre de telles mises à jour. Dans un premier temps, nous avons considéré une structure de pile en raison de sa simplicité. Cette structure ne permet pas de mettre en place la mise à jour décrite précédemment. Toutefois, nous verrons que le choix d'une structure peut dépendre de la stratégie de recherche de travail disponible, par un processeur oisif, et qu'une structure de pile peut être envisagée.

### 1.1.3 - La pile globale :

La pile globale, telle qu'elle a été définie dans le chapitre I, est utilisée dans une implantation séquentielle pour stocker les termes complexes tels que les listes et les termes fonctionnels. Elle contient l'ensemble des termes créés sur une branche de l'arbre de recherche. Cette définition reste valable dans notre modèle parallèle si l'on considère une seule tâche par processeur. Mais elle peut être généralisée en considérant toutes les tâches entreprises par un processeur grâce à leur gestion par pile. La pile globale contient tous les termes créés au cours de ces tâches. La mise à jour, au cours d'une tâche, de la pile globale est semblable à celle utilisée dans une implantation séquentielle : elle a lieu lors de chaque retour arrière. Une autre mise à jour est nécessaire lors d'un changement de tâche. Lorsqu'un processeur récupère une tâche suspendue, le sommet de la pile globale doit être celui qui était courant lors de la suspension de cette tâche. Cette mise à jour est possible grâce à la restauration de l'état des registres lors de la reprise d'une tâche suspendue, le registre G (voir chapitre I) sauvegardé permet de retrouver le sommet de la pile globale lors de la suspension.

Afin d'implanter la notion de *localité* d'une variable, introduite par le mécanisme de liaison décrit dans le chapitre précédent, un registre supplémentaire **BGS** est utilisé pour mémoriser la base du segment de pile qui contient l'environnement en pile globale, créé lors de la tâche courante.

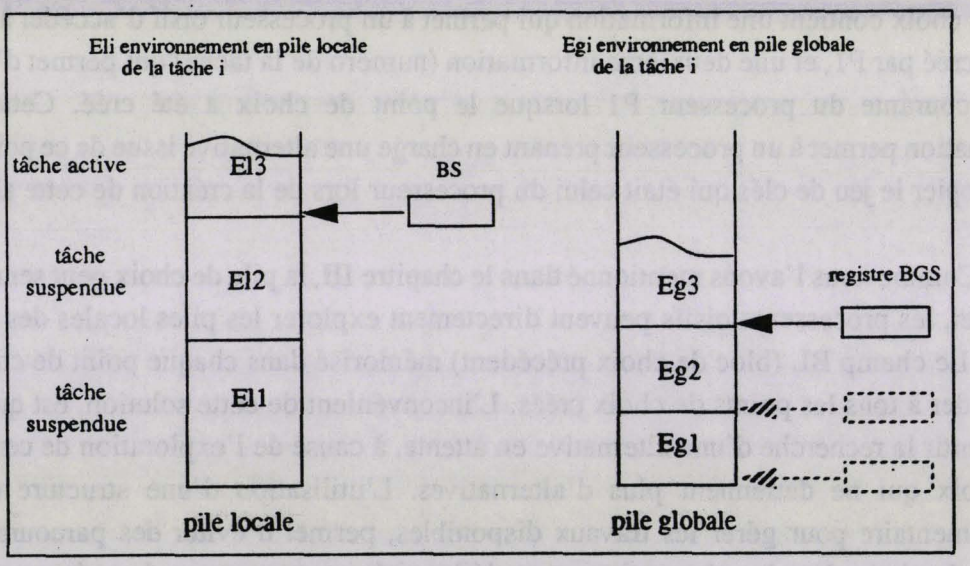



figure 4.8 : Gestion de la pile globale

La figure 4.8 montre l'état des piles locale et globale d'un processeur possédant deux tâches suspendues T1 et T2, et une tâche active T3.

Les registres BS et BGS permettent, lors de l'unification, de déterminer si une variable est créée lors de la tâche courante ou non, en comparant son adresse avec les valeurs de ces deux registres.

### 1.1.4 - Le tableau de liaisons :

Le tableau de liaisons contient les liaisons conditionnelles effectuées par le processeur. Comme précisé dans le chapitre III, à chaque variable créée, est associé un index dans le tableau de liaisons. Cet index est mémorisé dans le registre RI qui est incrémenté à chaque création de variable. La valeur courante du registre RI est mémorisée dans chaque bloc d'activation (bloc de choix ou bloc déterministe) afin de permettre la mise à jour du registre RI lors de la disparition de ce bloc, ou lors du retour arrière s'il s'agit d'un bloc de choix.

La mise à jour du registre RI doit accompagner chaque mise à jour de la pile locale, puisqu'il mémorise le nombre de variables créées sur la pile locale. La pile locale est mise à jour lors du retour arrière et lors du retour terminal.

#### 1.1.4.1 - Mise à jour du tableau de liaisons lors du retour en arrière :

Lors du retour arrière à un point de choix, la totalité de l'environnement créé au cours de la résolution de la plus récente alternative considérée dans ce point de choix, doit être détruit pour récupérer l'espace mémoire. Cet environnement est constitué d'une partie en pile locale et d'une partie en pile globale. La mise à jour du registre RI est simplement effectuée en le chargeant de la valeur courante lors de la création du point de choix en question. Cette valeur est mémorisée dans le champ I du point de choix.

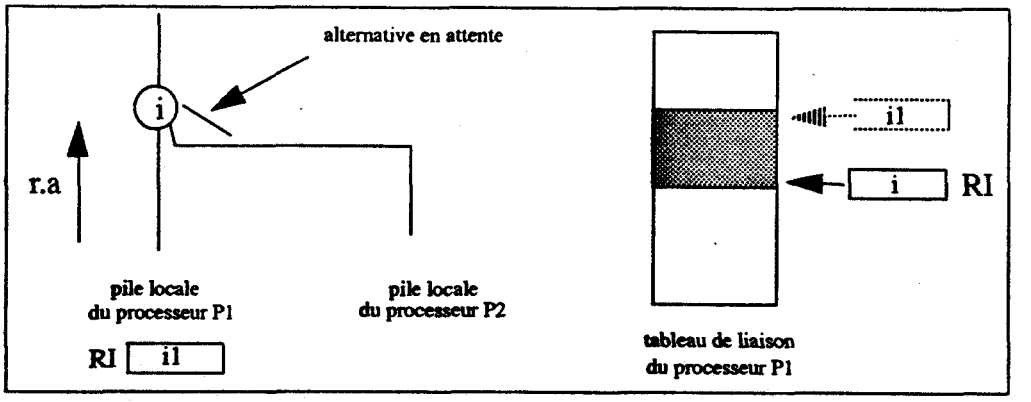



figure 4.9 : Mise à jour du tableau de liaisons lors du retour en arrière

La figure 4.9 illustre la mise à jour du tableau de liaisons du processeur *P1* lors du retour-arrière pour traiter l'alternative en attente. L'index *i*1 est la valeur courante du registre RI lorsque le retour en arrière est entrepris.

### 1.1.4.2 - Mise à jour du tableau de liaisons lors du retour terminal :

La plupart des implantations séquentielles de Prolog intègrent la transformation des appels terminaux, qui constitue une mesure d'économie de la pile locale. Cette transformation consiste à récupérer l'espace en pile locale, nécessaire à l'exécution d'une procédure (clause) déterministe, non pas à son retour, mais dès l'appel de son dernier littéral. Ainsi le bloc d'activation associé à l'appel du dernier littéral remplace le bloc initial. Par cette transformation, les récursions terminales peuvent s'exécuter dans un espace en pile locale, qui est constant. Deux conditions sont nécessaires pour pouvoir appliquer la transformation lors de l'appel d'un sous-but B se trouvant dans le corps d'une clause C activée par l'appel initial d'un but P. La première concerne la position terminale de B : il faut que B soit le dernier littéral de la clause C. La deuxième condition est l'inexistence de points de choix créés depuis l'appel du but P.

Sous les conditions énoncées ci-dessus, la même transformation peut être appliquée dans un modèle OU-parallèle multiséquentiel, et en particulier dans notre modèle.

Nous ne détaillons pas la mise en oeuvre d'une telle transformation, qui est étudiée dans [Boi 88], mais nous rappelons les deux notions nécessaires introduites, à savoir la sauvegarde des arguments du but dans des registres, et la notion de variables dangereuses [Boi 88].

Lors de l'application de la transformation terminale, le but à résoudre n'est plus capable d'accéder à ses arguments par référence à son bloc père, puisque ce dernier a disparu. Pour cela, les arguments d'un but doivent être sauvegardés, dès son appel, dans des registres spéciaux [War 80] (registres Ai introduits dans le chapitre I). De plus, cette sauvegarde ne devra laisser aucune référence vers le bloc initial soumis à récupération. Une telle référence ne peut avoir lieu si dans les arguments sauvegardés, il existe une variable libre ou une variable liée à une autre variable figurant dans le même environnement, une telle variable étant appelée une variable **dangereuse**. En effet, une variable libre est représentée comme étant liée à elle même: sa cellule contient sa propre adresse. Lors du chargement d'une telle variable dans un registre, celui-ci contient la référence de la cellule amenée à disparaître. Cette même situation peut se produire pour une variable apparaissant dans le but terminal, et qui est liée à une autre variable se trouvant dans le bloc détruit par la transformation terminale.

Pour pallier ce problème, une solution [War 83] consiste à déterminer dynamiquement les variables dangereuses et les ré-allouer sur la pile de recopie, lors de la sauvegarde des arguments en position terminale. Une autre solution [War 80] a été proposée lorsque la technique du partage de structures (voir chapitre I) est utilisé pour représenter les termes complexes. Elle consiste à déterminer statiquement les variables susceptibles d'être dangereuses : ces variables n'apparaissent pas dans les termes complexes, et figurent dans le dernier littéral sans apparaître en tête de clause. De telles variables sont systématiquement mémorisées dans la pile globale, puisque celle-ci n'est pas mise à jour lors du retour terminal.

Dans les deux solutions, la pile globale contient désormais, en plus des termes structurés (dans le cadre de la recopie de structures) ou des variables globales (dans le cadre du partage de structures), les variables dangereuses.

La généralisation de la transformation terminale aux tableaux de liaisons pose un problème dans notre modèle. En effet, lors d'un retour terminal, la récupération prématurée de toutes les cellules allouées dans le tableau de liaisons aux variables ayant une chronologie supérieure à celle du bloc amené à disparaître (y compris les variables de ce dernier), peut entraîner une perte irrémédiable d'informations. Ces informations sont les cellules dans le tableau de liaisons des variables allouées sur la pile globale. Ces variables sont celles qui figurent dans les termes structurés ou celles qui sont qualifiées de dangereuses.

**Exemple :**

Considérons un processeur qui résout le but courant *p(a, b)* dans la base suivante :

(1) p(X,Y):- q(Z,T), t(X,Y,Z,T).
(2) q(c, X1).
(3) t(a,b,c,d).
(4) t(a,a,a,a).

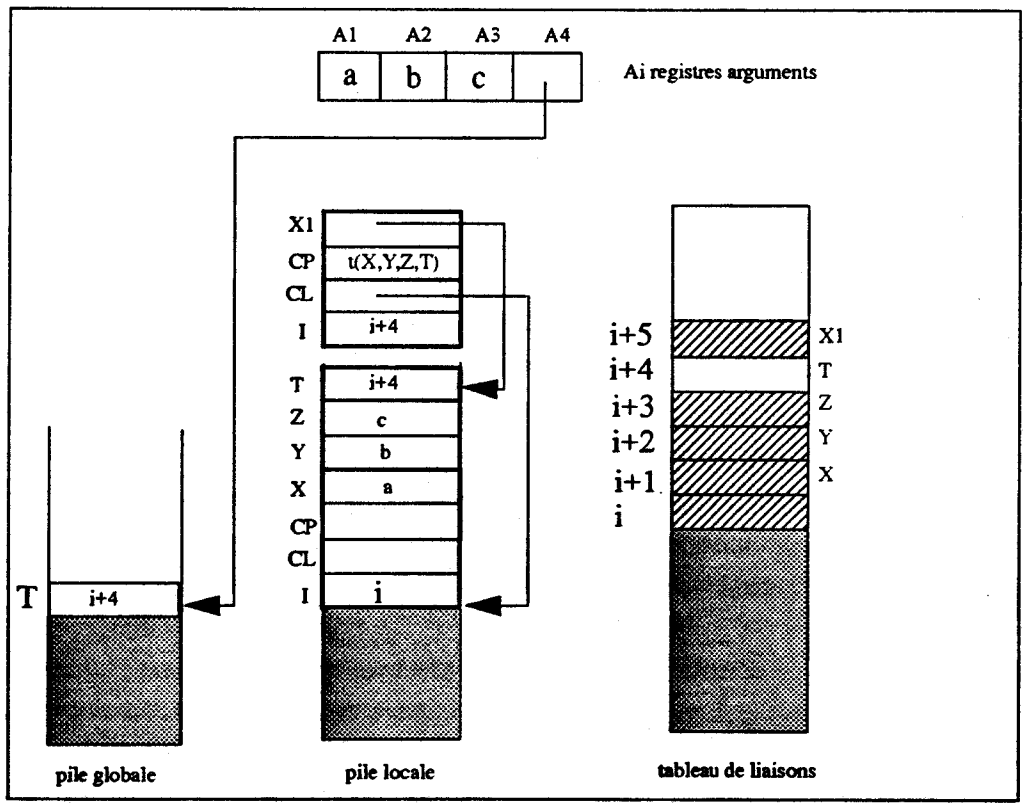



figure 4.10 : Retour terminal

La figure 4.10 décrit l'état de liaisons après l'unification du but *q(Z,T)* avec le fait *q(c,X1)*. Deux blocs d'activation ont été créés sur la pile locale. Le premier, associé à l'activation de la clause (1), possède un environnement de taille quatre (*X, Y, Z et T*). Le deuxième environnement, associé à l'activation du fait *q(c,X1)*, est de taille un (X1).

La partie en gris représente les environnements précédant l'appel du but p(a,b). Nous avons considéré uniquement la démonstration de ce but.

A chaque variable créée sur la pile locale, est associée une cellule dans le tableau de liaisons, qui est destinée à contenir une éventuelle liaison conditionnelle de la variable. Les cellules hachurées sont inutilisées puisque les variables correspondantes sont liées de façon non conditionnelle; le mécanisme de liaison superficielle a été utilisé suivant le mécanisme de liaison décrit dans le chapitre précédent.

Au retour du but *q(Z,T)*, son bloc disparaît de la pile locale. La transformation de l'appel terminal s'applique donc au but *t(X,Y,Z,T)*. Avant de mettre à jour la pile locale, il faut sauvegarder les arguments de cet appel dans les registres prévus à cet effet. La variable *T* étant dangereuse, elle est recopiée sur la pile globale. La récupération de l'espace alloué en pile locale de l'environnement associé à l'activation de la clause (1), peut être effectuée sans problème, mais la récupération des cellules allouées aux variables de cet environnement dans le tableau de liaisons, entraîne la disparition de la cellule *i+4* qui est référencée dans la pile globale.

Pour remédier à ce problème, nous avons distingué deux solutions dont l'une est statique, l'autre dynamique.

**Solution statique :**

Elle consiste à scinder le tableau de liaisons (ou utiliser deux tableaux de liaisons) de chaque processeur en deux parties. Une partie, dite locale, est destinée à contenir les cellules des variables locales. L'autre partie, dite globale, contient toutes les variables qui sont susceptibles d'être recopiées sur la pile globale. Ces dernières variables peuvent être déterminées statiquement. En effet, elles apparaissent dans les termes structurés, ou dans le dernier littéral et pas en tête de clause.

Lors du retour terminal, seule la partie locale du tableau de liaisons est mise à jour en récupérant les cellules des variables locales de l'environnement amené à disparaître. La partie globale est mise à jour lors du retour en arrière.

Pour mettre en oeuvre cette solution, chaque processeur gère deux compteurs de variables **RIL** et **RIG**, qui mémorisent respectivement le nombre de variables locales et le nombre de variables globales, créées sur la branche traitée par le processeur.

Lorsqu'une variable locale (respectivement globale) est créée, sa cellule sur la pile locale (respectivement globale) contient la valeur courante de RIL (respectivement RIG). En raison de ces modifications, le champ I de chaque bloc d'activation contient désormais deux informations (champs **IL** et **IG**) qui sont les deux valeurs courantes, lors de la création de ce bloc, des compteurs de variables RIL et RIG.

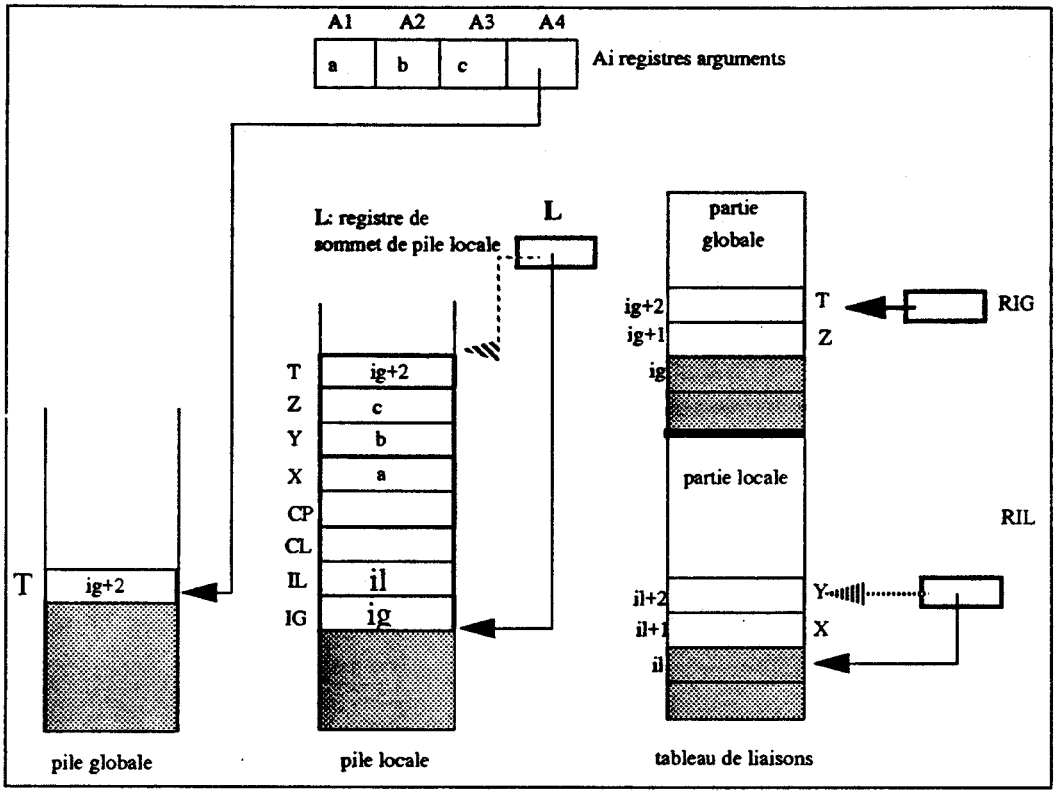



figure 4.11 : Mise à jour du tableau de liaisons lors du retour terminal (solution statique)

Reprenons l'exemple de la figure 4.10 en tenant compte des considérations introduites dans la solution statique.

La figure 4.11 montre l'état des piles et du tableau de liaisons après avoir effectué la transformation de l'appel terminal dans la pile locale et dans le tableau de liaisons. Les variables *Z* et *T* étant susceptibles d'être dangereuses, deux index (*ig+1* et ig+2) leurs sont respectivement associés, dès leur création, dans la partie globale du tableau de liaison.

La récupération de l'espace, lors du retour terminal, s'effectue en mettant à jour le registre L qui mémorise le sommet courant de la pile locale, ainsi que le registre RIL.

Le champ IG peut paraître inutile, puisque lors du retour terminal, seul le registre RIL est chargé par la valeur mémorisée dans le champ IL du bloc amené à disparaître. Mais il est indispensable lorsqu'il s'agit d'un bloc de choix.

En effet, compte tenu des spécifications du modèle de calcul, chaque bloc de choix doit mémoriser le nombre total des variables créées sur la branche sur laquelle apparait le point de choix, afin de faire hériter de ce nombre, un processeur prenant en charge une alternative de ce point de choix.

**Solution dynamique :**

Cette solution consiste également à diviser le tableau de liaisons en deux parties dites globale et locale. Mais, contrairement à la solution précédente, l'allocation des cellules dans la partie globale n'a lieu que lorsqu'une variable allouée sur la pile locale doit être recopiée sur la pile globale. Une variable est recopiée sur la pile globale dans le cas où elle est dangereuse, ou lors de recopie (sur la pile globale) d'un terme structuré où apparait cette variable.

Lorsqu'une variable est créée, l'index courant mémorisé dans le registre RI est associé à la variable. A la création, aucune distinction n'est faite entre les variables. Mais lorsqu'une variable est amenée à être recopiée sur la pile globale, un nouvel index lui est attribué, celui-ci référençant cette fois une cellule libre dans la partie globale du tableau de liaisons.

Reprenons l'exemple précédent en appliquant cette solution :

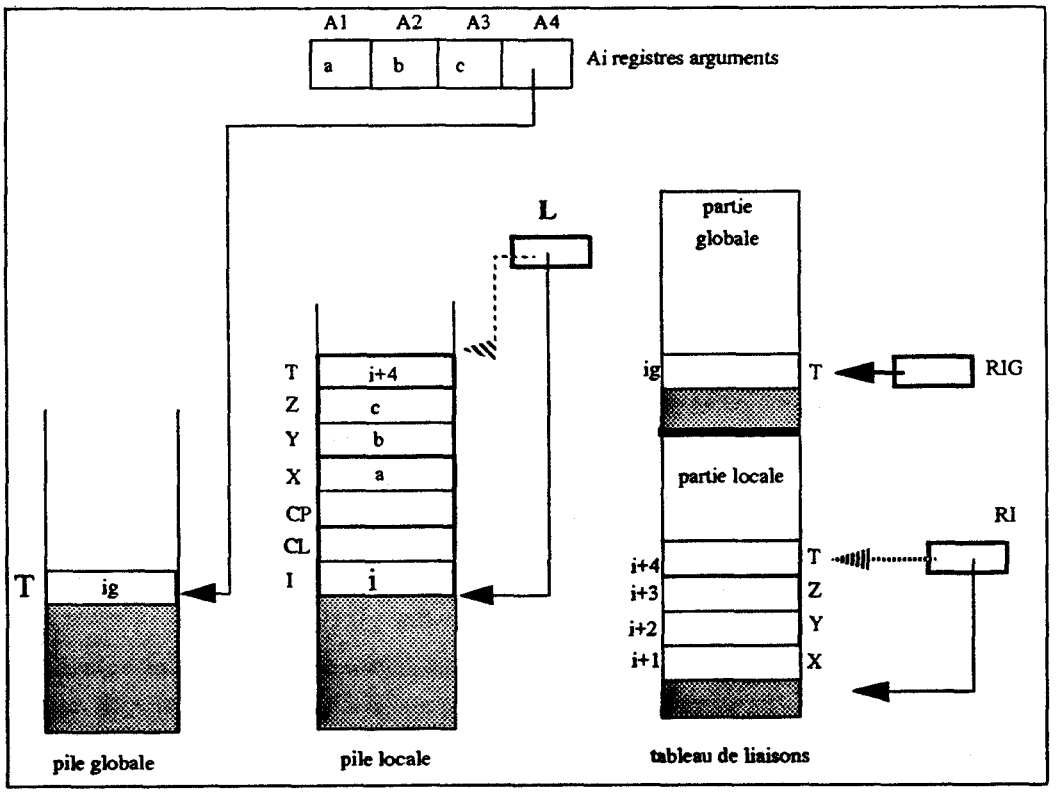



figure 4.12 : Mise a jour du tableau de liaisons lors du retour terminal (solution dynamique)

Lors de l'activation de la règle (1), quatre cellules (*i+1, i+2, i+3, i+4*) ont été réservées dans le tableau de liaisons pour accueillir les éventuelles liaisons conditionnelles des variables X, Y, Z et T figurant dans la clause. Lors du retour terminal, la variable T étant dangereuse, elle est recopiée sur la pile globale avec un nouvel index ig (valeur courante du registre RIG), aussi la partie locale du tableau de liaisons peut-elle être mise à jour. A la différence de la solution statique, seule la variable T est effectivement dangereuse et donc recopiée.

**Choix de solution**

La solution dynamique consiste à attribuer systématiquement, à toute variable risquant de poser problème pour l'application de la transformation terminale, un index dans la partie globale du tableau des liaisons. Dans la solution dynamique, la même attribution est effectuée que lorsque la variable considéré pose réellement problème.

On retrouve ici la même différence qui caractérise les deux approches de représentation des termes structurés (paragraphe 2.4) : une vision statique pour le partage de structures, et une vision dynamique pour la recopie de structures.

Notre choix s'est porté sur la solution dynamique afin de respecter notre volonté initiale d'utiliser la Machine Abstraite de Warren comme base d'implantation du modèle parallèle. En effet, celle-ci utilise, dans le cadre d'une représentation des termes, la recopie de structures et de ce fait, aucune analyse statique n'est faite pour déterminer les variables globales.

### 1.1.4.3 - Classification des variables :

Nous avons considéré, jusqu'à présent, que toute occurrence d'une variable figurant dans une clause activée, est accompagnée par l'allocation de deux cellules : celle d'entre elles se trouvant sur la pile locale contient un index, l'autre est créée dans le tableau de liaisons pour contenir une éventuelle valeur de liaison.

Dans une implantation séquentielle, il est intéressant, pour une économie de l'espace en pile locale, de distinguer les variables temporaires des variables permanentes.

variables temporaires :

Une variable est dite temporaire si elle vérifie les deux conditions suivantes [Boi 88] :

**C1 :** ne pas apparaître dans plus d'un littéral.
**C2 :** être telle que l'on puisse déterminer statiquement si sa première occurrence est obligatoirement liée.

Une variable vérifiant C1 et apparaissant en tête de clause est nécessairement une variable temporaire, puisque lors de l'activation de la clause, la variable est soit instanciée avec une constante ou un terme structuré, soit liée à une autre variable (la liaison de deux variables s'effectue toujours en liant celle dont la chronologie est supérieure).

De même, toute variable vérifiant C1, et dont la première occurrence apparait dans l'un des termes structurés d'un littéral situé en partie droite d'une clause, est temporaire. En effet, lors de l'appel de ce but, le chargement de ses arguments entraîne la recopie du terme et donc la variable est liée (la recopie d'une variable est effectuée par l'allocation d'une nouvelle variable sur la pile globale en liant la première à cette dernière).

Une variable vérifiant la condition C1, et figurant dans le dernier littéral d'une clause, est également considérée comme temporaire. En effet, si une telle variable est restée libre, il faudra de toute façon la ré-allouer dynamiquement sur la pile globale (paragraphe 1.1.4.b).

Une définition mettant en évidence les exemples de variables temporaires ci-dessus est donnée dans [War 83] : une variable temporaire est une variable dont la première occurrence figure dans la tête de clause, ou dans un terme structuré, ou dans le dernier but, et qui n'apparait que dans un seul but, tête et premier littéral de la partie droite comptant pour un.

variables permanentes :

Une variable permanente est une variable qui n'est pas temporaire.

Compte-tenu de cette classification des variables, seul un environnement associé aux variables permanentes est alloué sur la pile locale lors de l'activation d'une clause. Les variables temporaires sont gérées directement par le biais des registres arguments Ai.

Les répercutions dues à cette classification des variables sur la pile locale, ont une influence, dans le cadre de notre modèle parallèle, sur les tableaux de liaisons, puisque l'utilisation de ces derniers est étroitement liée aux piles locales. En effet, à chaque variable créée sur la pile locale, est associé un index dans le tableau de liaisons. Par conséquent, les variables temporaires ne possèdent plus d'entrée dans le tableau de liaisons. Mais cela ne remet pas en cause notre mécanisme de liaison, puisqu'une telle variable ne peut faire l'objet d'une liaison conditionnelle.

La notion de variables temporaires peut se généraliser pour les tableaux de liaisons. En effet, seules les variables susceptibles d'être liées de façon conditionnelle possèdent une entrée dans le tableau de liaisons. Ces variables sont toutes les variables temporaires définies précédemment, plus les variables apparaissant en tête de clause. En effet, une variable apparaissant en tête de clause est forcément liée lors de l'unification des arguments du but avec ceux figurant dans la tête de clause. Une variable apparaissant en tête de clause mais pas dans un terme structuré, est soit instanciée, soit liée (une liaison de deux variables est effectuée en liant la plus récente à la plus ancienne), lors de l'unification. Une variable figurant dans un terme structuré en position d'argument, est forcément liée à l'unification. En effet, une telle variable est soit liée, soit recopiée sur la pile globale; dans ce dernier cas, elle subit une liaison vers sa nouvelle instance.

Nous avons montré dans ce paragraphe, que les techniques d'optimisation de l'espace en pile locale, utilisées dans les implantations séquentielles, peuvent être généralisées aux tableaux de liaisons. Nous n'avons pas évoqué le ***trimming*** puisque celui-ci n'est qu'une généralisation de la transformation de l'appel terminal. Il est applicable sous les mêmes conditions qu'une transformation terminale. Seule la proportion des variables dangereuses est renforcée puisque le problème d'une variable référençant une partie d'environnement appelée à être récupéré, ne se pose plus uniquement pour le dernier littéral, mais à chaque appel de but pouvant faire diminuer la taille d'environnement.

La division du tableau de liaisons en deux parties globale et locale, et l'attribution (dans la solution dynamique) d'un nouvel index à une variable dangereuse, dans le tableau de liaisons, permettent de généraliser le *trimming* aux tableaux de liaisons.

### 1.1.5 - Unification :

Avant de présenter en détails les opérations de liaison et de déréférencement qui constituent les instructions essentielles de l'unification, nous donnons tout d'abord la représentation des objets manipulés par la machine abstraite parallèle. Nous distinguons deux types d'objets : les objets représentés sur les piles locale et globale d'une part, et les objets créés dans les tableaux de liaisons, d'autre part.

Objets créés sur les piles : Chaque objet est composé d'une étiquette et d'une valeur, l'étiquette désignant le type de l'objet :

| objets | étiquette | valeur |
|---|---|---|
| variable | | |
| . libre | LIBRE | un index du tableau de liaisons |
| . liée de façon conditionnelle | CND | un index du tableau de liaisons |
| . liée | REF | adresse de la variable |
| constante | ATOM | une référence vers la table des constantes |
| liste | LIST | une référence vers la représentation de la liste |
| terme fonctionnel | STR | une référence vers la représentation du terme |

Nous entendons par variable libre, une variable qui n'a subi aucune liaison. L'étiquette *LIBRE* permet d'identifier une telle variable afin d'éviter de parcourir inutilement les différents tableaux de liaisons lors du déréférencement de cette variable. Dès qu'une variable est liée conditionnellement pour la première fois, l'étiquette devient l'étiquette *CND*.

**Objets alloués dans les tableaux de liaisons :**

Chaque élément du tableau d'un processeur est composé de deux champs : le premier contient l'étiquette de la liaison, le second, ou champ *valeur*, contient la valeur de liaison. Cette dernière est un objet de type *REF, ATOM, LIST* ou *STR*. L'étiquette d'une liaison est la clé du processeur, lors de la liaison.

Durant l'unification de deux termes, une liaison doit être effectuée quand l'un de ces termes au moins est une variable libre. Dans une implantation séquentielle, une liaison est effectuée en écrivant dans la cellule de la variable libre (liaison superficielle), la valeur du deuxième terme. Dans le cas d'une liaison de deux variables, la cellule de la plus récente variable contient une référence vers la cellule représentant l'autre variable libre. Ainsi l'accès à la valeur de liaison d'une variable (opération de déréférencement) nécessite dans certains cas le parcours d'une chaîne de liaisons, l'extrémité de cette chaîne étant la valeur de liaison. Comme nous l'avons mentionné précédemment, ce mécanisme de liaison est efficace dans une implantation séquentielle, mais il est insuffisant dans une implantation OU-parallèle de Prolog. Toutefois, il est intéressant d'utiliser un tel mécanisme, tant que cela est possible. C'est pourquoi, il a été nécessaire, dans le cas de notre modèle parallèle, de distinguer les liaisons conditionnelles et les liaisons non conditionnelles, d'une part, et les liaisons locales et non locales d'autre part.

### 1.1.5.1 - Opération de liaison :

Avant d'examiner tous les cas possibles de liaisons de deux termes, rappelons tout d'abord la notion de liaison locale : une liaison est dite locale lorsque la variable liée a été créée par le processeur effectuant la liaison.

Une liaison non locale a lieu lorsqu'un processeur est amené à lier une variable créée par un autre processeur. Une liaison non locale effectuée par un processeur est forcément une liaison conditionnelle : en effet, il existe au moins un point de choix qui a été créé après l'occurrence de la variable à lier. Ce point de choix est le père de la branche traitée par le processeur considéré.

#### Liaison non conditionnelle :

Considérons un processeur Pi qui lie une variable locale X à un terme te :

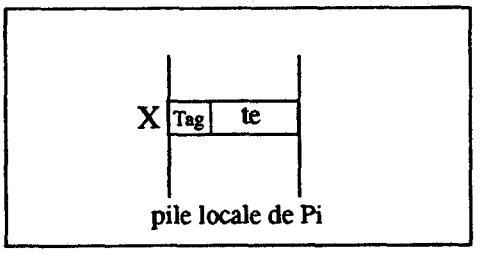



figure 4.13 : Liaison non conditionnelle

Lors d'une liaison non conditionnelle (il s'agit nécessairement d'une liaison locale), le mécanisme de liaison superficiel est utilisé en écrivant la valeur de liaison directement dans la zone représentant la variable.

La variable X ayant subi une liaison non conditionnelle, sa cellule contient une étiquette *Tag* qui désigne le type du terme te *(REF, ATOM, LIST, STR)*, ainsi qu'une référence vers la zone qui représente le terme te; cette zone peut être située dans la table des constantes si le terme te est une constante, ou dans une pile locale (respectivement globale) d'un processeur quelconque si le terme te est une variable (respectivement un terme structuré).

**Liaison conditionnelle :**

Une variable liée conditionnellement peut faire l'objet de liaisons multiples : le mécanisme de liaison profonde est donc utilisé. La cellule d'une variable liée conditionnellement contient un index dans le tableau de liaisons.

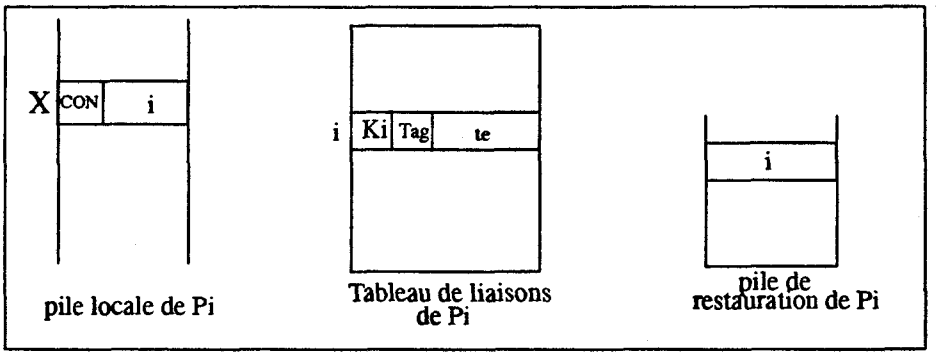



figure 4.14 : Liaison locale conditionnelle

La variable *X* (fig. 4.14) étant liée conditionnellement par le processeur *Pi*, le champ étiquette prend la valeur *CND*. Une liaison profonde est créée dans le tableau de liaison à l'index i mémorisé dans la cellule de la variable *X*. La liaison est étiquetée par la clé courante *Ki* du processeur *Pi*. La pile de restauration contient désormais les index des variables liées de façon conditionnelle, au lieu de leurs adresses, afin d'éviter les indirections au cours de leur restauration lors du retour en arrière.

Pour détecter une liaison conditionnelle, le processeur vérifie si la variable à lier est non locale, auquel cas il s'agit nécessairement d'une liaison conditionnelle. Dans le cas contraire, le processeur compare l'âge de la variable avec celui du dernier point de choix créé par le processeur. Ce test s'effectue par simple comparaison de l'adresse de la variable, à celle du dernier point de choix (cette dernière étant mémorisée dans le registre *BL* du processeur) si la variable est allouée sur la pile locale, ou au sommet de la pile globale qui était courant lors de la création du dernier point de choix (celui-ci est mémorisé dans le registre BG du processeur).

### 1.1.5.2 - Déréférencement :

L'opération de déréférencement a lieu dans les trois cas suivants :

**lors de l'unification :** si parmi les deux termes à unifier il existe une variable, il faut d'abord s'assurer qu'il n'existe pas une éventuelle liaison de la variable avant de lier celle-ci.

**lors de la recopie d'un variable dangereuse :** avant de recopier une variable pouvant être dangereuse sur la pile globale (afin de pouvoir appliquer la transformation terminale), il faut vérifier que la variable est demeurée libre lors du retour terminal.

**lors du chargement des registres arguments :** si une variable apparait en position d'argument du but appelé, le registre adéquat doit être chargé par l'éventuelle valeur de liaison de cette variable. Si la variable apparait dans un terme structuré en position d'argument du but appelé, elle doit être également déréférencée lors de la recopie de ce terme (chaque terme en position d'argument est tout d'abord recopié en sommet de pile globale; le registre Ai est chargé par une référence vers la nouvelle instance de terme ainsi créée).

En raison des spécifications précédentes de l'opération de liaison, le déréférencement d'une liaison non conditionnelle, est similaire à celui de la WAM. Une liaison non conditionnelle est identifiée par l'une des étiquettes : *REF, ATOM, LIST, STR*. La chaîne de liaisons est suivie jusqu'à ce qu'une liaison ayant une étiquette autre que *REF* soit rencontrée.

Lors d'un déréférencement, par un processeur Pi, d'une variable liée conditionnellement, trois cas peuvent se produire :

**La variable est créée lors de la tâche courante :**

Pour déréférencer une telle variable, un premier accès a lieu directement dans le tableau de liaison en utilisant l'index de la variable. Si une liaison a été enregistrée dans le tableau, elle est nécessairement valide : l'objet ainsi trouvé est le résultat du déréférencement (s'il s'agit d'une variable, le déréférencement se poursuit avec cette nouvelle variable). Dans le cas contraire, la variable est restée libre.

Pour vérifier qu'une variable est créée lors de la tâche courante, le processeur doit être capable de déterminer si la variable a été allouée sur l'un des segments de pile locale ou de pile globale associés à la tâche courante. Pour cela l'adresse de la variable est comparée aux valeurs des registres BS (base du segment de pile locale associé à la tâche courante) et BGS (base du segment de pile globale associé à la tâche courante).

**La variable est créée lors d'une tâche suspendue :**

A la différence du déréférencement d'une variable créée lors de la tâche courante, lorsqu'aucune liaison n'a été enregistrée dans le tableau du processeur Pi, la variable en question n'est pas forcément libre, et peut avoir subi une liaison, valide pour Pi, par un autre processeur. Dans ce cas, le processeur Pi doit effectuer la recherche d'une liaison valide enregistrée par l'un des processeurs. La recherche s'effectue en utilisant l'index de la variable pour accéder aux éventuelles liaisons effectuées par les différents processeurs. Pour valider une

liaison effectuée par un processeur Pj, le processeur Pi compare l'étiquette de la liaison à la clé Kj, cette dernière étant mémorisée à la base du segment de pile associé à la tâche courante (voir paragraphe 1.1.1.2 - page 95).

**La variable est non locale au processeur :**

Lors d'un déréférencement d'une variable non locale, le processeur effectue un accès direct à la cellule représentant la variable pour vérifier si une liaison non conditionnelle a été enregistrée; si une telle liaison existe et est étiquetée par *REF, ATOM, LIST* ou *STR*, le déréférencement est similaire à celui de la Wam. Si l'étiquette est *LIBRE*, alors la variable n'a subi aucune liaison : elle est donc libre. Dans le cas contraire (i.e. l'étiquette est *CND*), le processeur doit effectuer la recherche d'une éventuelle liaison valide dans les différents tableaux de liaisons.

Il est possible de faire coïncider le déréférencement dans les deux derniers cas, si l'on considère que lorsqu'un processeur est amené à effectuer la recherche d'une liaison valide dans les différents tableaux de liaisons, la première recherche a lieu dans le tableau du processeur ayant créé la variable à déréférencer. De ce fait, aucune différence n'est faite entre une variable non locale et une variable locale créée lors d'une tâche suspendue, en ce qui concerne leur déréférencement.

## 1.2 - Mémoire privée :

La mémoire privée constitue l'autre zone de travail d'un processeur. Elle contient toutes les données qui ne peuvent être partagées avec les autres processeurs. Dans le cadre de notre modèle, la pile de restauration et les registres abstraits d'un processeur peuvent être rangés dans la mémoire privée de ce dernier.

Dans une implantation séquentielle, le rôle de la pile de restauration est de mémoriser, en cours de démonstration, l'historique des liaisons conditionnelles effectuées sur une branche de l'arbre de recherche. Elle est mise à jour lors du retour arrière en défaisant les liaisons créées depuis le dernier point de choix. Le champ TR de ce point de choix indique la borne jusqu'où doit s'effectuer la mise à jour.

La définition de la pile de restauration peut se généraliser à un processeur en raison de la gestion des processus par pile. Dans la machine abstraite parallèle, la pile de restauration contient tous les index des variables (et non plus leurs adresses) liées conditionnellement lors de l'ensemble des tâches entreprises par le processeur.

## 1.3 - Zone code :

La zone code de la mémoire virtuelle de la machine abstraite parallèle sert à représenter les procédures Prolog sous forme compilée.

Dans cette partie, nous n'abordons pas les aspects de compilation que nous n'avons pas étudiés. Nous donnons uniquement la liste des instructions de la WAM qui doivent être modifiées en spécifiant leurs nouvelles fonctionnalités.

### 1.3.1 - Instructions de gestion des points de choix :

Ces instructions correspondent aux instructions de la WAM : *try_me_else EtiqCL, retry_me_else EtiqCL et trust_me_else fail.*

***try_me_else EtiqCL :***

Cette instruction précède la première règle d'un paquet de clause. Elle est forcément exécutée par le processeur rencontrant le point de choix. Ainsi le processeur crée une partie *Reprise* sur sa pile locale, en donnant la valeur *EtiqCL* au champ BP, et met à jour les registres BL et BG. La paire formée par *EtiqCL* et le numéro de la tâche courante (contenu du registre BS) est empilée sur la pile de choix du processeur.

***retry_me_else EtiqCL :***

Cette instruction peut être exécutée, soit par le processeur créant le point de choix lors du retour en arrière, soit par un processeur prenant en charge la première alternative du paquet mis en attente. Une telle instruction doit être exécutée en exclusion mutuelle.
Dans les deux cas, le champ BP du dernier point de choix est mis à jour avec *EtiqCL*; de même, le champ Lpr est-il mis à jour en positionnant le bit adéquat à 1 (voir paragraphe 1.1.1.2 - page 95).

***trust_me_else_fail :***

Cette instruction précède la dernière clause du paquet à essayer. Dans une stratégie séquentielle, son exécution entraîne la disparition du dernier point de choix. Comme nous l'avons mentionné précédemment, une telle mise à jour risque de détruire prématurément certaines données encore utilisées par les processeurs explorant des branches issues du dernier point de choix.
L'exécution de l'instruction *trust_me_else fail* par un processeur oisif prenant en charge cette dernière alternative, consiste à mettre à jour les champs BP et Lpr du point de choix.
L'exécution de l'instruction *trust_me_else fail* par le processeur créateur du point de choix, consiste tout d'abord à vérifier si toutes les alternatives sont entièrement explorées, ce contrôle s'effectuant en vérifiant le champ Lpr du point de choix. Si ce champ est nul (c'est à dire tous les bits sont positionnés à zéro), le processeur procède à la destruction du point de choix en mettant à jour les registres BL, BG, RI et RIG, ainsi que la clé privée Ki, en la décrémentant. Dans le cas contraire, seuls les champs BP et Lpr sont mis à jour.

### 1.3.2 - Instructions de contrôle :

Seules les instructions permettant d'allouer ou désallouer des environnements en pile locale doivent être étendues en raison de l'utilisation des tableaux de liaisons.

***allocate*** :

crée un nouveau bloc en sommet de pile locale; pour cela, cette instruction alloue un environnement formé des variables permanentes apparaissant dans la clause courante, et sauve les valeurs courantes des registres CL, CP (registres de continuation), RI (registre d'index) dans les champs *CL*, *CP* et *I* du bloc ainsi créé.

***deallocate*** :

restaure les registres de Continuation, le registre RI et applique la transformation terminale.

### 1.3.3 - Instructions d'unification et de chargement des registres arguments :

Dans la WAM, les instructions d'unification *get* et *unify* et les instructions *put* qui permettent de charger les registres arguments, font appel aux opérations de liaison et de déréférencement. Elles doivent donc être étendues en raison du nouveau mécanisme de liaison.

#### Instructions d'unification :

***get_const C, Ai*** (respectivement ***get_nil, Ai)*** :

cette instruction effectue une liaison si Ai contient la référence d'une variable. Si une telle référence existe, les mécanismes de liaison superficielle ou profonde sont utilisés selon qu'il s'agit d'une liaison conditionnelle ou non. Si Ai n'est pas une variable, une simple comparaison est effectuée entre la constante C (respectivement la liste vide) et la valeur du registre Ai pour déterminer si les deux termes sont unifiables.

***get_variable Vn, Ai*** :

quelle que soit la valeur de Ai, cette instruction effectue une liaison de la variable Vn. S'il s'agit d'une liaison non conditionnelle, la liaison superficielle est utilisée. Dans le cas contraire, la liaison profonde est utilisée.

***get_value Vn, Ai :***

cette instruction effectue tout d'abord le déréférencement de la variable Vn. Le déréférencement est effectué comme nous l'avons spécifié dans le paragraphe 1.1.5.b du présent chapitre. Si le déréférencement de Vn détermine que Vn est libre ou liée à une variable Yn, alors une liaison de Vn ou de Yn est effectuée selon le principe décrit dans le paragraphe 1.1.5.a de ce chapitre. Dans le cas contraire, si le registre Ai contient une variable, alors celle-ci est liée à la valeur de liaison de Vn.

***get_list Ai*** **et** ***get_structure F,Ai :***

ces instructions font appel à l'opération de recopie de terme lorsque le registre Ai contient une variable. La recopie d'une variable figurant dans un terme structuré, s'effectue en attribuant un nouvel un index à la nouvelle instance de la variable.

**Exemple :**

Considérons l'état des piles d'un processeur Pi (figure 4.15) sur le point d'effectuer la recopie de la variable X :

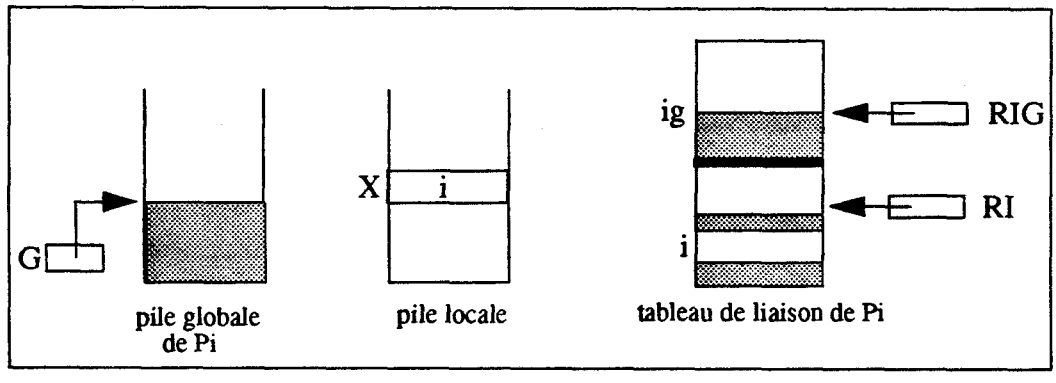



figure 4.15 : Etat des piles avant la recopie

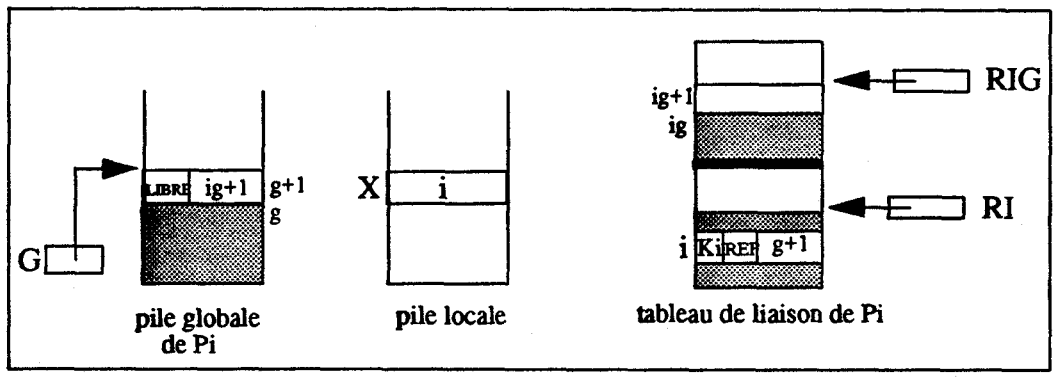



figure 4.16 : Recopie d'une variable en utilisant la liaison profonde

La variable X doit être recopiée : une nouvelle variable a été créée sur la pile globale, un nouvel index lui est associé dans le tableau de liaison. La duplication de la variable X se traduit par une liaison de celle-ci à la variable créée sur la pile globale.

Cette liaison peut être une liaison conditionnelle si un point de choix a été créé sur la pile locale, après l'apparition de la variable. Cette situation qui est celle de la figure 4.16, a entraîné l'utilisation du mécanisme de liaison profonde.

Dans le cas contraire, la liaison de la variable X avec sa nouvelle instance s'effectue en utilisant le mécanisme de liaison superficielle comme le montre la figure ci-contre :

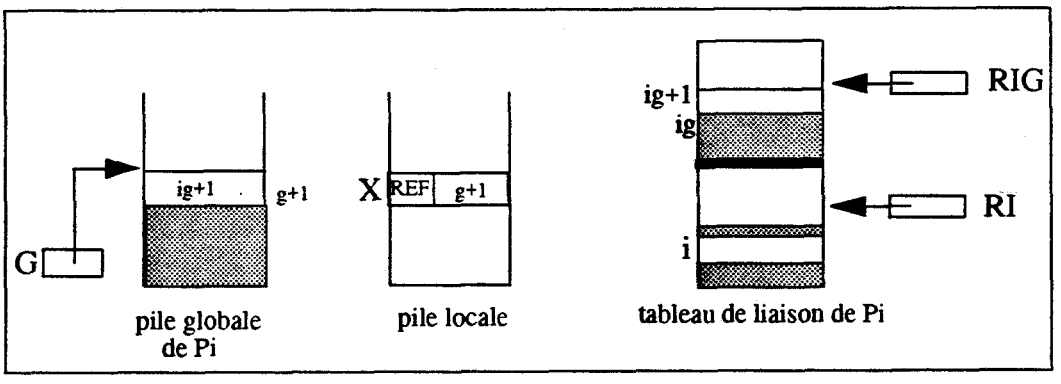



figure 4.17 : Recopie d'une variable en utilisant la liaison superficielle

**Instructions de chargement des registres arguments :**

Seules les instructions qui font appel au déréférencement et à la recopie doivent être modifiées :

***put_list Ai*** **et** ***put_structure F, Ai :***

Lors de la sauvegarde d'un terme structuré dans un registre, une nouvelle instance de ce terme est créée sur la pile globale. Dans ce cas, les variables figurant dans le terme doivent être recopiées selon la méthode décrite ci-dessus.

## 2 - Gestion des activités des processeurs :

Dans cette partie, nous abordons les notions qui n'apparaissent pas dans une machine abstraite séquentielle comme la WAM, puisque liées à la nature parallèle du système.

Une machine abstraite séquentielle n'a pas à spécifier la gestion du processeur. Celle-ci est à la charge du système d'exploitation en raison de la linéarité de l'exécution du programme. Une telle approche peut être utilisée, mais le système d'exploitation constitue alors un contrôle centralisé, ce qui va à l'encontre de notre modèle de calcul, où chaque processeur est capable de gérer lui même ses activités.

Au cours de l'exécution d'un programme, l'activité de chaque processeur peut être amenée à passer par quatre états, en raison de cette autonomie du processeur. Au départ, un seul processeur est à l'état **actif**, les autres processeurs sont à l'état **oisif**. Dès qu'une alternative est trouvée par un processeur oisif, celui ci passe, après initialisation, à l'état actif.

Un processeur actif peut être amené à passer dans l'état **bloqué**. Cette situation se produit lorsque, ne pouvant effectuer le retour arrière, parce qu'une branche issue du point de choix à détruire n'est pas terminée, le processeur décide d'aller aider à l'exploration de cette branche. L'état bloqué est similaire à l'état oisif, puisque dans les deux cas le processeur est à la recherche d'un travail. La différence entre ces deux états concerne les zones de recherche de travail disponible. En effet, dans l'état bloqué, la recherche est limitée au sous-arbre dont la racine est le dernier point de choix.

Enfin, dans le dernier état, le processeur est **suspendu**. Nous verrons les raisons de suspension d'un processeur lorsque seront abordés les effets de bord dans le chapitre suivant.

### 2.1 - Recherche de travail :

La recherche d'un travail disponible par un processeur, consiste à explorer les piles de choix des processeurs actifs. L'allocation statique des différentes piles de choix permet à un processeur d'y accéder directement. La stratégie de l'exploration de ces piles par un processeur oisif peut être quelconque. Elle n'est prise en compte que lorsque l'efficacité du parallélisme est considérée. Nous discuterons ce dernier point dans le chapitre suivant.

## 2.2 - Initialisation d'une branche :

Lorsqu'un processeur recherchant du travail rencontre un point de choix possèdant une alternative en attente, une nouvelle tâche est créée.

Le processeur doit, dans un premier temps, protéger le point de choix par une exclusion mutuelle, à cause des accès concurrents de la part des autres processeurs oisifs et du processeur créateur de ce point de choix.

Dans un deuxième temps, le processeur recopie l'ensemble des informations nécessaires pour continuer la résolution avec l'alternative ainsi trouvée. Pour cela, il recopie les informations suivantes :

le jeu de clés du processeur fournissant le travail :

le paragraphe 1.1.1.2. décrit cette recopie.

les arguments du but courant :

la recopie des arguments du but courant est confondue avec le chargement des registres arguments du processeur. Nous rappelons que ces arguments ont été sauvegardés dans le point de choix.

la continuation :

les registres CL et CP sont chargés directement à partir des champs *CL* et *CP* du point de choix;

les index courants au point de choix :

les registres RI et RIG sont initialisés par les valeurs mémorisées dans les champs *I* et *IG* du point de choix. Les valeurs de *I* et *IG* sont respectivement les index courants dans la partie locale et la partie globale du tableau de liaisons du processeur ayant créé le point de choix;

l'adresse du point de choix :

le registre BL du processeur est initialisé avec l'adresse du point de choix (cette adresse a permis au processeur oisif l'accès au point de choix).

## 2.3 - Synchronisation et interaction des processeurs :

L'interaction des processeurs se traduit seulement par les accès concurrents à l'ensemble des données partagées. C'est pourquoi certaines zones doivent être protégées par exclusion mutuelle. Ces zones sont constituées uniquement par les points de choix qui font l'objet d'écritures multiples lors de l'initialisation d'une branche parallèle. Le processeur prenant en charge cette branche doit mettre à jour le champ *BP q*ui indique les alternatives restantes, ainsi que le champ *Lpr* qui permet d'identifier les processeurs traitant des alternatives issues du point de choix considéré.

Le modèle de calcul parallèle introduit deux types de synchronisation. La première, inhérente à l'utilisation du partage de l'environnement, peut avoir lieu lorsqu'un processeur effectue une tentative d'accès à une section critique. La deuxième a lieu lorsqu'un processeur effectue le retour arrière vers un point de choix dont toutes les alternatives ont été commencées mais non terminées. Pour réduire les déperditions dues à cette synchronisation, nous avons opté pour la solution décrite dans le paragraphe 1.1.2 du chapitre III.

Dans ce chapitre nous avons décrit une machine abstraite parallèle où chaque processeur actif s'identifie à la machine abstraite de Warren ("modulo les extensions introduites").

Les caractéristiques du modèle de calcul ont permis une gestion des processus par pile, chaque processeur utilisant sa pile locale pour gérer les tâches qu'il entreprend. De ce fait, la zone de travail de chaque processeur peut être gérée en utilisant les techniques efficaces de gestion de la mémoire dans une implantation séquentielle, à savoir la transformation des appels terminaux et le trimming. Nous avons montré que ces dernières techniques peuvent être également adaptées à la gestion de l'espace dans les tableaux de liaisons, structures de données supplémentaires introduites par le modèle de calcul.

# -=- Chapitre V -=-

*Gestion des effets de bord*

# 1 - Prédicats à effets de bord :

Si l'on considère que Prolog est un langage de programmation à part entière, il est impossible d'ignorer les effets de bord, d'autant plus que l'utilisation généralisée d'effets de bord pose des problèmes dans un système OU-parallèle multiséquentiel.

La difficulté à utiliser certains prédicats extra-logiques comme les prédicats *I/O*, *cut*, *assert* et *retract*, est due au fait que leur sémantique dépend fortement de la stratégie séquentielle e*n profondeur d'abord.*

Les solutions qui ont été proposées pour prendre en considération les effets de bord dans un système logique parallèle, peuvent être classées selon deux approches différentes. Le principe de la première approche est de faire la distinction entre les prédicats qui peuvent être exécutés en parallèle et ceux qui doivent être exécutés en séquentiel : tout prédicat produisant un effet de bord est déclaré séquentiel. Cette solution est simple à mettre en oeuvre, mais elle nécessite un style de programmation bien particulier, ou l'utilisation de dialectes par lequel le programmeur signale au système qu'un prédicat est ou n'est pas générateur de parallélisme.

Les modèles cités ci-après adoptent cette philosopie en empêchant tout parallélisme permettant d'accélerer la résolution d'un sous-but faisant appel à un appel d'un prédicat à effet de bord :

## Le Modèle Argonne

Le modèle Argonne [But 88] met à la disposition du programmeur deux méta-prédicats : *get_all* et *get_one* pour contrôler le parallélisme lors de la résolution d'un sous-but. La sémantique de tels prédicats est la suivante :

```
get_all (X, G, L) :- bagof (X,G,L).
get_one (G) :- call (G), !.
```

Où

1 - Le prédicat bagof est celui utilisé dans Prolog, sauf qu'aucune hypothèse n'est faite sur l'ordre des éléments de la liste L.

2 - Le prédicat attaché au sous-but G est un prédicat "*sûr*" (safe). Un prédicat est dit *sûr* s'il vérifie les conditions suivantes :

   a - son appel n'introduit pas un effet de bord lié aux entrées-sorties ou aux assert/ retract.

   b- il ne fait pas appel à un prédicat qui rend indispensable l'ordre des solutions fourni par Prolog.

   c- L'alternation des clauses définissant ce prédicat, n'a aucun effet sur l'ensemble des solutions, si ce n'est leur ordre. Dans le cas contraire, un tel prédicat est signalé *séquentiel* par la déclaration suivante : :-sequential <predicat>/<arity>.

3- Le CUT (!) dans la définition de get_one, permet d'obtenir une seule et quelconque solution, si elle existe, au but G, et non nécessairement la première solution - selon l'ordre fourni par la startégie en profondeur d'abord- comme c'est le cas dans Prolog. Une telle extension du CUT (*cavalier commit*) a été introduite [War 87b] afin de faciliter l'implantation du coupe-choix dans un environnement parallèle.

Ainsi, lors de la résolution d'un sous-but attaché à un prédicat "*sûr*", le parallélisme peut être exploité sans modifier la sémantique des méta-prédicats *get_all* et *get_one*. Le programmeur est alors conseillé , d' "encapsuler" les résolutions attachées au prédicats sûrs par les méta-pédicats *get_all* et *get_one*, et de renforcer le nombre des prédicats sûrs. Par exemple, en évitant d'utiliser le prédicat *write* pour afficher les solutions : Une clause de type :

```
go :- p(X), q(X), write(X).
```

doit être transformée en :

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

Le principal inconvénient de cette approche est qu'elle exige un style de programmation bien particulier, et une détection effectuée par le programmeur des prédicats sûrs compte tenu de la définition enoncée précédemment.

## Le langage PEPSys

Le langage PEPSys [Rat 87] est proche de Prolog séquentiel. Cependant, les programmes sont divisés en modules séquentiels et parallèles : il est possible d'appeler un prédicat défini dans un modèle parallèle depuis un modèle séquentiel mais la réciproque n'est pas vraie. Les effets de bord ainsi que le CUT sont autorisés dans les modules séquentiels mais pas dans les modules parallèles.

Dans les modules parallèles, chaque prédicat est précédé de *déclarations* permettant au programmeur de signaler au système qu'un prédicat est ou n'est pas générateur de parallélisme, que l'ordre d'exécution des clauses est important ou non, et enfin que l'on désire l'ensemble des solutions au prédicat ou seulement la première calculée.

Pour cela, une déclaration de propriétés est attachée à chaque définition de prédicat.

Une déclaration de propriétés et un terme :

**-propriétés ([p1(v1), . . . ,pn(vn)]).**

où p1, . . ., pn sont des noms de propriétés et v1, . . . ,vn sont les valeurs de chacune des propriétés.

### La propriété *solutions*

La proprité solutions peut avoir pour valeur *all* ou *one*. Elle permet de contrôler le nombre de solutions fournies par les résolutions d'un prédicat. Lorsque sa valeur est *all*, le prédicat se comporte comme un prédicat Prolog, c'est à dire qu'il peut produire plusieurs solutions. A l'inverse, la valeur *one* permet de limiter, le nombre de solutions produites par un prédicat, à au plus une. Lorsqu'il existe plusieurs alternatives pour résoudre un sous-but, seule la première solution produite par l'une de ces alternatives est considérée pour la continuation de la résolution, les autres sont abondonnées, ou éliminées si leur résolution n'avait pas encore commencé. Cette propriété a pour but de simuler l'effet du prédicat CUT dans un environnement parallèle.

### La propriété *clauses*

Cette propriété permet au programmeur de définir si l'ordre des clauses définissant un prédicat est significatif. Elle peut prendre la valeur *unordered* ou *ordered*. Dans ce dernier cas, l'ordre des solutions produites par les clauses en question, est le même que celui obtenu par la stratégie séquentiele.

### La propriété *execution*

La propriété *execution* a été introduite afin de contrôler explicitement l'efficacité du parallélisme. Le programmeur peut intervenir pour "conseiller" sur l'utilisation du parallélisme. La propriété *execution* peut avoir les valeurs *eager* ou *lazy* et qui spécifie que l'exécution parallèle est conseillée (*eager*) ou non (*lazy*).

**Exemple** : Codage du programme 8-reines dans le langage PEPSys

```
        /* module séquentiel */
go(Taille) :- bagof( sol, go(Taille, Sol), Solutions), member(S, Solutions),
              writeln(S), fail.

      /* module parallèle */
-properties([solutions(all), clauses(undordered), execution(lazy)]).
        go (Board_size, Soln) :- solve (Board_size, [], Soln, 0).

-properties([solutions(all), clauses(undordered), execution(lazy)]).
        solve ( Bs, L, L, Bs).
        solve( Board_size, Current , Final, Row):-
              Row \== Board_size,
              New_Row is Row + 1,
               index (New_Line, Board_size),
              safe(Current, New_Row, New_line),
              solve( Board_size, [s(New_Row, New_Line) | Current], Final, New_Row).

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

```
-properties([solutions(all), clauses(undordered), execution(lazy)]).
        safe ([], _ , _).
        safe(s(I,J) | L], X, Y) :- not(Threatened (I, J, X, Y)), safe(L, X, Y).

-properties([solutions(all), clauses(undordered), execution(lazy)]).
        threatened ( I, _ , I, _).
        threatened (_ , J, _ , J).
        threatened (I, J, X, Y) :- (U is  I-J ), (U is X-Y).
        threatened (I, J, X, Y) :- (U is  I+J ), (U is X+Y).
```

Dans le module parallèle, seul le prédicat *index* peut générer un parallélisme significatif, c'est pourquoi il a pour propriété *execution* la valeur *eager* alors que pour tous les autres prédicats cette valeur est *lazy* indiquant que le parallélisme n'est pas désiré. Il faut noter, à cet égard, que seul un programmeur "averti" du principe des modèles multiséquentiel, est capable de juger de l'importance du parallélisme dans le prédicat *index*.

Pour le prédicat *threatened*, la combinaison des propriétés solutions (one) et clause (ordered), ont permis de remplacer les CUTs de la version Prolog. Il faut noter que cette tradution ne s'effectue pas toujours de manière aussi simple. Considérons l'exemple [Rob 88] Prolog suivant :

```
si_alors_sinon :- condition, !, alors.
si_alors_sinon.
```

Un tel programme se traduit en PEPSys par :

```
-properties ([ solutions (all), . . .]).
 si_alors_sinon :- test_condition (Resultat), alors_ou_sinon(Resultat).

 -properties ([ solutions(one), clauses(ordered), . . .]).
  test_condition(oui) :-condition.
  test_condition(non).

-properties ([ solutions (all), . . .]).
  alors_ou_sinon(oui) :- alors.
  alors_ou_sinon (non) :- sinon.
```

Ceci constitue l'un des inconvénients du Langage PEPSys comparé à Prolog, dans la mesure où les clauses intermédiares peuvent, allourdir une programmation aisée offerte par Prolog. Ajouté à cela, toutes les déperditions dues aux appels suuplémentaires.

La deuxième approche [War87] utilisée pour tenir compte des effets de bord dans une exécution parallèle, veille à ce que le programmeur ne se pose aucun problème de parallélisme, et à ce que l'on puisse réutiliser des programmes écrits pour des implantations séquentielles. Le système doit être capable de gérer la synchronisation des effets de bord lorsqu'ils surviennent durant une exécution parallèle. Pour cela, un processus actif qui, parcourant une branche, rencontre un prédicat à effet de bord, se suspend et attend que la partie gauche de l'arbre de recherche soit entièrement explorée.

L'avantage d'une telle approche est, d'une part, d'éviter au programmeur d'intervenir par le biais d'un style de programmation, où par des annotation afin de contrôler le parallélisme, et d'autre, d'exploiter le maximum de parallélisme présent dans le programme.

La restriction qui consiste à empêcher le développement du parallélisme dans la résolution d'un sous-but, du fait qu'il fait apparaître un effet de bord, risque de pénaliser sévérement le gain espéré dû au parallélisme. Consiérons l'exemple suivant où le programmeur n'a pas respecté le style de programmation conseillé par le modèle Argonne :

go :- generer(X), tester(X), write(X).

La résolution du but *go* se réaliserait en séquentiel même si les sous-buts *generer(X)* et *tester(X)* peuvent présenter un parallélisme significatif.

## 1.1 - Les prédicats d'entrée-sortie :

Les prédicats systèmes *read* et *write* sont interprétés comme étant des prédicats déterministes. Le prédicat *read* permet de lire un terme Prolog sur le flux d'entrée et de l'unifier avec son argument. L'appel d'un *read* peut donc échouer si le terme lu ne s'unifie pas avec l'argument. Le prédicat *write* imprime son argument, quelque soit sa nature, constante, variable, terme fonctionnel ou liste. Contrairement au prédicat *read*, l'appel du prédicat *write* conduit toujours à un succès.

L'ordre d'exécution de plusieurs opérations d'entrée-sortie figurant dans un programme, est celui fourni par la stratégie *en profondeur d'abord*. Cet ordre peut être dans certains cas insignifiant pour le programmeur; ainsi, dans un système expert qui interagit avec l'utilisateur en lui demandant les valeurs de certains faits afin de satisfaire un but en essayant différentes règles, l'ordre des appels parallèles du prédicat *read* qui apparait dans ces règles, peut être sans importance.

De même, l'ordre de la sortie des résultats n'est pas significatif du tout pour le programmeur dans certains programmes où le but consiste à trouver toutes les solutions à un problème donné (8-reines, permutation de listes,. . .). Mais lorsque ces prédicats sont combinés avec des prédicats tels que *assert*, *retract* et *cut* qui permettent de modifier dynamiquement le programme lui même et le contrôle de l'exécution, l'ordre des appels des entrée-sorties doit être celui fourni par la stratégie *en profondeur d'abord*.

Dans le programme 1, l'exécution du sous-but *write('No (more) solutions')* dans la deuxième clause définissant le prédicat *shell_solve*, n'est réalisée que si la première clause donne un échec. Dans une exécution OU-parallèle, il est nécessaire de suspendre une telle exécution tant que la branche sur laquelle l'appel a lieu n'est pas la plus à gauche dans l'arbre de recherche.

```
shell :-
                shell_prompt,
                read(Goal),
                shell(Goal).

shell(exit) :-  !.
shell(Goal) :-
                 ground(Goal),
                !,
                shell_solve_ground (Goal),
                shell.
shell(Goal) :-  shell_solve(Goal),
                shell.


shell_solve(Goal) :-
                Goal,
                write(Goal),nl,
                 write('Next Solution?'),
                read(Answer),
                check(Answer),
                !.
shell_solve(Goal) :-
                 write('No (more) solutions), nl.


shell_solve_ground(Goal) :-
                 Goal,
                 !,
                 write('Yes'), nl.
shell_solve_ground(Goal) :-
                write('No'), nl.

check(no).

shell_prompt :- write('Next Command?').
```

figure 5.1 : Programme *shell Prolog* [Ste 86]

## 1.2 - Les prédicats *assert* et *retract* :

Dans la résolution Prolog, lorsque toutes les solutions au but initial sont données, l'environnement de calcul est défait. A la fin du parcours de l'arbre de recherche aucune information nouvelle n'a donc été créée à l'exception des messages apparus à l'écran [Del 88].

C'est pourquoi il a été utile d'introduire un moyen permettant de mémoriser une partie de l'historique du calcul, importante pour l'utilisateur. Les prédicats *assert* et *retract* répondent à ce besoin en permettant d'ajouter et de retirer des règles quelconques du programme. Certains interpréteurs Prolog comme Turbo-Prolog n'autorisent de telles opérations que sur des faits (i.e des règles sans corps), ce qui permet de manipuler des bases de données.

**Exemple :** Considérons le programme suivant :

```
q:- p.
q:- s.
p:- treslent, s.
p:-lent,assert(s).
p:-rapide,s.
treslent :- . . .
lent :- . . .
rapide :- . . .
```

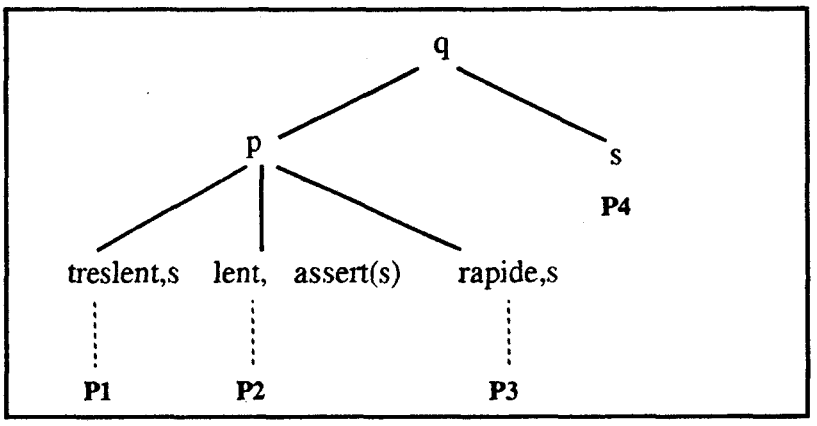



figure 5.2 : Arbre de recherche associé à la question q?.

Dans une exécution séquentielle (*en profondeur d'abord*), seule la première branche issue de p donnerait un échec, puisqu'il n'existe pas de règle qui puisse satisfaire le sous-but *s*. On suppose que les sous-buts *treslent*, *lent* et *rapide* donnent tous des succès. Une exécution *OU-parallèle*, où chacune des branches de l'arbre est parcourue par un processus, ne donnerait pas les mêmes résultats. En effet, si les processus *P3* et *P4* se terminent avant le processus *P2*, ils donneront tous les deux un échec, tandis que le processus *P1* explorant la première branche peut donner un succès si le sous-but *assert(s)* est exécuté par *P2* avant l'exécution du sous-but *s* par *P1*.

Afin de respecter la sémantique "séquentielle" du prédicat *assert*, il est nécessaire de suspendre non seulement toute branche sur laquelle a lieu l'appel de ce prédicat, et qui n'est pas la branche la plus à gauche dans l'arbre de recherche, mais aussi toute branche rencontrant un appel de prédicat dynamique. Un prédicat est dit dynamique s'il figure en tête d'une clause qui peut être retirée ou ajoutée à la base de connaissances. Dans notre exemple, l'exécution du but *s* par *P4* n'est possible que lorsque la branche parcourue par *P4* devient la branche la plus à gauche.

La même règle doit être appliquée au prédicat *retract*. En effet, dans l'exemple précédent, si le prédicat *assert* est remplacé par le prédicat *retract* et si le fait *s* est ajouté à la base de fait, dans le même cas de figure, les processus *P3* et *P4* donneront un succès, tandis que le processus P1 donnera un échec. La stratégie *en profondeur d'abord* aurait entraîné la situation inverse.

## 1.3 - Le prédicat coupe-choix *(cut)* :

Le prédicat *coupe-choix* (noté "! "ou "/") a été introduit afin de contrôler l'expansion de l'arbre de recherche en évitant que des explorations inutiles ne soient effectuées. Sa sémantique ne peut être définie qu'en termes de contrôle. L'exécution d'un coupe-choix consiste à supprimer tous les points de choix en attente à partir de celui qui a activé la règle contenant le coupe-choix, jusqu'au dernier point de choix créé. En termes de représentation arborescente, le *coupe-choix* supprime la partie contenant tous les noeuds *OU* créés qui figurent dans le sous-arbre ayant pour racine le but père du coupe-choix, y compris le but père s'il est lui-même un noeud *OU*. Nous appellerons simplement cette partie de l'arbre de recherche, le **champ du coupe-choix**.

**Exemple :**

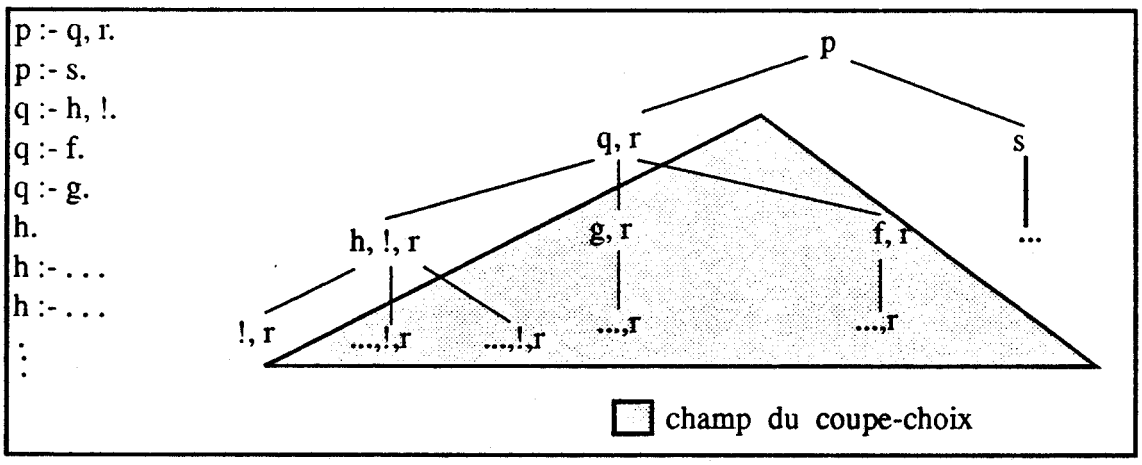



figure 5.3 : Arbre de recherche avec un coupe-choix.

Pour résoudre le but p, il y a deux choix possibles : la première alternative introduit la liste des sous-buts *q, r*; la deuxième alternative, *s*, est mise en attente.

Le but *q* introduit un point de choix à trois alternatives. La première donne la liste des sous-buts *h, !* (les deux autres sont en suspens). Le but *h* est satisfait par la première clause qui est un fait (les deux autres alternatives pour *h* sont mises en attente). L'exécution du coupe-choix provoque à ce moment, la suppression des choix en suspens pour *q* et *h*, mais pas pour *p*. Ainsi, lors du retour-arrière, il faudra directement essayer l'alternative en suspens du but *p*, en ignorant les points de choix relatifs à *q* et *h*.

Afin d'implanter le coupe-choix dans un modèle OU-parallèle, deux approches peuvent être utilisées.

La première consiste à explorer en séquentiel chaque sous-arbre associé à un sous-but dont l'une des alternatives possibles contient un coupe-choix. L'implantation de cette solution est simple à réaliser, mais son inconvénient est le risque de réduire les sources de parallélisme offertes par le programme.

La deuxième approche repose sur un principe énonçant que toute branche de l'arbre de recherche est une source potentielle de parallélisme. Cette approche introduit la notion de **travail spéculatif**. Nous appellerons le travail spéculatif, l'exploration des branches qui peuvent être amenées à disparaître suite à un coupe-choix, c'est à dire les branches se trouvant dans le champ d'un coupe-choix. Le problème à résoudre dans une telle approche apparait lorsque le programme contient plusieurs coupe-choix. En effet, dans la stratégie séquentielle, les branches se trouvant dans le champ d'un coupe-choix ne sont explorées que si le coupe-choix n'est pas rencontré. Dans une exécution OU-parallèle, ces mêmes branches peuvent être parcourues par plusieurs processus et donc, plusieurs coupe-choix peuvent être rencontrés sur ces branches. L'exécution de l'un de ces coupe-choix peut couper des branches qui auraient été explorées lors d'un parcours séquentiel. L'exemple ci-dessous illustre cette situation.

Exemple :

```
p :- q, !(1).
p:- speculatif1.
q:-r(X), !(2), s(X).
q:-speculatif2.
r(a). r(b). s(b).
```

Dans la figure 5.4 (voir page suivante), l'exécution du coupe-choix (2) sur la branche *B1* entraîne la suppression des branches *B2* et *B3*, le coupe-choix (1) n'étant jamais exécuté. Mais si les branches *B2* et *B3* sont explorées au même moment que la branche *B1*, et si le coupe-choix (1) est rencontré sur la branche *B2* avant que le coupe-choix (2) sur la branche *B1* ne soit exécuté, la branche *B4* se trouvant dans le champ de coupe-choix (1) est supprimée, alors que par la stratégie *en profondeur d'abord*, cette branche aurait été explorée puisque le coupe-choix (1) n'y serait jamais franchi.

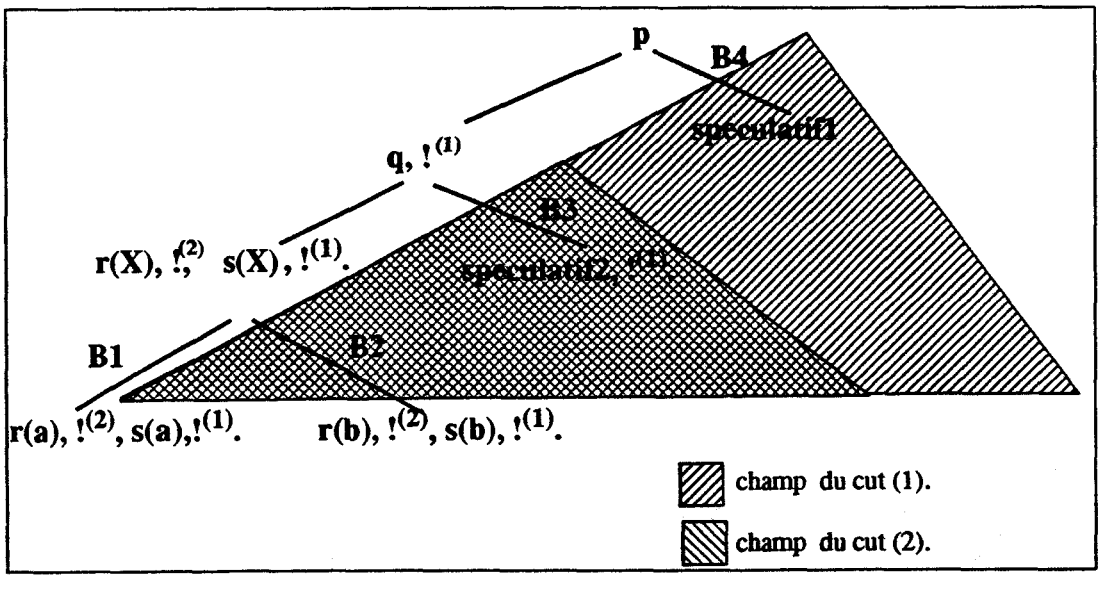



figure 5.4 : Arbre de recherche avec deux champs de coupe-choix

Avant de présenter les règles qui doivent être appliquées afin de respecter la sémantique du coupe-choix d'une part, et de considérer le travail spéculatif d'autre part, nous allons discuter de l'intérêt d'une telle volonté.

Le coupe-choix est fréquemment utilisé pour implanter implicitement une structure de type *si-alors-sinon*. Considérons l'exemple suivant :

```
si_alors_sinon :- condition0, !, action0.
si_alors_sinon :- condition1, !, action1.
si_alors_sinon :-action2.
```

Le "sens" d'un tel programme est : **si** condition0 **alors**
action0
**sinon si** condition1 **alors**
action1
**else**
action2.

Il pourrait sembler que l'utilisation du parallélisme lors d'un appel du prédicat *si_alors_sinon*, peut être inefficace puisque seule une alternative parmi les trois peut donner un succès. Ceci est vrai si les résolutions des sous-buts *condition0* et *condition1* ne nécessitent pas beucoup de calcul. Dans le cas contraire, et si la le sous-but *condition0* donne un échec, la continuation de la résolution tire avantage du parallélisme utilisé pour résoudre le sous-but *condition1*, d'où l'appelation de travail spéculatif.

Une autre utilisation du coupe-choix, qui est fréquente dans les programmes, a pour but de contrôler l'explosion de l'arbre de recherche, car seule une solution est importante au vue du programmeur. Considérons l'exemple suivant :

```
une_sol(X) :- generer(X), tester(X), !.
```

Dans un tel programme, même si le but *une_sol(X)* ne fournit qu'au plus une solution, l'utilisation du parallélisme permettrait d'accélerer la recherche de celle-ci, pourvu que les sous-buts *generer(X)* et *tester(X)* introduisent un parallélisme significatif.

Une telle utilisation du coupe-choix a été remplacée dans certains systèmes de programmation logique parallèle [War 87b], par l'utilisation d'un prédicat similaire (appelé en anglais *cavalier commit*) déja cité au début de ce chapitre. Ce dernier se prête facilement, de par sa définition,à une implantation dans un environnement parallèle, puisqu'il ne nécessite pas un mécanisme de suspension pour contrôler la branche la plus à gauche. Mais un tel prédicat ne peut remplacer entièrement le coupe-choix de Prolog, et plus particulièrement dans les programmes ou l'ordre des clauses définissant un prédicat a une importance, comme c'est le cas du prédicat *si_alors_sinon* définie ci-dessus.

Afin de respecter la sémantique du coupe-choix en tenant compte du travail spéculatif, l'une des règles suivantes doit être appliquée :

- La première règle, qui est la plus naturelle, consiste à autoriser l'exécution d'un coupe-choix sur une branche uniquement si celle-ci est la branche la plus à gauche dans l'arbre de recherche.
  Dans notre exemple précédent, l'exécution du coupe-choix (2) sur la branche *B1* ne pose aucun problème.

- La deuxième règle [Cal 88] consiste à restreindre la première règle précédente, au sous-arbre dont la racine est le noeud qui a donné naissance au coupe-choix. L'exécution d'un coupe-choix sur une branche n'est possible que si celle-ci est la branche la plus à gauche dans le sous-arbre de racine le noeud père du coupe-choix.

- La troisième règle [Haus 88], qui est la plus optimale, consiste à suspendre l'exécution d'un coupe-choix sur une branche uniquement si le champ du coupe-choix contient le champ d'un autre coupe-choix. Dans l'exemple de la figure 5.4, l'exécution du coupe-choix (2) sur la branche B2 peut être effectuée même si B2 n'est pas la branche la plus à gauche; par contre, l'exécution du coupe-choix (1) sur la même branche n'a lieu que lorsque la branche B2 devient la plus à gauche dans l'arbre de recherche, puisque le champ du coupe-choix (1) contient le champ du coupe-choix (2).

Par la troisième règle, l'exécution d'un coupe-choix n'est donc possible que s'il se trouve sur la branche la plus à gauche dans l'arbre de recherche, ou si le champ du coupe-choix ne contient aucun champ d'un autre coupe-choix.

Dans la figure 5.5, l'exécution de l'un des coupe-choix (3), (2) et (1) sur la branche la plus à gauche n'est pas suspendue. L'exécution du coupe-choix (3) sur n'importe quelle branche ne nécessite aucune suspension si l'on suppose que le champ du coupe-choix (3) ne contient pas le champ d'un autre coupe-choix. Mais l'exécution du coupe-choix (2) (resp (1)) dans le (les) sous-arbre (s) "speculatif3" (resp "speculatif 2" ou "speculatif 3") est suspendue jusqu'à ce que la branche sur laquelle est rencontré le coupe-choix devienne la branche la plus à gauche.

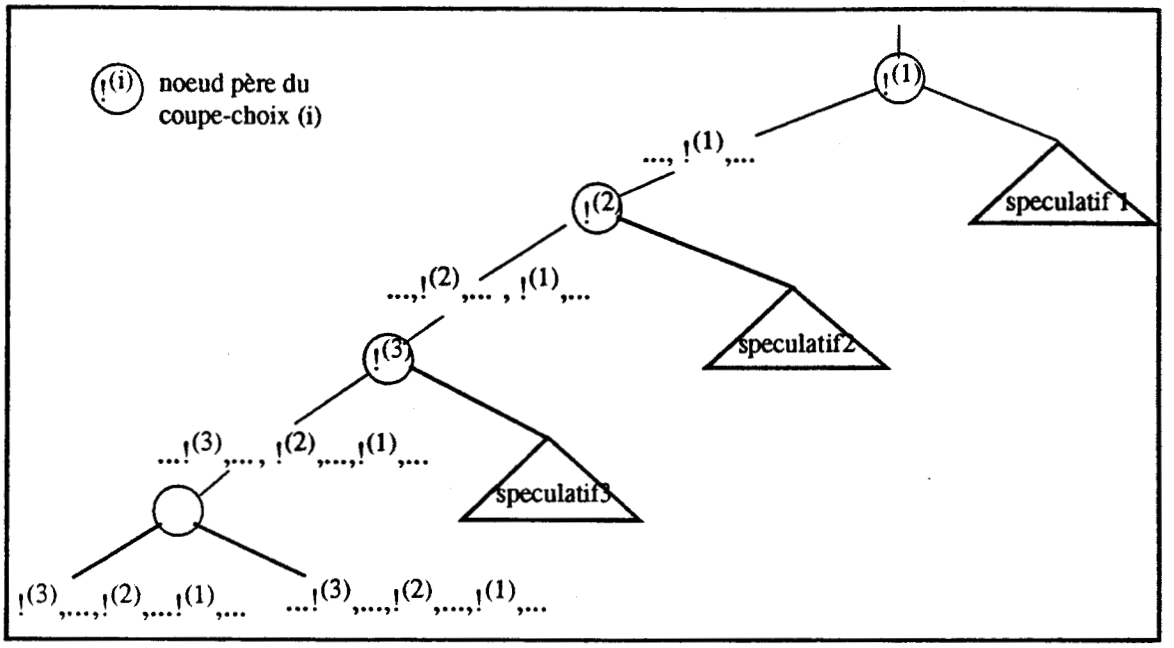



figure 5.5 : Arbre de recherche avec plusieurs champs de coupe-choix

Une autre solution proposée dans [Ali 87], mais qui ne considère pas le travail spéculatif, consiste à partitionner le paquet de clauses associé à chaque prédicat. Chaque sous-paquets est formé soit par des clauses successives ne contenant pas de coupe-choix, soit par des clauses successives contenant toutes au moins un coupe-choix. Dès qu'un sous-paquet ne contenant pas de coupe-choix est considéré, les clauses formant ce sous-paquet peuvent être essayées en parallèle. Un sous-paquet de clauses contenant des coupes choix, est traité en utilisant uniquement le retour-arrière.

## 2 - Suspension d'une branche :

Le fait de considérer que toute branche de l'arbre de recherche peut être parcourue en parallèle, nécessite un mécanisme de suspension afin de respecter la sémantique des prédicats à effets de bord. Lorsqu'un processeur rencontre un tel prédicat sur une branche, celle-ci doit être suspendue jusqu'à ce qu'elle devienne la branche la plus à gauche dans l'arbre de recherche.

La récupération de la ressource de calcul pour traiter une autre branche risque de remettre en cause l'hypothèse sur le nombre de liaisons d'une même variable effectuées par un processeur. En effet, un processeur suspendant une branche, peut se trouver en présence de plusieurs liaisons d'une même variable comme l'illustre l'exemple ci-dessous.

Exemple :

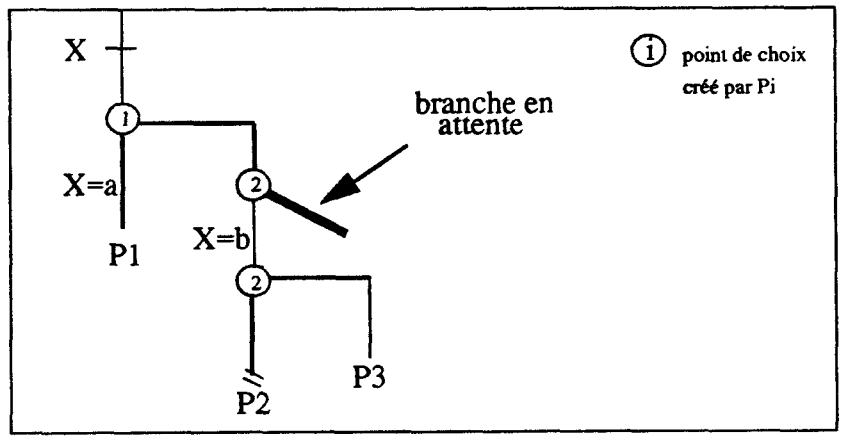



figure 5.6 : Suspension d'une branche

Dans la figure 5.6, si le processeur *P2* doit suspendre la branche courante suite à la rencontre d'un effet de bord, il ne peut pas traiter l'alternative en attente, puisque dans celle-ci, la variable *X* peut être liée une deuxième fois conditionnellement, par le processeur *P2* (la première liaison *X/b* ne peut pas être défaite).

Dans cette situation, le processeur *P2* se suspend et attend que le processeur *P1* termine la résolution de l'alternative en cours. Cette solution maintient, d'une part, l'hypothèse sur le nombre de liaisons d'une même variable, qui ne doit pas dépasser le nombre de processeurs, et d'autre part, la gestion simple des processus par pile. En effet, si un processeur entreprend une autre tâche après avoir suspendu une tâche donnant naissance à un effet de bord, cette dernière risque d'être réactivée avant que la nouvelle tâche ne soit terminée.

Pour réduire le temps d'attente d'un processeur suspendu à cause d'un effet de bord, la règle de suspension d'une branche peut être restreinte au sous-arbre qui peut être affecté par l'effet de bord. L'exécution d'un prédicat à effet de bord est possible dès que la branche en question constitue la branche la plus à gauche, non pas dans l'arbre de recherche associé au but initial, mais dans le sous-arbre qui peut être affecté par l'exécution de l'effet de bord.

Par exemple, l'exécution d'un coupe-choix peut être effectuée dès que la branche devient la branche la plus à gauche dans le sous-arbre de racine le noeud père du coupe-choix. Par contre, les prédicats d'entrée-sorties, et les prédicats *assert* et *retract*, affectent le programme tout entier; c'est pourquoi leur exécution doit être suspendue tant que la branche considérée n'est pas la branche la plus à gauche de l'arbre de recherche.

## 3 - Contrôle de la branche la plus à gauche :

Dans une approche multi-séquentielle (OU-parallèle), le parallélisme est contrôlé par la disponibilité des ressources. C'est, en effet, uniquement quand une ressource de calcul devient disponible qu'un processus est créé. La sémantique opérationnelle d'un modèle multi-séquentiel ne permet pas de déterminer de quelle façon se développe le parallélisme. De ce fait, la détection de la branche la plus à gauche dans l'arbre de recherche, n'est pas facile à réaliser à cause du caractère purement aléatoire du parallélisme.

Pour contrôler la branche la plus à gauche, la solution que nous proposons consiste à maintenir, au cours de l'exécution, l'identification du processeur parcourant la branche la plus à gauche. Les processeurs coopèrent par le biais d'un jeton qui permet d'identifier le processeur traitant la branche la plus à gauche. Lorsqu'un processeur possédant le jeton termine une branche, ou effectue un retour arrière vers un point de choix, il passe le jeton (il peut être amené à le garder) au processeur adéquat. Pour cela, chaque point de choix doit mémoriser et maintenir triée, selon l'ordre d'apparition des alternatives, la liste des tâches associées à ces alternatives. Considérons l'exemple suivant :

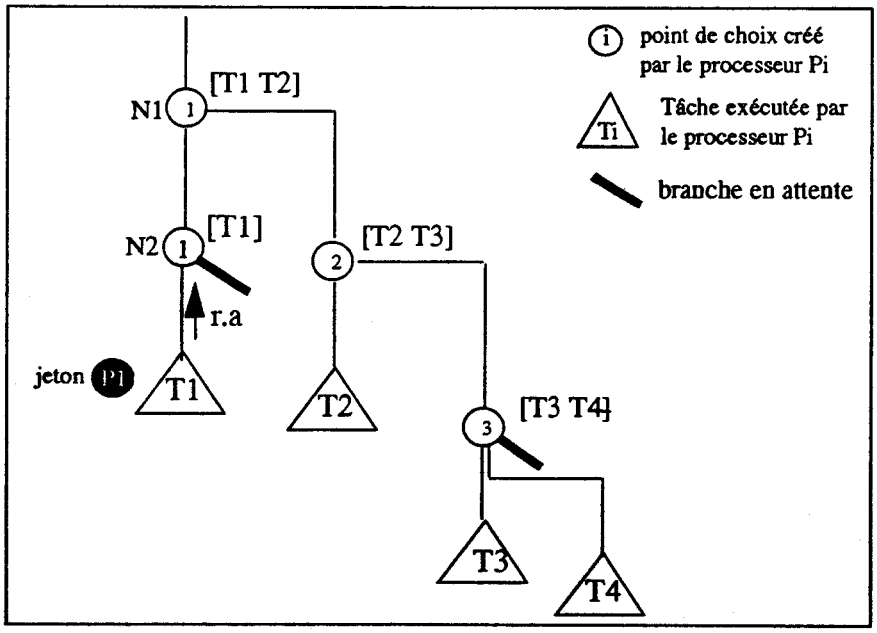



figure 5.7 : Identification de la branche la plus à gauche à l'aide d'un jeton

Dans la figure ci-dessus, lorsque le processeur *P1*, possesseur du jeton, effectue le retour arrière vers le noeud *N2*, il doit garder le jeton puisque la prochaine branche à traiter reste la plus à gauche dans l'arbre de recherche.

Par contre, lorsque le retour-arrière a lieu vers le noeud *N1*, le jeton doit passer au processeur *P2*; dans ce cas, le processeur *P1* identifie ce dernier par l'intermédiaire de la liste mémorisée dans le point de choix *N1*. Rappelons que le numéro d'une tâche, en raison de sa définition (voir paragraphe 1.1.1 dans le chapitre 1), permet de déterminer le numéro du processeur sur lequel elle s'exécute.

Le passage du jeton de *P1* vers *P2* lors du retour arrière effectué par *P1* vers *N1*, ne peut s'effectuer immédiatement car la tâche *T2* est simplement susceptible de correspondre au parcours de la branche la plus à gauche.

En effet, le processeur *P2* peut avoir changé de tâche, et dans ce cas, la branche courante n'est pas forcément la plus à gauche. Cette situation peut se produire dans l'exemple précédent, si le processeur *P2* interrompt la tâche *T2* (suite au retour arrière) pour aller traiter l'alternative en attente chez le processeur *T3* (et, ainsi," l'aider").

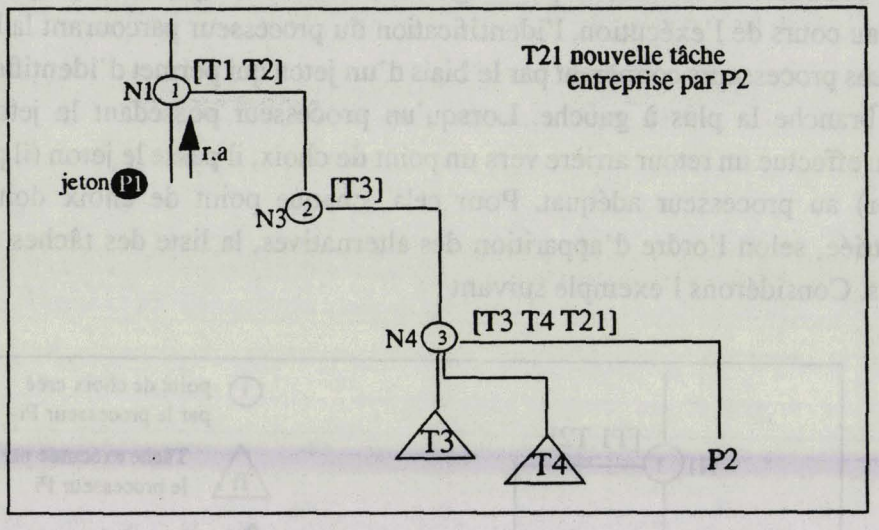



figure 5.8 : Passage du jeton

Lorsque le processeur *P1* tente de passer le jeton à *P2* (fig. 5.8), celui ci a déjà effectué un changement de tâche (la tâche courante étant T2*1*). Dans ce cas, le processeur *P1* doit être capable de déterminer que la branche la plus à gauche est celle traitée par le processeur *P3*.

Pour cela, le processeur accède tout d'abord au dernier point de choix créé au cours de la tâche *T2* (celui ci peut être mémorisé à la base du segment de pile alloué à *T2*). La tête de liste contenant les tâches issues de ce point de choix, est susceptible de représenter la branche la plus à gauche (elle le sera si elle n'a pas été interrompue ou si le processeur qui l'exécute n'a pas effectué un retour arrière).

Dans notre exemple, cette tâche étant *T3*, la même opération est recommencée en considérant *T3* comme la tâche susceptible de correspondre au parcours de la branche la plus à gauche. Le passage du jeton a lieu que lorsque la tâche susceptible de représenter la branche la plus à gauche, est en tête de la liste des tâches entreprises à partir du dernier point de choix créé au cours de cette tâche. Si l'on suppose que *N4* est le seul point de choix créé au cours de la tâche *T3*, cette dernière correspond à la branche la plus à gauche dans l'exemple de la figure 5.8; par conséquent le processeur *P3* devient possesseur du jeton.

Cette solution est facile à utiliser pour mettre en oeuvre la suspension d'une branche faisant apparaître un effet de bord. En effet, lorsqu'un processeur rencontre sur une branche un prédicat à effet de bord, il se met en attente du jeton.

Les inconvénients de cette solution sont les déperditions introduites, premièrement, pour maintenir ordonnée (selon l'ordre d'apparition des alternatives) la liste des tâches issues d'un point de choix, et, deuxièmement, lors de la désignation, par un processeur, du prochain processeur possesseur du jeton. Ce travail risque d'être inutile si aucun effet de bord n'a lieu au cours de l'exécution du programme.

Une première amélioration de cette solution consiste à supprimer le jeton en maintenant ordonnées les tâches actives qui sont issues de chaque point de choix. Pour déterminer si une branche parcourue est la plus à gauche, le processeur considéré doit vérifier que toutes les tâches ancêtres de la tâche courante (ainsi que la tâche courante elle même), sont en tête des listes mémorisées dans les points de choix créés au cours de ces tâches. Ce n'est donc qu'au moment où un processeur a besoin de vérifier qu'une branche est la plus à gauche, que le coût est supporté.

Une deuxième amélioration, tenant compte de la précédente, consiste à éviter à un processeur de parcourir tous les points de choix créés au cours des tâches ancêtres à la tâche active sur ce processeur. Pour cela, lorsqu'un processeur a déterminé que la branche qu'il est en train d'explorer est la plus à gauche, il marque tous les noeuds sur celle-ci. Si par la suite, un autre processeur est amené à vérifier que sa branche est la plus à gauche dans l'arbre de recherche, il ne parcourt pas tous les points de choix créés au cours des tâches ancêtres à la tâche active, mais il s'arrête dès qu'il rencontre un point de choix marqué et où la tête de liste des tâches issues de ce point de choix est une tâche ancêtre.

## 4 - Conclusion :

Dans ce chapitre, nous avons rappelé les deux approches utilisées pour tenir compte des prédicats à effets de bord dans un système parallèle de programmation logique. La première exige du programmeur d'intervenir soit par le biais d'annotations mises à sa disposition, soit en adoptant une discipline de programmation. La deuxième approche a pour principe d'épargner le programmeur de tout problème lié au parallélisme, et vise à garde une totale compatibilité avec Prolog.

Nous avons choisi la deuxième approche, en montrant par des exemples que même en présence des prédicats à effets de bord, l'utilisation du parallélisme peut constituer une source d'efficacité.

Nous avons ensuite montré les problèmes posés par l'utilisation d'une telle approche. Afin de préserver la même sémantique que celle donnée par le modèle séquentiel, l'implantation des prédicats à effets de bord nécessite la mise en place d'un mécanisme de suspension d'une branche donnant naissance à un prédicat à effet de bord, celle-ci étant réactivée lorsqu'elle devient la branche la plus à gauche dans l'arbre de recherche.

Si l'implantation des prédicats à effet de bord, dans les modèles OU-parallèles multi-séquentiels, est relativement simple à réaliser, l'efficacité du parallélisme, risque quant à elle, d'être réduite considérablement par la présence des effets de bords dans les programmes, même si ces derniers offrent un parallélisme potentiel.

Dans notre modèle, deux raisons peuvent engendrer une diminution des bénéfices obtenus par le parallélisme. La première d'entre elle est la suspension du processeur exécutant un effet de bord, et non pas seulement la suspension de la tâche courante qui s'exécute sur le processeur. Rappelons que cette restriction à été introduite afin de maintenir notre hypothèse sur le nombre de liaisons d'une même variable, qui ne doit pas dépasser le nombre de processeurs.

La deuxième raison pouvant être source de déperditions, est le fait de considérer que toute branche de l'arbre de recherche peut constituer une source de parallélisme, ce qui risque d'augmenter la quantité du travail spéculatif dans les programmes (travail qui peut être "gaspillé").

Pour pallier ces problèmes, nous montrerons, dans le chapitre suivant, qu'une stratégie de recherche de travail disponible appliquée aux processeurs oisifs, et qui consiste à favoriser les branches les plus à gauche dans l'arbre de recherche, permet de réduire, d'une part, le temps d'attente d'un processeur suspendu suite à la rencontre d'un effet de bord, et, d'autre part, la quantité du travail spéculatif.

# -=- CHAPITRE VI -=-

## SIMULATION DE LA MACHINE ABSTRAITE PARALLELE

Dans cette dernière partie, nous présentons les réalisations effectuées, qui ont succédé à l'élaboration du modèle et à la définition d'une machine abstraite pour ce modèle. Ces réalisations sont axées sur une simulation effectuée sur une station de travail *Sparc*.

Le but initial de cette simulation est de répondre, dans un délai relativement court par rapport à une réelle implantation, aux objectifs suivants :

- **Valider le modèle de calcul ainsi que la machine abstraite parallèle;**
- **Etudier et comparer différentes méthodes (mécanismes de liaisons, stratégies d'allocation des travaux aux processeurs oisifs);**
- **Prendre en compte des programmes significatifs et de tailles importantes;**
- **Comparer avec une implantation séquentielle.**

Après avoir décrit le simulateur, nous donnerons, dans ce chapitre, les résultats des mesures de performances obtenus.

# 1 - Description du simulateur :

Pour répondre aux deux derniers objectifs énoncés ci-dessus, il a été nécessaire de prendre comme base de départ, une implantation séquentielle de Prolog. La machine abstraite séquentielle qui a été utilisée comme base pour la simulation, est une version interprétée de la machine abstraite de Warren, donnée dans [Boi 88] (version microlog 2, écrite en Le_Lisp). Elle correspond à la description de la WAM donnée dans le premier chapitre. Les caractéristiques principales de cette machine sont les suivantes :

• indexation des clauses :

L'indexation des clauses ne correspond pas tout à fait à la solution donnée dans le paragraphe 2.7.a du chapitre I. En effet, la discrimination des clauses ne s'effectue pas en testant la nature ou la valeur du premier argument du but, mais en essayant directement l'unification. Tant que le paquet possède plusieurs clauses, et que l'unification avec la tête de la première clause échoue, les tentatives sont poursuivies. Lorsqu'une tentative réussit, si la clause en question est la dernière du paquet, un bloc d'activation déterministe est créé; dans le cas contraire, un bloc de choix est créé comportant le reste des alternatives à essayer.

• transformation des appels terminaux :

Les trois piles sont mises à jour lors du retour-arrière. De plus, la pile locale est mise à jour par l'application de la transformation terminale. C'est pourquoi la sauvegarde des arguments d'un but dans des registres et la détection des variables dangereuses ont été prises en compte.

• prédicats à effets de bord :

En plus des prédicats arithmétiques, l'interpréteur intègre certains prédicats à effets de bords comme *read, write* et *cut*, qui représentent un grand intérêt pour notre étude.

## 1.1 - Extensions et limitations :

Nous avons étendu la version séquentielle afin d'obtenir une machine abstraite parallèle. Le simulateur est constitué de 1320 lignes de code Le_lisp, soit environ 180 fonctions et macro fonctions.

Les caractéristiques de la machine initiale, décrites précédemment, ont été modifiées dans le simulateur de la machine abstraite parallèle de la manière suivante :

• indexation :

Dans une implantation séquentielle de Prolog, l'indexation des clauses d'un paquet a lieu pour la première fois, lors de l'appel d'un but non déterministe pouvant être satisfait par ces clauses, l'indexation étant ensuite poursuivie, lors de chaque retour-arrière, sur les clauses restantes.

Dans une implantation OU-parallèle de Prolog, l'indexation peut avoir lieu également lorsqu'un processeur prend en charge une alternative d'un point de choix, celle-ci peut être sélectionnée en utilisant l'indexation, ce qui permet d'éviter à un processeur oisif de s'engager dans des branches donnant rapidement un échec.

Dans l'état actuel du simulateur, seul le processeur ayant créé un point de choix est autorisé à effectuer l'indexation des clauses associées à ce point de choix. Lorsqu'un processeur oisif trouve un point de choix détenant plusieurs alternatives, il traite la première sans tenir compte des autres.

• transformation des appels terminaux :

La transformation terminale est appliquée uniquement aux piles locales. La solution proposée dans le quatrième chapitre, permettant de généraliser la transformation terminale aux tableaux de liaisons, n'a pas encore été intégré dans le simulateur. Les tableaux de liaisons sont mis à jour uniquement lors du retour-arrière.

• prédicats à effets de bord :

Le simulateur supporte uniquement les prédicats d'entrée-sorties. Le prédicat *cut*, n'a pas été entièrement implanté. En effet, il est pris en compte uniquement dans les programmes faisant apparaître le *cut* à la racine de l'arbre de recherche. Nous reviendrons sur ce point, lorsque nous aborderons l'évaluation des performances dans les programmes introduisant des effets de bord.

## 1.2 - Simulation du parallélisme :

Afin de prendre compte le parallélisme, nous avons utilisé le mécanisme de coroutine fourni par le système Le_Lisp. Les processeurs virtuels fonctionnent en temps partagé. La gestion du temps s'effectue en allouant une unité de temps à chaque processeur virtuel, qui est une instance de la machine séquentielle initiale, pour avancer dans la résolution s'il est actif, ou pour continuer la recherche d'un travail disponible, dans le cas où il est oisif.

Une unité de temps est équivalente à 2 centièmes de seconde du temps horloge réel. Cette unité n'est pas toujours constante; en effet, pour tenir compte de l'exclusion mutuelle, un processeur accédant à une section critique est ininterruptible, pendant cet accès, par le système gérant le temps partagé. De ce fait, l'unité de temps allouée à un processeur peut varier en fonction des accès en exclusion mutuelle. Mais il s'est avéré que le temps total alloué à un processeur est globalement constant pour un programme donné.

Afin d'effectuer des mesures de performances, nous avons défini un temps logique comme étant le nombre de cycles nécessaires pour l'exécution d'un programme : un cycle est une période du temps partagé, où chaque processeur travaille durant une unité de temps définie précédemment (i.e 1 cycle = nombre de processeurs * l'unité du temps partagé).

Nous utiliserons également le temps réel, qui est le temps de la simulation de l'exécution d'un programme. Ce temps ne permet pas d'évaluer le gain de parallélisme, mais il offre une précision plus importante, lorsque différentes méthodes (mécanismes de liaisons, stratégies d'allocation de travail aux processeurs oisifs) sont comparées.

## 1.3 - Stratégies de recherche de travail par les processeurs oisifs :

Comme nous l'avons vu dans le deuxième chapitre, les systèmes multiséquentiels ne nécessitent pas un contrôle centralisé pour allouer les travaux aux processeurs oisifs. La recherche d'un travail est prise en charge directement par un processeur oisif. Dès qu'une alternative est trouvée, elle peut être traitée immédiatement par le processeur en question. C'est ici qu'apparait un point sensible pour une implantation efficace du parallélisme-OU multiséquentiel. En effet, le fait de considérer que toute branche de l'arbre de recherche peut être explorée en parallèle, fait apparaître deux facteurs qui peuvent être critiques. Le premier concerne la granularité du parallélisme, tandis que le deuxième concerne le travail spéculatif.

Une implantation efficace du parallélisme-OU consisterait, d'une part, à augmenter la granularité des tâches, en empêchant les processeurs oisifs de s'engager dans l'exploration des branches donnant rapidement un échec et par là même augmentant le nombre de changements de tâches, et d'autre part, à réduire la quantité du travail spéculatif, en interdisant aux processeurs de s'engager, cette fois, dans des branches qui peuvent être supprimées par un coupe-choix.

C'est pour ces raisons que certains modèles laissent la charge au programmeur d'indiquer au système, par le biais d'annotations, les points de choix qui peuvent constituer un parallélisme potentiel. Cette approche n'est pas suffisante pour extraire le maximum de parallélisme dans les programmes, en indiquant simplement qu'un paquet de choix est une source de parallélisme. L'exemple de la figure 3.1, donné dans le troisième chapitre, montre la difficulté d'extraire un parallélisme significatif à l'aide de simples annotations.

Une autre approche, utilisée dans le système Aurora [Lus 88], consiste à utiliser des mécanismes d'allocations du travail aux processeurs oisifs, permettant de réduire le coût moyen des changements de contextes, et d'augmenter le grain du parallélisme.

Cependant, dans ces mécanismes d'allocations, le deuxième facteur critique, qui est le travail spéculatif, n'est pas pris en compte.

Afin d'exploiter au mieux la granularité des tâches parallèles et de réduire la quantité du travail spéculatif, nous avons étudié trois stratégies de recherche du travail appliquées aux processeurs oisifs :

• La première stratégie, appelée ***topmost***, consiste à concentrer la recherche du travail par les processeurs oisifs, à proximité de la racine de l'arbre de recherche. Cette stratégie est fréquemment utilisée dans les systèmes basés sur la recopie, afin de réduire la taille de l'environnement à recopier lors de l'installation d'une tâche parallèle.

• La deuxième stratégie, appelée ***leftmost***, consiste à favoriser l'exploration des branches se trouvant le plus à gauche dans l'arbre de recherche. Le but de cette stratégie est de faire éviter aux processeurs oisifs de s'engager dans l'exploration des branches se trouvant dans le champ d'un coupe-choix. L'idée intuitive de cette stratégie découle de la sémantique opérationnelle du coupe-choix.

• La troisième stratégie, appelée ***deepmost***, consiste à donner la priorité à un travail disponible dans le fond de l'arbre de recherche. Cette stratégie est proche de la stratégie *leftmost*, dans la mesure où le grain de parallélisme est plus fin que celui qui peut être offert par la stratégie *topmost*. Mais il semblait intéressant d'étudier une telle stratégie dans la mesure où, contrairement à la stratégie *leftmost*, elle ne nécessite pas le contrôle de l'ordre des branches de l'arbre de recherche, lorsqu'il s'agit de détecter le travail le plus à gauche. De plus, elle offre aux processeurs oisifs un domaine de recherche de travail plus large que celui offert par la stratégie *leftmost*.

La figure 6.1 de la page suivante illustre ces définitions en indiquant quelles sont les branches sélectionnées par des processeurs oisifs, suivant la stratégie utilisée.

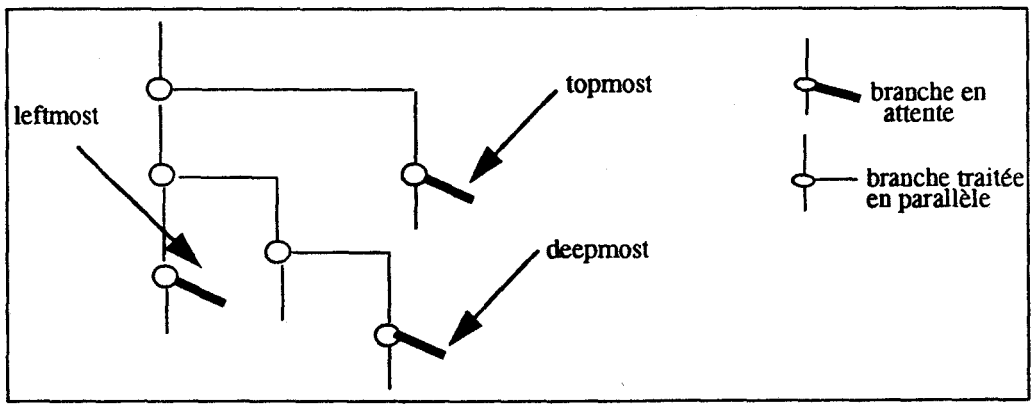



figure 6.1 : Stratégies d'allocation du travail aux processeurs oisifs.

Afin d'étudier le comportement du parallélisme obtenu par ces trois stratégies, celles-ci ont été intégrées dans le simulateur. Leur implantation a été réalisée de manière naïve. En effet, nous n'avons pas étudié d'algorithmes optimaux. La structure de données utilisée, qui permet aux processeurs oisifs d'accéder aux points de choix pouvant être traités en parallèle, est une pile. Cette structure peut s'adapter à l'implantation des stratégies *lefmost* et *deepmost*, puisqu'un travail disponible trouvé en appliquant l'une de ces stratégies, est nécessairement fourni par un point de choix situé en sommet de pile de choix de l'un des processeurs. Cependant, le choix de cette structure n'est pas optimale pour l'implantation de la stratégie *topmost*. Une structure de données de type liste doublement chaînée, permettrait une flexibilité dans l'implantation de ces différentes stratégies.

## 1.4 - Implantation du mécanisme de liaison :

L'algorithme d'unification de la machine séquentielle initiale a été étendu afin de tenir compte de la gestion des liaisons multiples. Ces extensions correspondent à la solution proposée dans le modèle décrit au chapitre III. Les trois méthodes de représentation des liaisons profondes, à savoir, l'utilisation de tables de hachage, de vecteurs de liaisons, ou de tableaux de liaisons, ont été implantées.

Les résultats qui seront présentés ci-après, sont obtenus par deux séries de mesures. La première série de mesures concerne le mécanisme de liaison employé, ainsi que les performances de chacune des trois méthodes de représentation des liaisons profondes.

Le but de la deuxième série de mesures est d'étudier le comportement des stratégies présentées ci-dessus, dans des programmes avec ou sans effets de bord, et d'évaluer ensuite le gain de performance.

# 2 - Evaluation :

## 2.1 - Programmes de tests :

Les programmes de tests que nous avons sélectionnés sont utilisés dans de nombreux systèmes pour évaluer les performances de ceux-ci. Ils vont du simple programme de quelques centaines d'inférences à des programmes comportant des milliers d'inférences. Certains programmes mettent en jeu un grand nombre de termes complexes. D'autres introduisent un nombre important de faits.

La liste proposée n'est pas exhaustive, mais elle représente plusieurs types d'algorithmes non déterministes Prolog.

Programme ***map*** :

Ce programme traduit un jeu qui consiste à colorier une carte de pays avec une palette finie de couleurs, sachant que deux pays voisins doivent avoir des couleurs différentes.

Programme ***mutation*** :

Ce programme permet de construire, à partir d'un ensemble de mots, des mots dérivés en juxtaposant tous ceux dont la chaîne terminale de caractères de l'un coïncide avec la chaîne initiale de l'autre.

Programme ***Mu*** :

Il s'agit du fameux système formel "Mu". Le programme permet de générer tous les théorèmes pour une longueur de dérivation donnée, et de vérifier si un mot est un théorème pour le système.

Programme ***permutation*** :

Ce programme consiste à donner toutes les permutations d'une liste donnée.

Programme ***Ham*** :

Ce programme calcule tous les chemins hamiltoniens dans un graphe.

Programme ***interrogation*** :

Ce programme permet d'interroger, dans un pseudo-français, une base de connaissances sur des liens de parenté. Il se compose d'un analyseur de formules logiques, d'un constructeur-évaluateur de requêtes pour la base de connaissances, de l'ensemble des règles de parenté, et des relations de base. Nous avons relevé des mesures pour deux interrogations (proposées dans [Per 90]).

Programme ***8-reines*** :

Il s'agit du célèbre problème du placement de n reines sur un échiquier, sans que l'une d'entre elles ne soit attaquée.

N.B. : les sources de ces programmes sont donnés dans la partie annexe.

## 2.2 - Méthodes de liaisons :

Les programmes présentés ci-dessus ont été testés pour une configuration de dix processeurs. La stratégie *topmost* a été choisie pour répartir le travail entre les processeurs.

Pour chacune des trois méthodes de représentation des liaisons profondes (tables de hachage, vecteurs de liaisons et tableaux de liaisons), nous avons mesuré le paramètre "critique", ainsi que d'autres paramètres liés au mécanisme de liaison utilisé.

Paramètres mesurés :

| | |
|---|---|
| **Hw_coll** | nombre total des collisions dues au hachage.<br>La fonction de hash code employée consiste à prendre le reste de la division de l'adresse de la variable par la taille de la table de hachage.<br>Les éléments entrés en collision dans une table sont chaînés entre eux. |
| **V_syn** | estimation du nombre de vecteurs de liaisons créés nécessitant une synchronisation. Lorsqu'une variable est liée de façon conditionnelle pour la première fois, un vecteur de taille *N+1* (au lieu de *N* qui est le nombre de processeurs), est créé pour contenir les éventuelles liaisons multiples de la variable. La cellule supplémentaire contient la date logique de création de ce vecteur. Lorsqu'un processeur lie une variable conditionnellement, il compare la date logique courante à celle enregistrée dans le vecteur associé à la variable (si celui-ci est déjà créé); si les deux dates coïncident, le paramètre V_syn est incrémenté. |
| **Pba_max** | taille maximale de l'espace utilisé dans un tableau de liaisons. |
| **cbd** | nombre total des liaisons profondes effectuées. |
| **sbd** | nombre total de liaisons effectuées avec le mécanisme de liaison superficielle. |
| **deref** | nombre total d'appels de déréférencement. |
| **%df_dp** | pourcentage de déréférencement nécessitant la recherche d'une liaison valide. |
| **df_max** | nombre moyen d'accès aux liaisons profondes d'une variable lors d'une recherche de liaison valide. |
| **T_log** | temps logique d'exécution (en cycles). |
| **T_sim** | temps de simulation (en secondes). |

Les tableaux ci-après donnent, pour chacun des programmes de test, les résultats obtenus par les mesures des paramètres précédents. Chacune des trois premières lignes correspond au paramètre critique pour la méthode. Les autres lignes correspondent aux paramètres liés au mécanisme de liaison employé.

| Hw_Coll | 0 | | |
|---|---|---|---|
| V_syn | | 24 | |
| Pba_max | | | 14 |
| deref | 695 | 695 | 695 |
| cbd | 81 | 90 | 90 |
| sbd | 23 | 14 | 14 |
| %df_bd | 8 | 8 | 8 |
| df_max | 1 | 1 | 1 |
| T_log (en cycles) | 32 | 30 | 18 |
| T_sim (en sec) | 8 | 7.5 | 6.5 |

Table 6.1 : Mesures de l'exécution du programme ***map***

| Hw_Coll | 0 | | |
|---|---|---|---|
| V_syn | | 45 | |
| Pba_max | | | 82 |
| deref | 8943 | 8943 | 8943 |
| cbd | 515 | 825 | 850 |
| sbd | 3844 | 3534 | 3509 |
| %df_bd | 11 | 11 | 11 |
| df_max | 2 | 2 | 2 |
| T_log | 114 | 110 | 112 |
| T_sim | 26 | 25 | 25.5 |

Table 6.2 : Mesures de l'exécution du programme ***mutation***

| Hw_Coll | 8 | | |
|---|---|---|---|
| V_syn | | 7 | |
| Pba_max | | | 237 |
| deref | 25923 | 25923 | 25923 |
| cbd | 206 | 613 | 590 |
| sbd | 12377 | 11970 | 11990 |
| %df_bd | 1 | 1 | 1 |
| df_max | 4.5 | 4.5 | 4.5 |
| T_log | 171 | 147 | 151 |
| T_sim | 59 | 49 | 50 |

Table 6.3 : Mesures de l'exécution du programme ***Mu***

| Hw_Coll | 12 | | |
|---|---|---|---|
| V_syn | | 605 | |
| Pba_max | | | 124 |
| deref | 51245 | 51245 | 51245 |
| cbd | 657 | 7767 | 7801 |
| sbd | 27287 | 20177 | 20143 |
| %df_bd | 1 | 1 | 1 |
| df_max | 4 | 4 | 4 |
| T_log | 575 | 530 | 431 |
| T_sim | 144 | 132 | 108 |

Table 6.4 : Mesures de l'exécution du programme ***interrogation 1***

| Hw_Coll | 47 | | |
|---|---|---|---|
| V_syn | | 7989 | |
| Pba_max | | | 207 |
| deref | 2862654 | 2862654 | 2862654 |
| cbd | 5723 | 394588 | 394631 |
| sbd | 1525450 | 1136585 | 1136542 |
| %df_bd | 0.2 | 0.2 | 0.2 |
| df_max | 3 | 3 | 3 |
| T_log | 28277 | 22767 | 20113 |
| T_sim | 7246 | 5786 | 4978 |

Table 6.5 : Mesures de l'exécution du programme ***interrogation 2***

| Hw_Coll | 20 | | |
|---|---|---|---|
| V_syn | | 625 | |
| Pba_max | | | 105 |
| deref | 65686 | 65686 | 65686 |
| cbd | 423 | 6827 | 6729 |
| sbd | 29684 | 23240 | 23338 |
| %df_bd | 1 | 1 | 1 |
| df_max | 4 | 4 | 4 |
| T_log | 645 | 507 | 344 |
| T_sim | 165 | 162 | 122 |

Table 6.6 : Mesures de l'exécution du programme ***permutation***

| Hw_Coll | 188 | | |
|---|---|---|---|
| V_syn | | 71 | |
| Pba_max | | | 494 |
| deref | 308260 | 308260 | 308260 |
| cbd | 1470 | 16029 | 15993 |
| sbd | 142429 | 127870 | 127906 |
| %df_bd | 0.6 | 0.6 | 0.6 |
| df_max | 4 | 5 | 5 |
| T_log | 4644 | 4259 | 3715 |
| T_sim | 1715 | 1386 | 1299 |

Table 6.7 : Mesures de l'exécution du programme ***ham***

| Hw_Coll | 0 | | |
|---|---|---|---|
| V_syn | | 10 | |
| Pba_max | | | 420 |
| deref | 1269579 | 1269579 | 1269579 |
| cbd | 15718 | 15720 | 15720 |
| sbd | 466736 | 466734 | 466734 |
| %df_bd | 4.3 | 4.3 | 4.3 |
| df_max | 3 | 4 | 4 |
| T_log | 11449 | 9516 | 9164 |
| T_sim | 2812 | 2416 | 2291 |

Table 6.8 : Mesures de l'exécution du programme ***8-reines***

D'après les résultats ci-dessus, la méthode basée sur l'utilisation des tableaux de liaisons montre de meilleures performances que les deux autres méthodes.

Malgré une moyenne des liaisons profondes enregistrées inférieure à celle des deux autres méthodes[1], et malgré un faible nombre de collisions, la méthode de hachage produit plus de déperditions à cause des accès dans les tables, qui nécessitent, chaque fois, un appel de la fonction de hachage.

En ce qui concerne les deux autres méthodes, celle basée sur l'utilisation des vecteurs de liaisons est moins performante dans les programmes offrant un parallélisme massif, et présente des performances très proches de celles obtenues par la méthode basée sur l'utilisation des tableaux de liaisons, dans les programmes de tailles relativement petites (*map, Mu et mutation*).

Ceci s'explique par le besoin de synchroniser les créations dynamiques des vecteurs de liaisons. Cette synchronisation ne peut se produire véritablement dans notre simulateur, à cause

1. Rappelons que dans la méthode de hachage, le mécanisme de liaison profonde est utilisé uniquement lors d'une liaison conditionnelle non locale, tandis que dans les 2 autres méthodes, celui-ci est également utilisé lors d'une liaison conditionnelle locale au processeur effectuant la liaison.

du pseudo-parallélisme ("multi-tâches") utilisé. Pour éviter plusieurs créations d'un même vecteur, la création de celui-ci a été réalisée en exclusion mutuelle. C'est ce qui a pénalisé la méthode des vecteurs, à cause des appels du mécanisme d'exclusion mutuelle à chaque création d'un vecteur.

Ces résultats ont mis en évidence le faible nombre de déréférencements nécessitant la recherche d'une liaison profonde valide.

En moyenne 4% du nombre total de déréférencements font appel à une recherche de liaison valide. Le nombre d'accès dans les structures (tables, vecteurs ou tableaux) est en moyenne égal à 3.5.

Ce faible nombre s'explique par la localité des références pour chacun des processeurs; celle-ci est obtenue grâce à la grosse granularité du parallélisme offerte par la stratégie *topmost* dans les programmes de tailles importantes. Constatons que dans les programmes *map* et *mutation*, où le parallélisme est plus faible (relativement aux 10 processeurs utilisés), le nombre de déréférencements nécessitant une recherche dans les structures, est plus élevé (10%).

## 2.3 - Evaluation du gain de performances :

Avant de donner les résultats, il faut considérer la façon dont ils sont obtenus. Comme il a été vu dans le paragraphe 1.2, le temps simulé est géré par le système contrôlant le temps partagé. Il est incrémenté à chaque cycle, et est donc mesuré en cycles.

Les résultats présentés ci-après, concernant l'activité des processeurs, ont été obtenus de la manière suivante : chacun des processeurs gère deux horloges du temps simulé, fonctionnant de manière alternée, l'une étant utilisée pour compter la durée d'activité du processeur, l'autre pour compter la durée d'oisiveté du processeur. Seuls ces deux états sont à considérer pour l'instant, du fait de l'inexistence des effets de bord dans les programmes testés.

### 2.3.1 - Activités des processeurs :

Nous avons utilisé le même jeu de programmes que précédemment. Pour chaque processeur, nous avons mesuré le taux d'activité et d'oisiveté, en faisant varier le nombre de processeurs de deux en deux lorsque la taille du programme justifie l'ajout de nouveaux processeurs.

Pour chaque configuration, et pour chaque programme, nous avons considéré les trois stratégies de recherche du travail (*leftmost*, *deepmost* et *topmost*) définies précédemment.

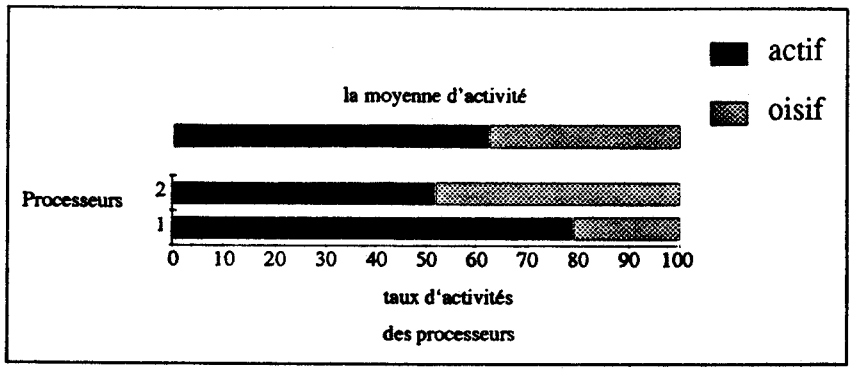



figure 6.2 : Taux d'occupation des processeurs dans le programme ***map*** (leftmost)

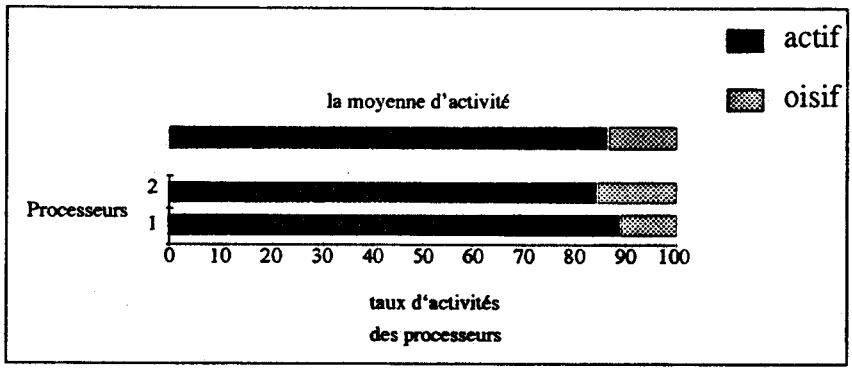



figure 6.3 : Taux d'occupation des processeurs dans le programme ***map*** (deepmost)

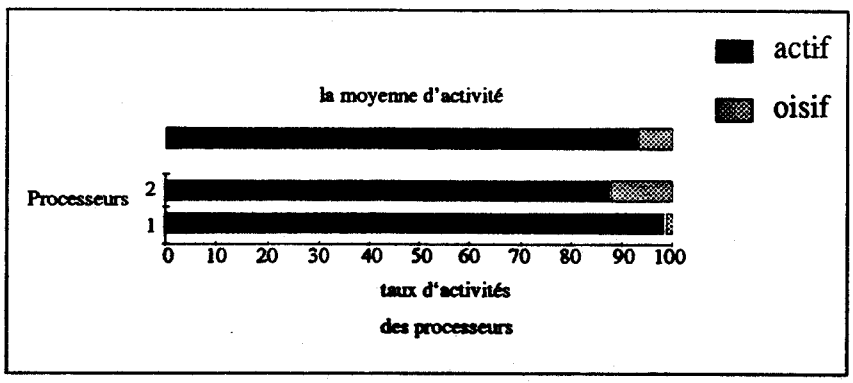



figure 6.4 : Taux d'occupation des processeurs dans le programme ***map*** (topmost)

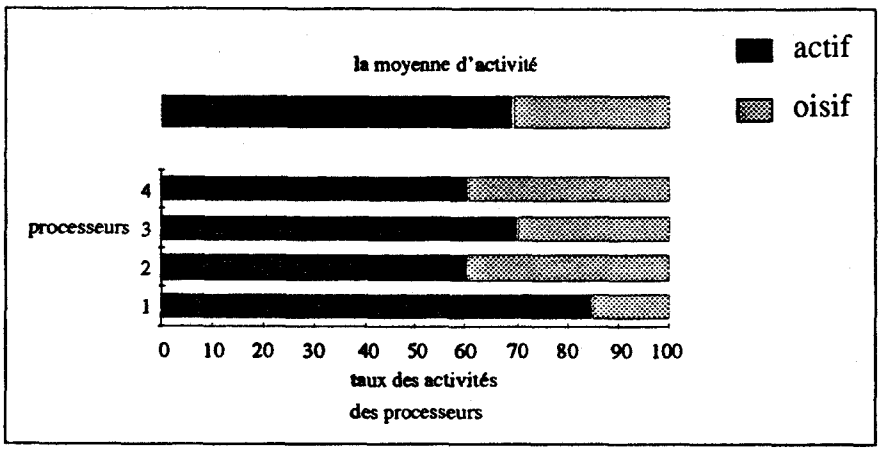



figure 6.5 : Taux d'occupation des processeurs dans le programme ***mutation*** (leftmost)

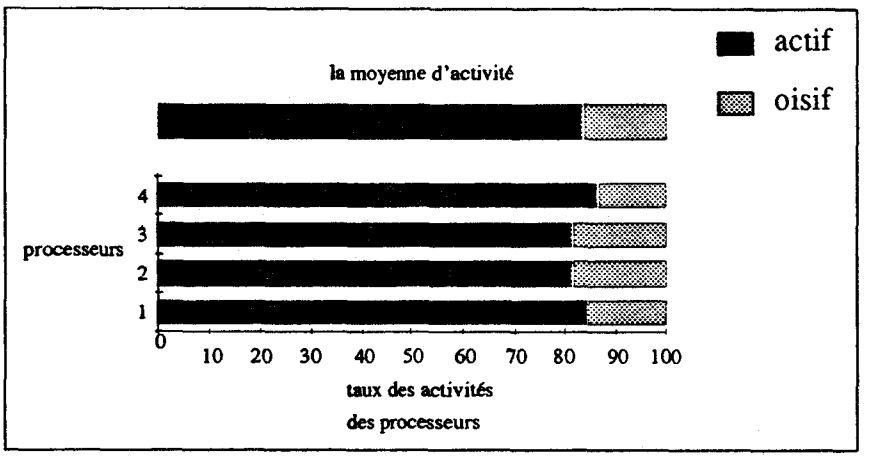



figure 6.6 : Taux d'occupation des processeurs dans le programme ***mutation*** (deepmost)

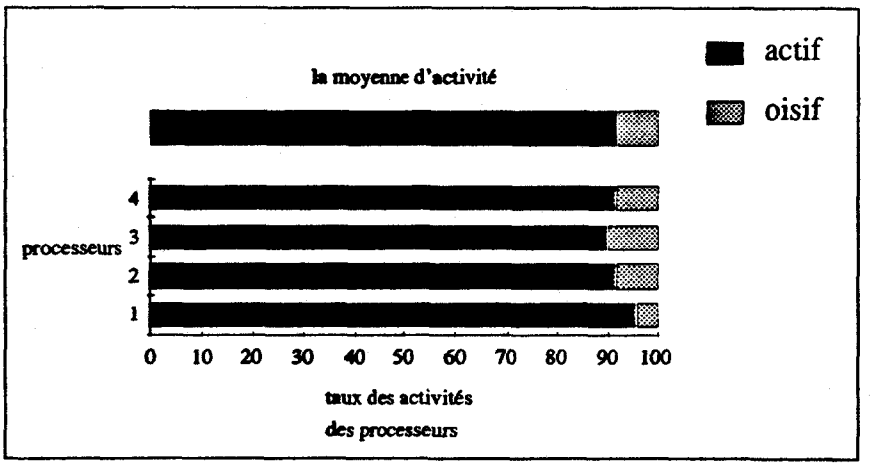



figure 6.7 : Taux d'occupation des processeurs dans le programme ***mutation*** (topmost)

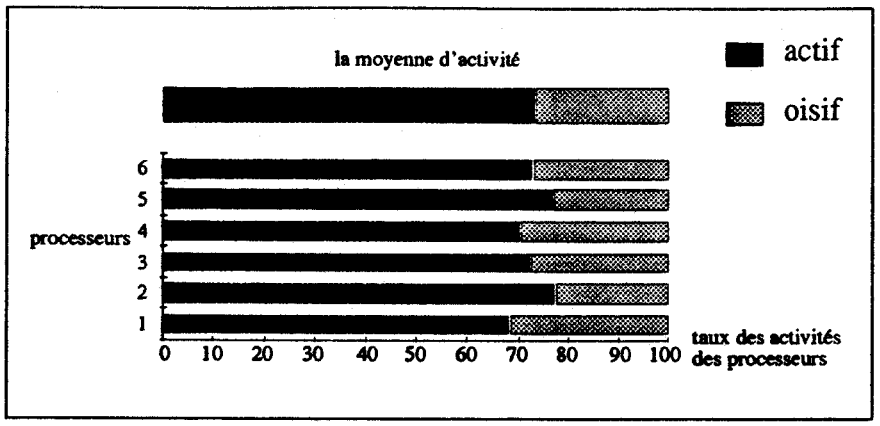



figure 6.8 : Taux d'occupation des processeurs dans le programme ***Mu*** (*leftmost*)

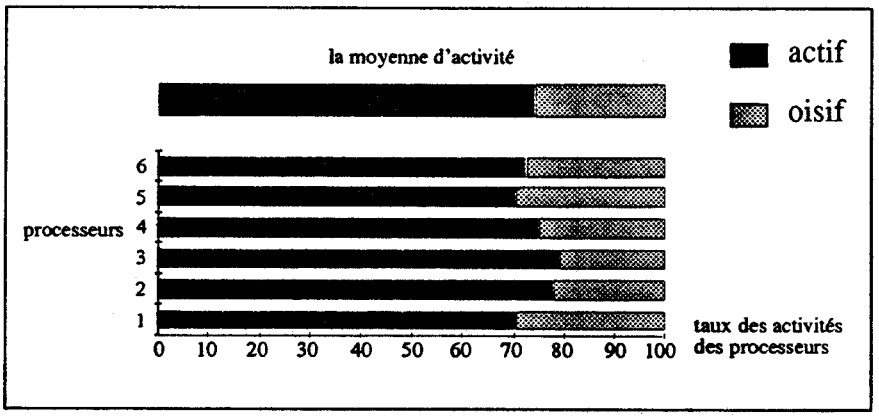



figure 6.9 : Taux d'occupation des processeurs dans le programme ***Mu*** (*deepmost*)

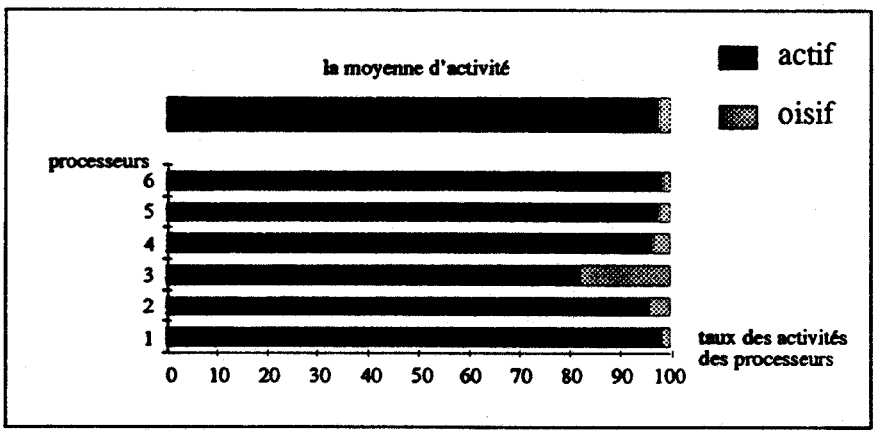



figure 6.10 : Taux d'occupation des processeurs dans le programme ***Mu*** (*topmost*)

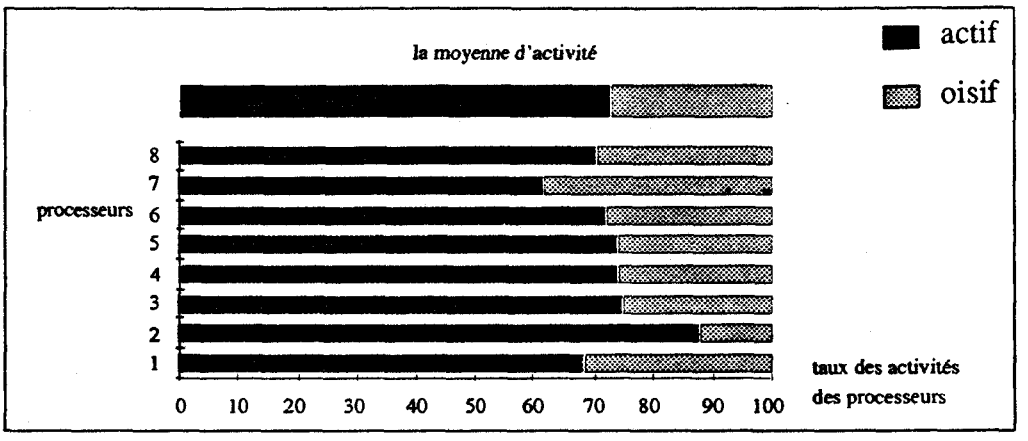



figure 6.11 : Taux d'occupation des processeurs dans le programme ***ham*** (*leftmost*)

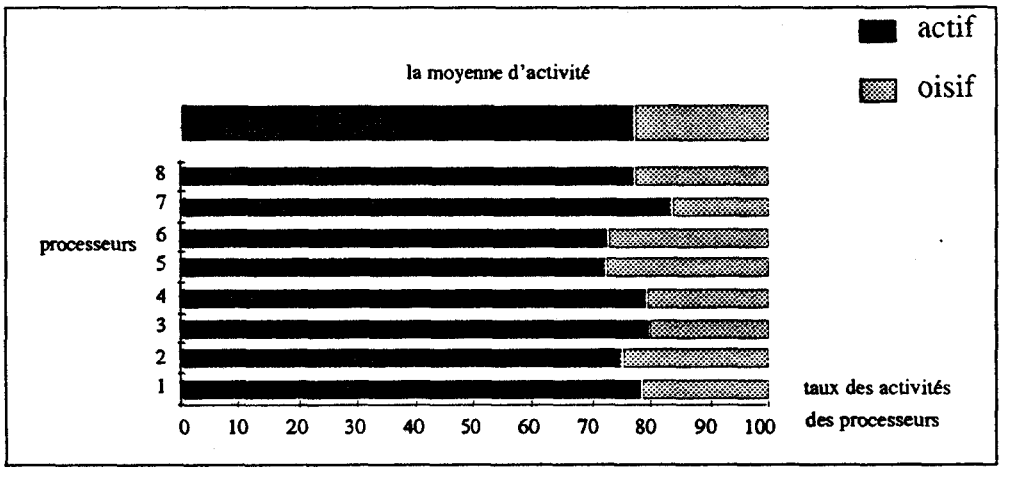



figure 6.12 : Taux d'occupation des processeurs dans le programme ***ham*** (*deepmost*)

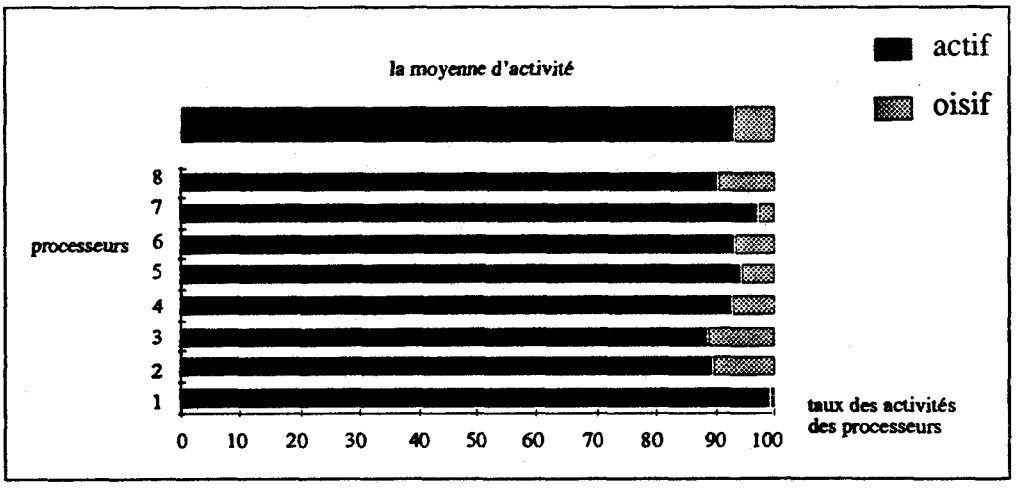



figure 6.13 : Taux d'occupation des processeurs dans le programme ***ham*** (*topmost*)

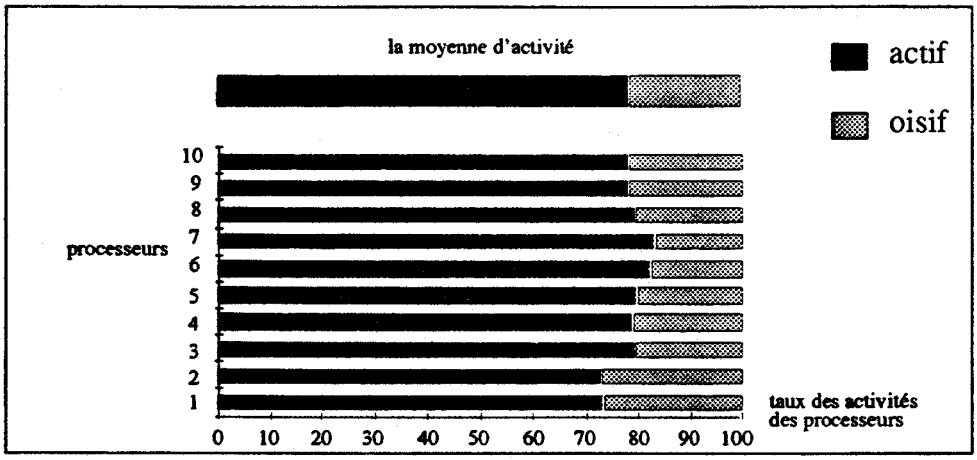



figure 6.14 : Taux d'occupation des processeurs dans le programme ***interrogation 2*** (leftmost)

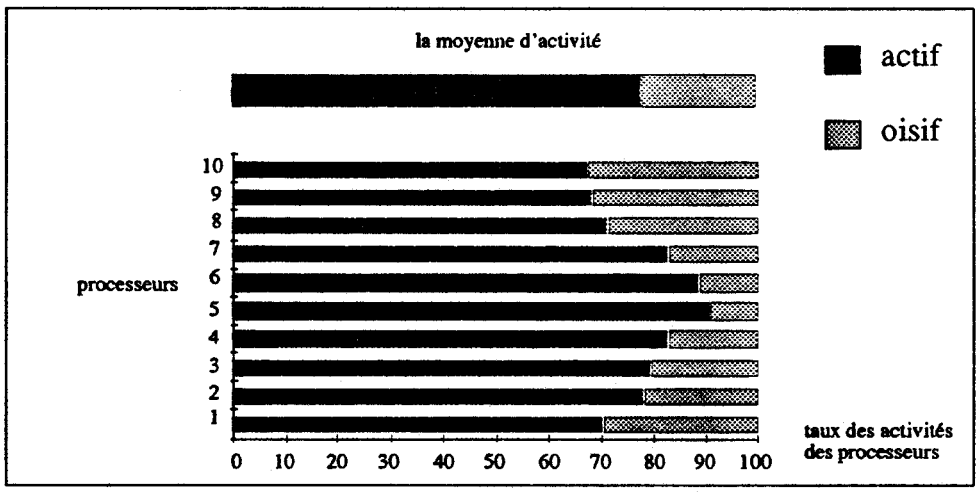



figure 6.15 : Taux d'occupation des processeurs dans le programme ***interrogation 2*** (deepmost)

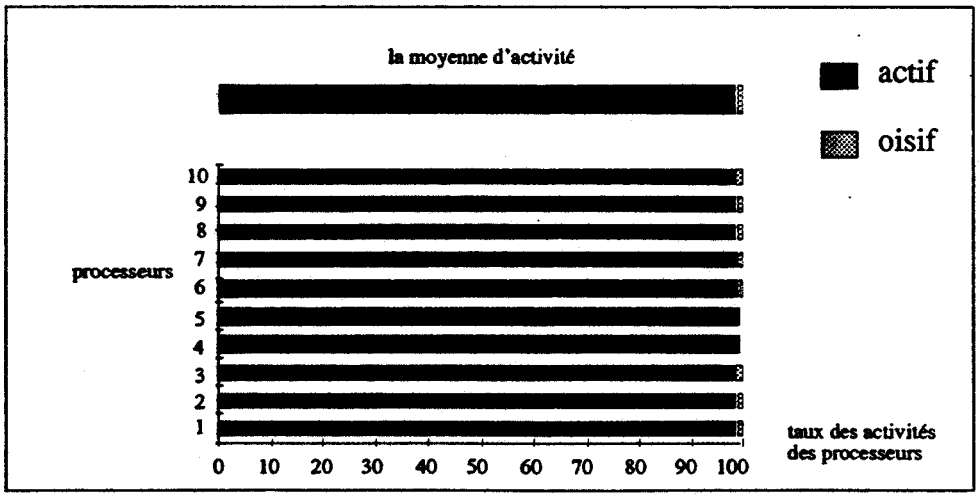



figure 6.16 : Taux d'occupation des processeurs dans le programme ***interrogation 2*** (topmost)

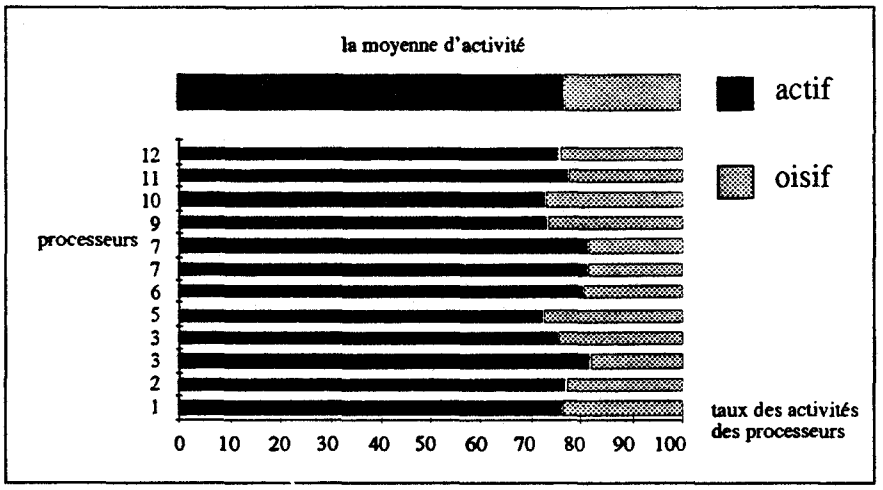



figure 6.17 : Taux d'occupation des processeurs dans le programme ***8-reines*** (leftmost)

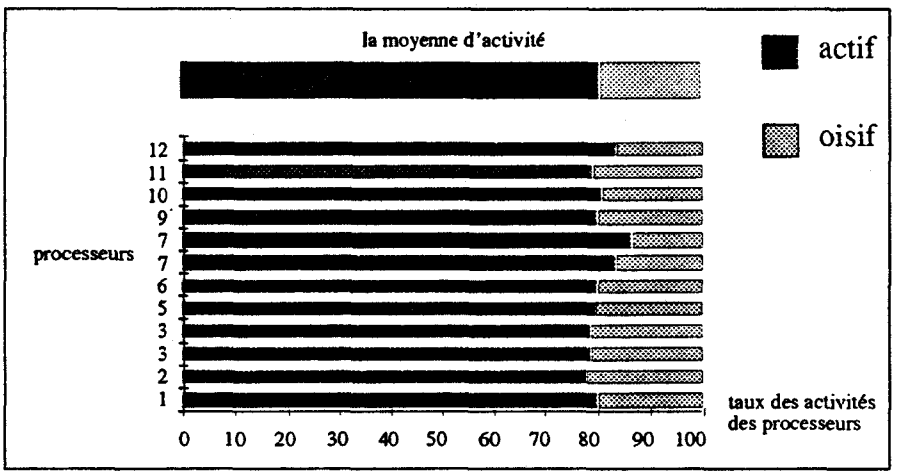



figure 6.18 : Taux d'occupation des processeurs dans le programme ***8-reines*** (deepmost)

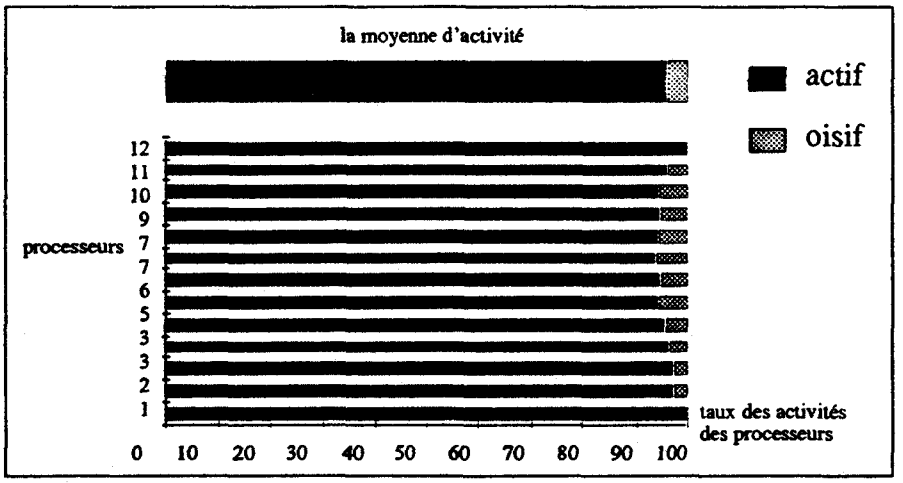



figure 6.19 : Taux d'occupation des processeurs dans le programme ***8-reines***(topmost)

Au vu de ces résultats, la stratégie *topmost* est plus efficace dans tous les programmes et pour toutes les configurations essayées. Le taux d'activité moyen est plus élevé que ceux obtenus par les deux autres stratégies. Ceci est principalement dû aux changements de tâches plus fréquents dans le cas de la stratégie *deepmost*, et plus particulièrement dans la stratégie *leftmost*.

Dans l'ensemble, les trois stratégies offrent une répartition équilibrée de la charge du travail sur les processeurs. Relativement au taux d'activité moyen, l'écart type est entre 10% (obtenu dans le programme *map* avec la stratégie *leftmost*) et 15% (obtenu dans le programme interrogation 2 avec la stratégie *deepmost*).

### 2.3.2 - Gain de performances *(speedup)* :

Nous avons utilisé la stratégie *topmost*, du fait de ses meilleurs résultats, pour évaluer le gain de performances en fonction du nombre de processeurs utilisés.

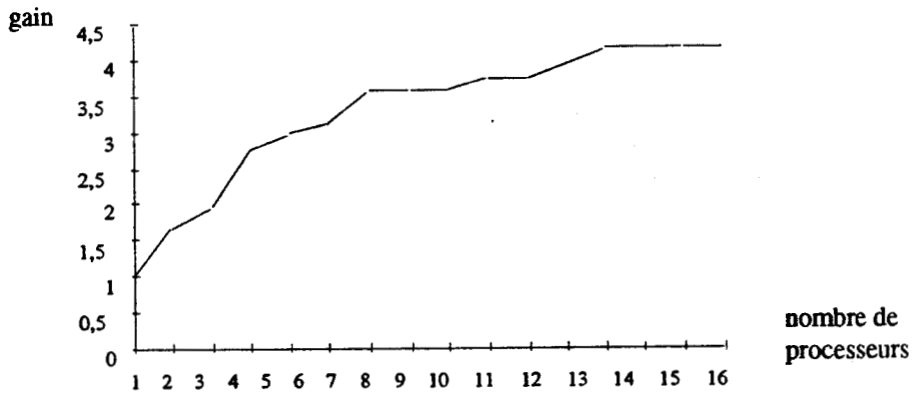
Les mêmes programmes ont été exécutés pour différentes configurations allant de 2 à 16 processeurs. Le tableau 6.9 donne le temps (logique) en cycles, de chaque exécution.

| Processeurs / programmes | 2 Prs | 3 Prs | 4 Prs | 5 Prs | 6 Prs | 7 Prs | 8 Prs | 9 Prs | 10 Prs | 11 Prs | 12 Prs | 13 Prs | 14 Prs | 15 Prs | 16 Prs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| map | 45 | 38 | 27 | 25 | 24 | 21 | 21 | 21 | 20 | 20 | 19 | 18 | 18 | 18 | 18 |
| mutation | 245 | 179 | 126 | 94 | 84 | 80 | 77 | 65 | 61 | 59 | 54 | 54 | 45 | 44 | 48 |
| Mu | 599 | 405 | 307 | 247 | 208 | 186 | 163 | 149 | 134 | 127 | 119 | 117 | 111 | 101 | 96 |
| permuter | 3196 | 3044 | 848 | 634 | 525 | 476 | 421 | 384 | 344 | 317 | 286 | 286 | 266 | 255 | 249 |
| ham | 9078 | 6349 | 5016 | 4065 | 3589 | 3021 | 2867 | 2611 | 2320 | 2150 | 2004 | 1932 | 1781 | 1572 | 1477 |
| interrog 1 | 2589 | 1358 | 698 | 550 | 465 | 397 | 363 | 340 | 309 | 284 | 267 | 250 | 230 | 213 | 200 |
| interrog 2 | 74238 | 49178 | 37018 | 29730 | 24586 | 20963 | 18516 | 16620 | 14735 | 13539 | 12491 | 11749 | 10975 | 10201 | 9596 |
| 8-reines | 26878 | 17752 | 13101 | 10631 | 9034 | 7686 | 6657 | 6182 | 5544 | 5161 | 4721 | 4497 | 4148 | 3929 | 3719 |
| moyenne | 14456 | 9731 | 7143 | 5747 | 4814 | 4104 | 3636 | 3297 | 2933 | 2707 | 2495 | 2363 | 2197 | 2042 | 1925 |

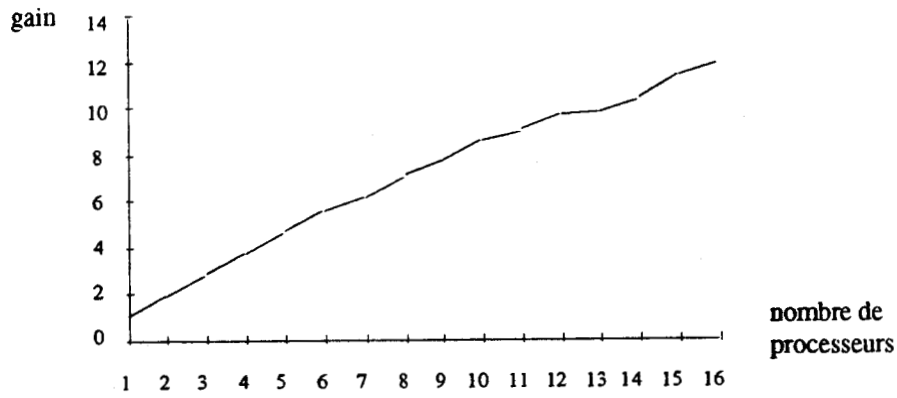
Table 6.9 : Temps (logiques) des exécutions des programmes de test.

Les résultats ci-dessus ont permis de tracer les courbes de gain de performances en fonction du nombre de processeurs. Ce gain est défini comme étant le rapport du temps d'exécution mis par un processeur (donné dans le tableau 6.9) sur le temps d'exécution mis par N processeurs.

Les courbes ci-dessous, concernant le gain de performance montrent un gain linéaire dans les programmes massivement parallèles (ham, 8-reines, interrogation 2). Pour les autres programmes, qui ne présentent pas assez de parallélisme, l'ajout de processeurs à partir d'une certaine limite n'améliore plus les performances du système.

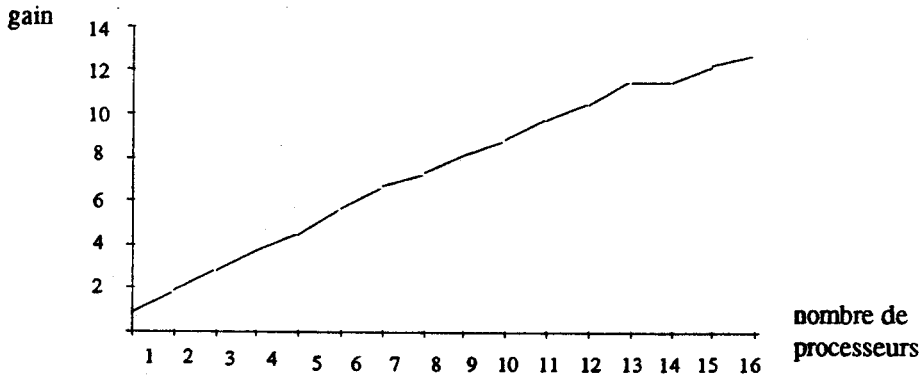



gain de parallélisme dans ***map***

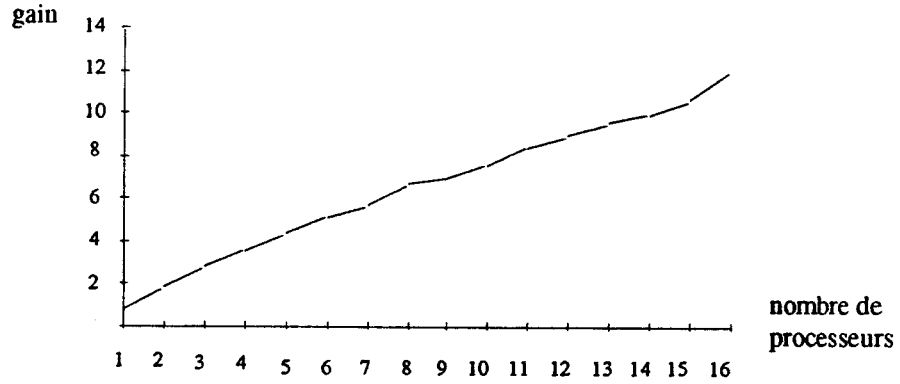



gain de parallélisme dans ***Mu***

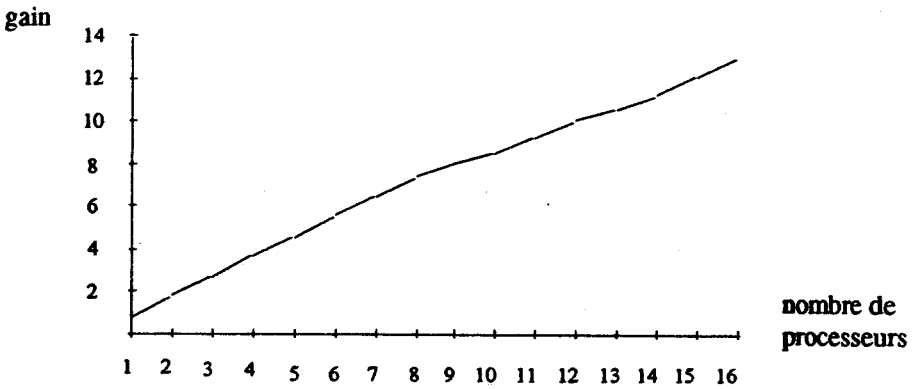

gain de parallélisme dans ***mutation***



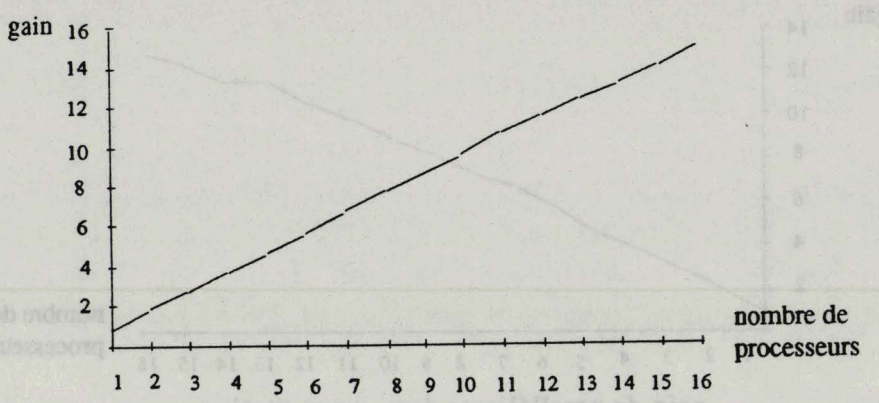

gain de parallélisme dans ***permutation***



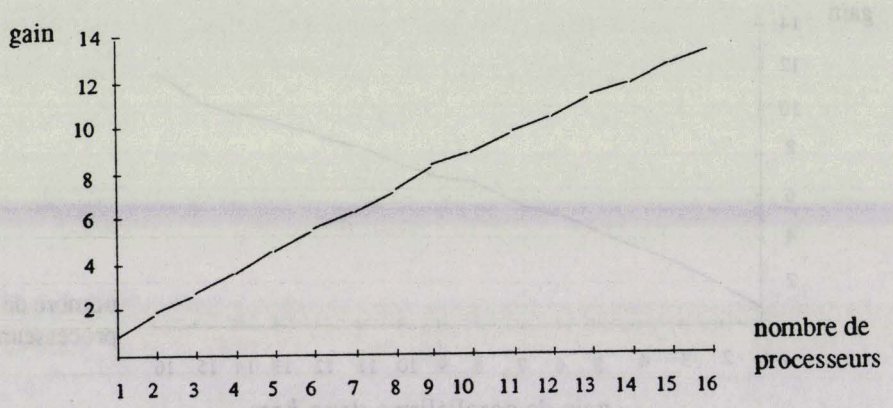

gain de parallélisme dans ***ham***



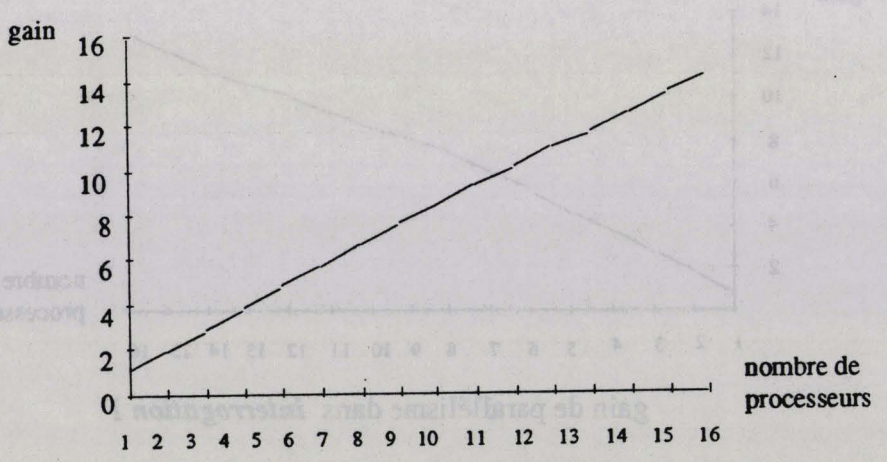

gain de parallélisme dans ***interrogation 1***




gain de parallélisme dans ***interrogation 2***




gain de parallélisme dans ***8-reines***




gain moyen de parallélisme

### 2.3.3 - Comparaison avec une implantation séquentielle :

Le niveau de simulation ne permet pas de comparer les performances de notre modèle pour une configuration quelconque de processeurs, à une implantation séquentielle de Prolog, ni même à d'autres systèmes de programmation logique parallèles. Cependant, il a semblé raisonnable de comparer les performances de la machine séquentielle (microlog 2) qui a servi de base à cette simulation, à celles obtenues par le simulateur pour une configuration de un processeur. Cette comparaison donne le taux de déperditions dues au mécanisme de liaison employé, et à la gestion des structures (pile de choix, tableau de liaisons) et des champs supplémentaires introduits par le modèle d'exécution.

Pour cela, nous avons évalué 11 programmes dont trois sont déterministes (*qsort, nreverse, fibonacci*) et ne présentent aucun parallélisme pour notre modèle d'exécution. Pour chacun d'entre eux, nous avons relevé le temps réel d'exécution (temps CPU en secondes) obtenu par microlog2, puis celui obtenu par le simulateur pour une configuration de un processeur. Le temps logique (mis entre parenthèses dans le tableau 6.10) est également donné pour les exécutions des programmes sur le processeur virtuel.

| Programmes testés | temps (en sec) mis par la microlog 2 | temps (en sec) mis par Un processeur | taux de déperditions |
|---|---|---|---|
| map | 1.1 | 1.4 (75) | 27% |
| mutation | 5.3 | 7.3 (409) | 38% |
| Mu | 17 | 22.5 (1154) | 33% |
| permutation | 37 | 54 (3047) | 46% |
| ham | 234 | 327 (17780) | 39% |
| interrogation 1 | 34 | 47 (2629) | 38% |
| interrogation 2 | 1830 | 2596 (144211) | 42% |
| 8-reines | 639.5 | 920.75 (49485) | 44% |
| qsort (10) | 0.4 | 0.5 | 25% |
| nreverse(10) | 0.26 | 0.3 | 15% |
| fibonacci (15) | 12.4 | 12.78 | 3% |

Table 6.10 : Comparaison avec une implantation séquentielle.

Le taux de déperditions moyen observé sur les 11 programmes est de 30%. On peut comparer ces résultats avec ceux obtenus dans les systèmes Aurora et Muse [Ali 91] :

Pour le système Muse (basé sur la recopie d'environnement), le taux moyen mesuré sur un grand nombre de programmes de test est de :

- 5% sur Sequent Symmetry,
- 8% sur BBN Butterfly GP1000,
- 22% sur BBN Butterfly TC2000.

Ces performances sont très inférieures aux nôtres, du fait que les déperditions dans les modèles basés sur la recopie n'apparaissent réellement qu'au delà d'une configuration de un processeur, i.e lorsque le parallélisme a lieu effectivement.

Pour le système Aurora (i.e basé sur la recopie des liaisons conditionnelles valides, voir modèle SRI), le taux moyen de déperditions, relevé sur le même jeu de programmes, est de [Ali 91] :

- 25% sur Sequent Symmetry,
- 30% sur BBN Butterfly GP1000,
- 77% sur BBN Butterfly TC2000.

Si l'on considère les deux premiers résultats ci-dessus, notre taux de déperditions reste donc comparable avec celui introduit par le système Aurora pour une configuration de un processeur.

Le faible taux de déperditions engendré par les programmes déterministes s'explique par le peu de créations de points de choix, et par conséquent par un nombre de liaisons profondes très limité (accès plus rapide lorsque le mécanisme de liaison superficielle est utilisé).

## 2.4 - Evaluation des programmes à effets de bord :

Le but de cette évaluation est d'étudier le comportement des stratégies d'allocation des travaux aux processeurs, lors des programmes faisant apparaître des effets de bord.

Nous avons considéré deux types d'effet de bord : le premier est introduit dans les programmes faisant appel aux entrée-sorties, tandis que le deuxième est provoqué par l'utilisation du coupe-choix.

### 2.4.1 - Programmes faisant appel aux entrées-sorties :

Nous avons considéré trois autres versions des programmes *mutation*, *interrogation 1* et *8-reines*, où l'affichage des solutions est demandé explicitement par le programmeur à travers le prédicat *write* (le choix de ces programmes est arbitraire). L'exécution de ces programmes a été réalisée en respectant la sémantique "séquentielle" du prédicat *write*, ce qui revient, dans ce cas, à sortir les solutions dans le même ordre que celui obtenu par la stratégie *en profondeur d'abord.*

Pour une configuration de dix processeurs, nous avons relevé le taux moyen d'activité de chaque processeur. Trois états sont envisagés : l'état actif, l'état oisif, lorsque le processeur cherche une tâche à prendre en charge, et l'état suspendu, pendant lequel le processeur attend que la branche traitée devienne la plus à gauche dans l'arbre de recherche.

Afin de contrôler la branche la plus à gauche, la méthode basée sur l'utilisation d'un jeton (présentée dans le chapitre 5) a été utilisée.

**T_log** = temps logique exprimé en cycles.
**T_sim** = temps (CPU) de simulation, exprimé en secondes.
**M_swi** = nombre moyen de changements de tâches par processeur.

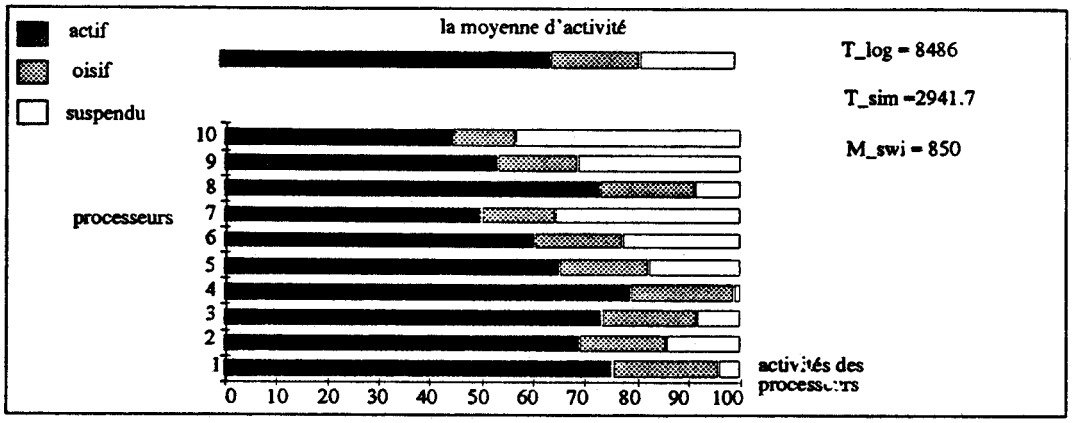



figure 6.20 : Taux des activités des processeurs dans le programme ***8-reines*** (*lefimost*)

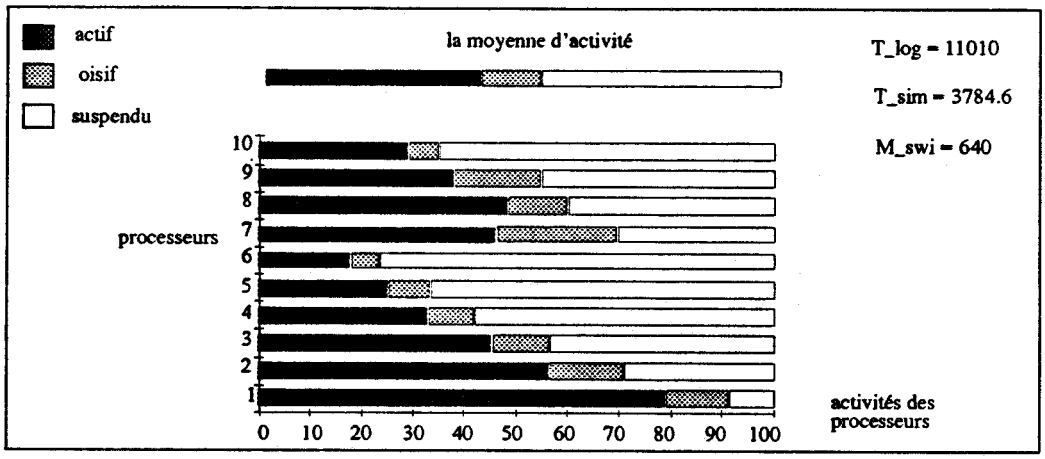



figure 6.21 : Taux des activités des processeurs dans le programme ***8-reines*** (deepmost)

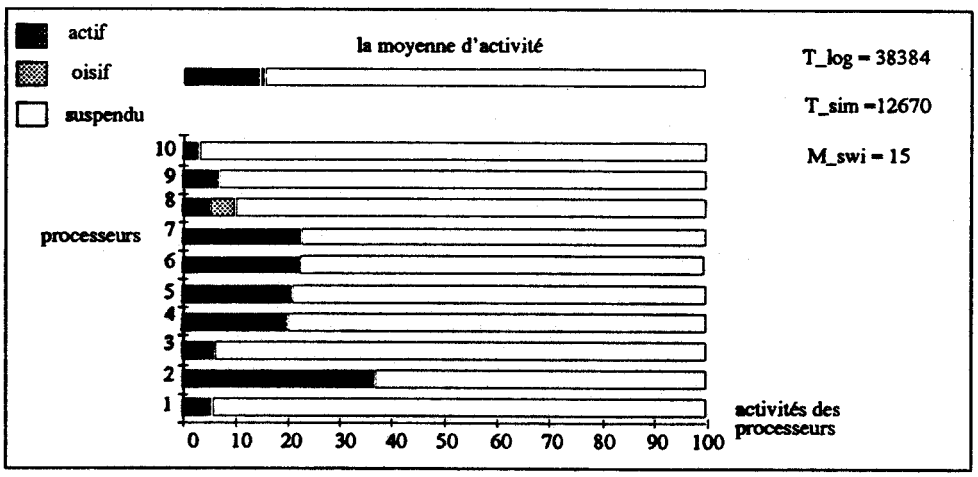



figure 6.22 : Taux des activités des processeurs dans le programmes ***8-reines*** (topmost)

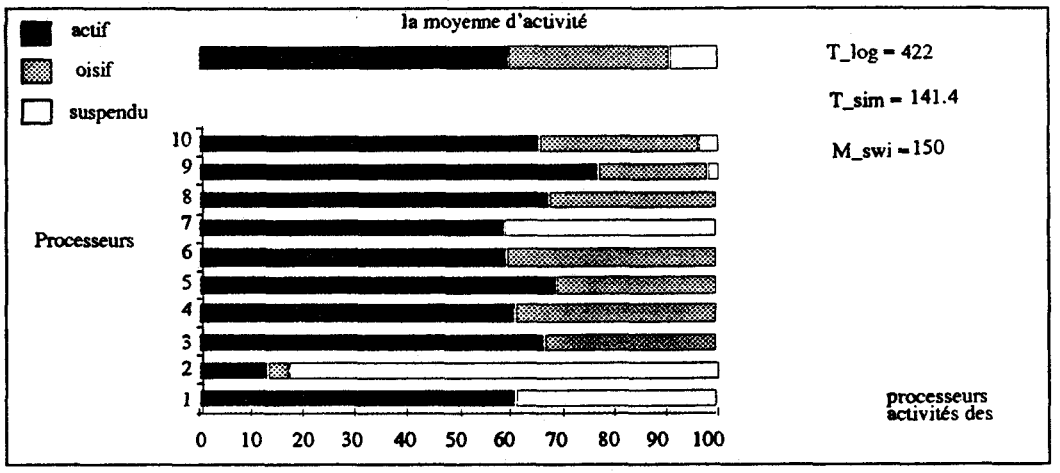



figure 6.23 : Taux des activités des processeurs dans le programme ***interrogation 1*** (*leftmost*)

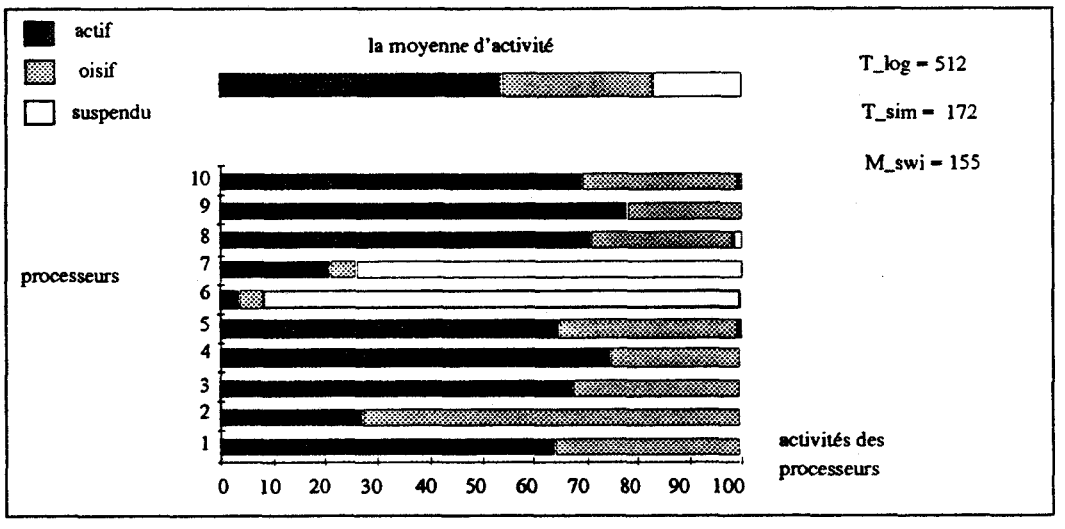



figure 6.24 : Taux des activités des processeurs dans le programme ***interrogation 1*** (*deepmost*)

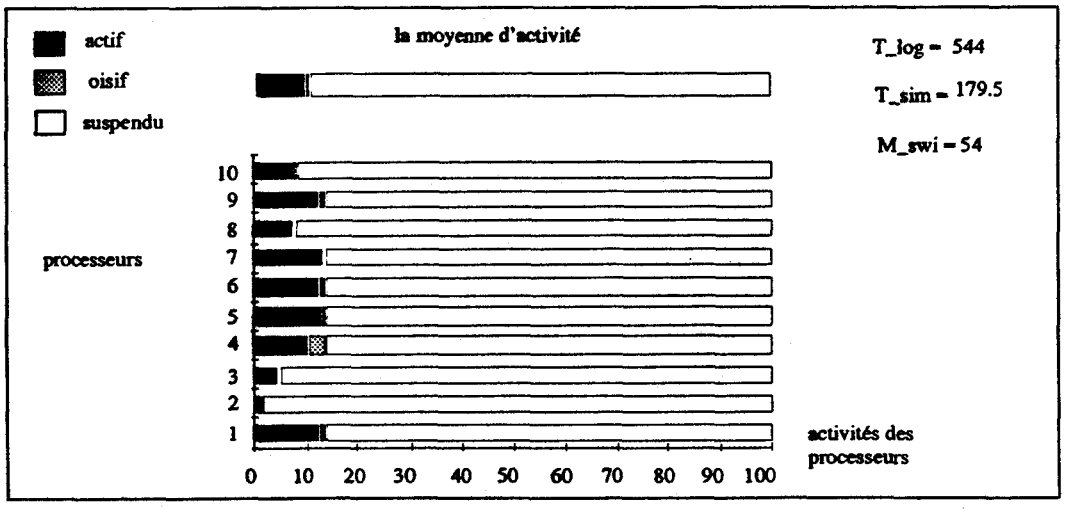



figure 6.25 : Taux des activités des processeurs dans le programme ***interrogation 1*** (*topmost*)

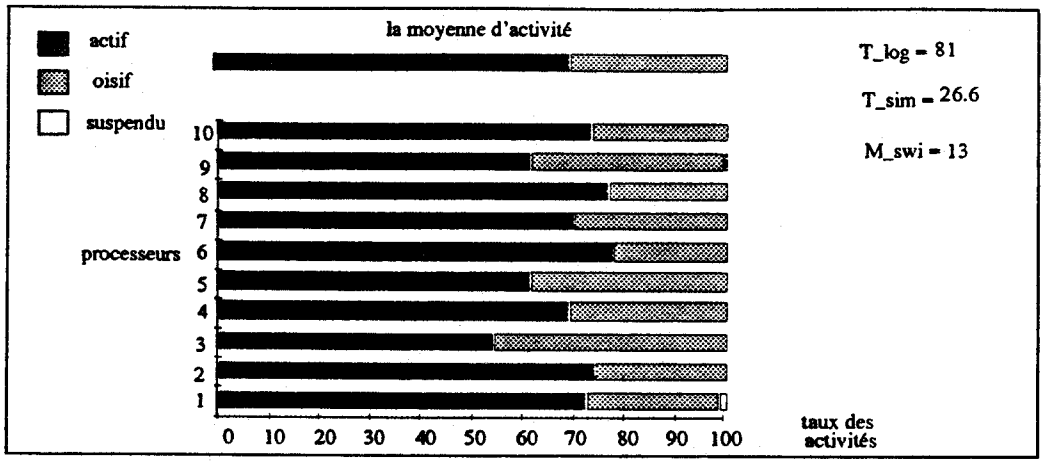



figure 6.26 : Taux des activités des processeurs dans le programme ***mutation*** (*leftmost*)

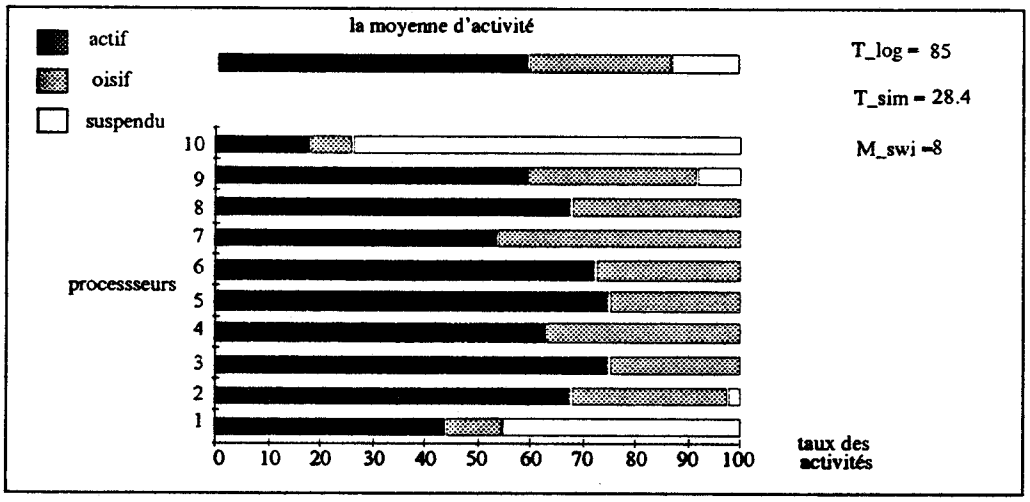



figure 6.27 : Taux des activités des processeurs dans le programme ***mutation*** (*deepmost*)

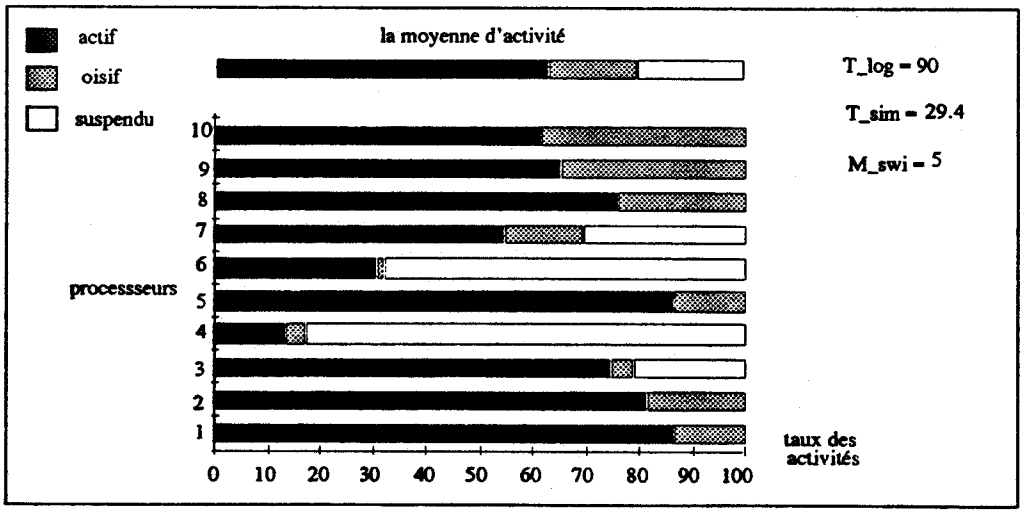



figure 6.28 : Taux des activités des processeurs dans le programme ***mutation*** (*topmost*)

Si dans les programmes sans effets de bord, la stratégie topmost s'est révélée plus efficace en exploitant mieux la granularité des tâches, l'utilisation d'une telle stratégie dans les programmes faisant appel à des entrées-sorties ne garantit pas la même efficacité d'après les résultats ci-dessus.

En effet, le gros grain offert par la stratégie *topmost* "condamne" les processeurs effectuant des effets de bord, à rester plus longtemps dans l'état suspendu (le taux moyen de suspension dans interrogation 1 et 8-reines varie entre 70% et 80%).

La stratégie *leftmost*, de par sa définition, favorise le travail disponible situé à gauche dans l'arbre de recherche, et par conséquent permet d'accélérer la réactivation des processeurs suspendus.

Malgré le nombre plus élevé de changements de tâches entraîné par la stratégie *leftmost*, que celui enregistré dans la stratégie *topmost*, (1.5 (mutation), 3 (interrogation 1) et 56 (8-reines) fois plus grand), la stratégie *leftmost* présente les meilleures performances en temps logique et en temps de simulation. Ceci est principalement dû au coût très réduit d'installation d'une tâche parallèle, offert par le modèle de calcul.

La stratégie *deepmost*, bien que moins efficace que la stratégie *leftmost*, présente de meilleures performances que la stratégie *topmost*.

Remarque :

Dans le programme *mutation*, la stratégie *topmost* présente des résultats assez proches que ceux obtenus par les deux autres stratégies. Ceci s'explique par l'utilisation d'un nombre de processeurs élevé par rapport au peu de parallélisme offert par ce programme, ce qui laisse entendre que, pour réduire les déperditions dues aux suspensions prolongées des processeurs dans la stratégie *topmost*, il faut augmenter leur nombre.

La question est : *de combien faut-il augmenter le nombre de processeurs dans les programmes massivement parallèles et faisant appel aux entrées-sorties* ?.

Mais avant de répondre à cette question, il est judicieux de répondre à la question suivante: *est ce que de tels programmes peuvent exister réellement* ?

Si l'on considère la contrainte drastique consistant à respecter la sémantique opérationnelle des prédicats d'entrées-sorties, les programmes précédents constituent une réponse à la deuxième question.

### 2.4.2 - Travail spéculatif :

Dans cette partie, les programmes introduisant le coupe-choix sont évalués. Comme nous l'avons mentionné au début de ce chapitre, le coupe-choix n'a pas été intégré réellement dans le simulateur. L'implantation d'un coupe-choix dans un système OU-parallèle nécessite d'une part, un contrôle de la branche la plus à gauche, et d'autre part, un mécanisme permettant d'arrêter les exécutions des branches se trouvant dans le champ d'un coupe-choix, lorsque ce dernier est autorisé à être exécuté. C'est ce dernier mécanisme qui n'a pas été pris en compte actuellement dans le simulateur. Pour éviter le besoin d'un tel mécanisme, nous avons considéré uniquement les programmes dont l'exécution du coupe-choix coïncide avec la fin du programme.

Pour cela, il a été nécessaire de constituer une base de programmes significatifs et qui respectent, en même temps, la contrainte précédente, afin d'évaluer les performances de chacune des trois stratégies (*leftmost, deepmost* et *topmost*).

La constitution de cette base de programmes de test s'appuie dans un premier temps, sur une technique de modélisation du travail spéculatif, et dans un deuxième temps, sur la réutilisation des programmes précédents en y introduisant un coupe-choix permettant de délivrer une seule solution à la question initiale.

#### 2.4.2.1 - Modélisation du travail spéculatif :

Pour effectuer des mesures de performances, nous avons utilisé une technique proposée dans [Bea 91], qui consiste à modéliser le travail spéculatif à partir d'un programme de recherche offrant beaucoup de solutions. L'idée est d'utiliser un coupe-choix pour arrêter la recherche lorsqu'une solution spécifiée au départ est trouvée. L'exploration en parallèle des branches se trouvant à droite de la branche donnant la solution, constitue le travail spéculatif (travail "gaspillé").

Le programme de recherche que nous avons utilisé permet de générer toutes les permutations d'une liste donnée :

```
permutation([],[]).
permutation(X, [Z|Zs]) :- select(Z,X,Y), permutation(Y,Zs).

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

query (List, GP) :- permutation (List, Lperm), Lperm= GP, !.
```

figure 6.29 : Modélisation du travail spéculatif

La "procédure" *permutation* permet de générer, dans un ordre lexicographique, toutes les permutations possibles d'une liste donnée. La "procédure" *query* permet de vérifier si la liste *GP* est une permutation de la liste *List*; les deux paramètres sont donc en entrée lors de l'appel du but *query*. Le prédicat *cut* permet d'arrêter la résolution dès qu'une solution est trouvée.

Pour évaluer la quantité du travail spéculatif, nous avons mesuré le nombre total de résolutions (appels de clauses) effectuées par les processeurs, ainsi que le nombre de résolutions effectuées en utilisant la stratégie en profondeur d'abord (*dephfirst*). La différence de ces deux nombres donne la quantité du travail "gaspillé".

Pour effectuer plusieurs mesures, nous avons fait varier la quantité du travail spéculatif en faisant varier la liste de départ *GP* ("*Goal Pattern*"). Deux séries de mesures ont été relevées. La première concerne la variation de *GP* dans le segment S1=[[1,2,3,4,5,6,7,8] . . . . .[1,8,7,6,5,4,3,2]], tandis que la deuxième est obtenue en considérant une valeur de *GP* dans chacun des 8 segments de solutions possibles :

S1=[[1,2,3,4,5,6,7,8] . . .[1,8,7,6,5,4,3,2]]. . . S2=[[2,1,3,4,5,6,7,8] . . . .[2,8,7,6,5,4,3,1]], . . . et

S8= [[8,,1,2,3,4,5,6,7]. . . .[8,7,6,5,4,3,2,1]].

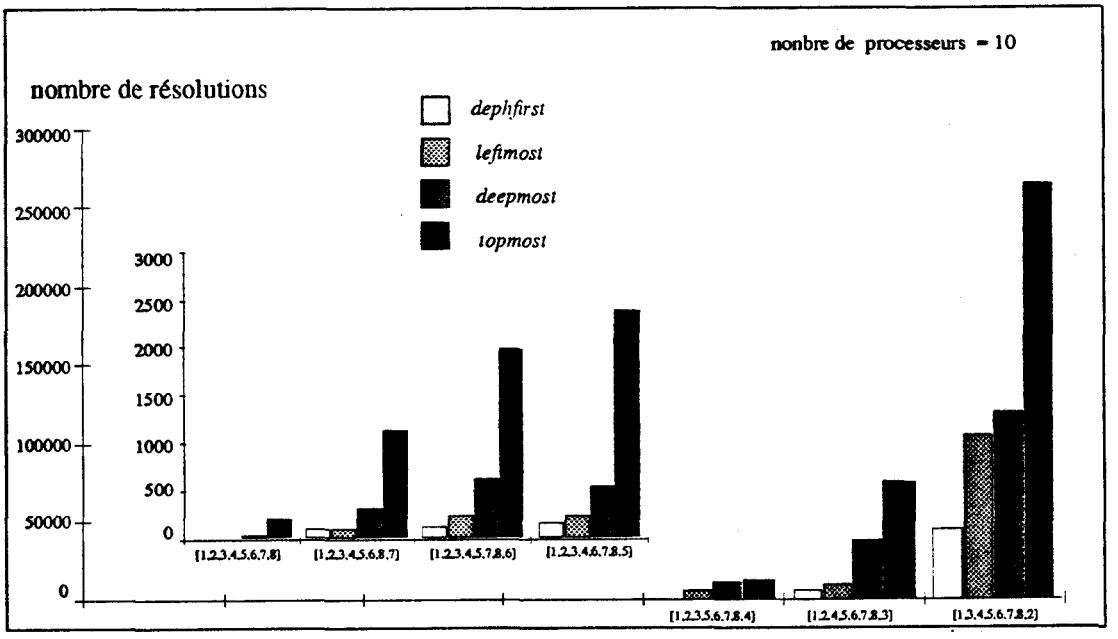



figure 6.30 : Comparaison du nombre de résolutions obtenus par chacune des stratégies, avec celui obtenu par la stratégie *dephfirst* (segment S1).

La figure 6.30 montre le nombre total de résolutions effectuées par les 10 processeurs, pour chacune des stratégies. A titre de comparaison, nous avons fourni les résultats obtenus par la stratégie séquentielle (*dephfirst*).

Le nombre de résolutions relevé dans la stratégie *topmost* est nettement plus important que ceux obtenus par la stratégie *leftmost* et *deepmost*. La quantité du travail spéculatif est donc plus élevée dans la stratégie *topmost*, et augmente lorsque la solution GP se déplace vers la gauche de l'arbre de recherche. Ceci se traduit, dans la figure 6.31, par des temps de simulations plus lents lorsque la stratégie *topmost* est utilisée.

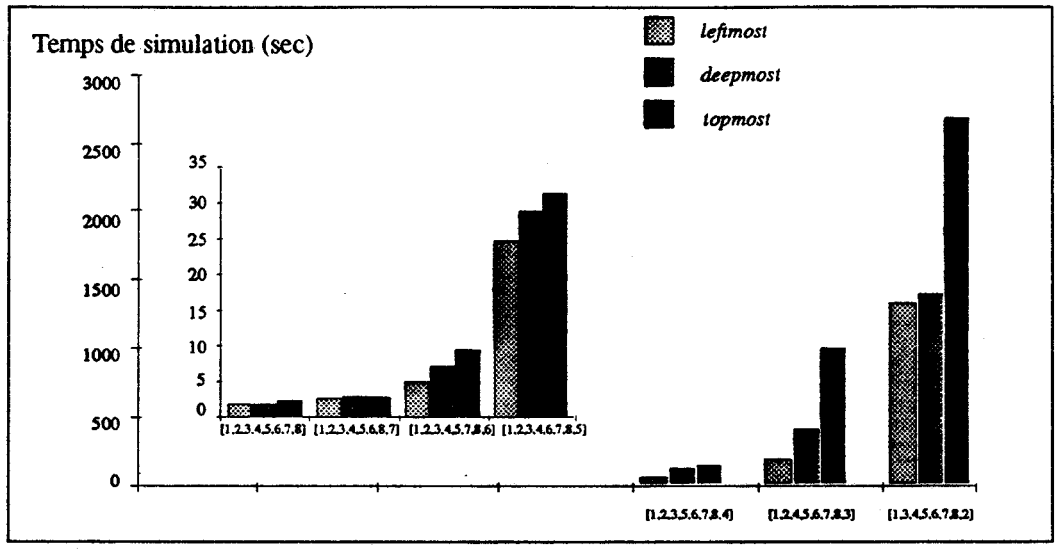



figure 6.31 : Temps des simulations obtenus par les trois stratégies (segment S1).

Au vu de ces résultats, la stratégie *leftmost* offre les meilleures performances, en introduisant moins de travail spéculatif. Pour les deux premières solutions (valeurs de *GP*), les temps de simulations sont très proches pour les trois stratégies : ceci est dû au fait que le travail spéculatif ne s'est pas suffisamment développé.

Dans les résultats de la figure 6.32, nous avons fait varier la solution GP en considérant une valeur dans chacun des segments S1 ... S8.

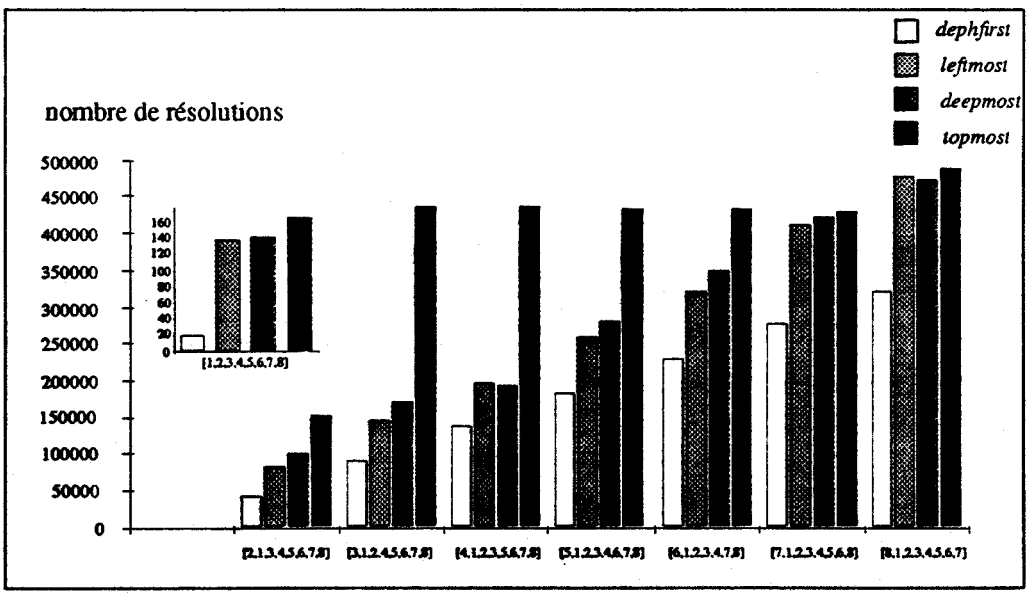



figure 6.32 : Comparaison du nombre de résolutions obtenus par chacune des stratégies, avec celui obtenu par la stratégie *dephfirst* ( GP=[1,...],[2,...],.....[8,...]).

La stratégie *leftmost* reste celle qui introduit le moins de travail spéculatif. Nous constatons qu'au fur et à mesure que la solution *GP* se déplace vers la droite dans l'espace des

solutions, la quantité du travail spéculatif produite par la stratégie *deepmost* se rapproche de celle obtenue par la stratégie *leftmost*. De la même façon, le taux du travail spéculatif obtenu par la stratégie *topmost* diminue, puisque la zone de l'arbre de recherche engendrant le travail spéculatif diminue en taille.

La figure 6.33 donne les temps des simulations précédentes.

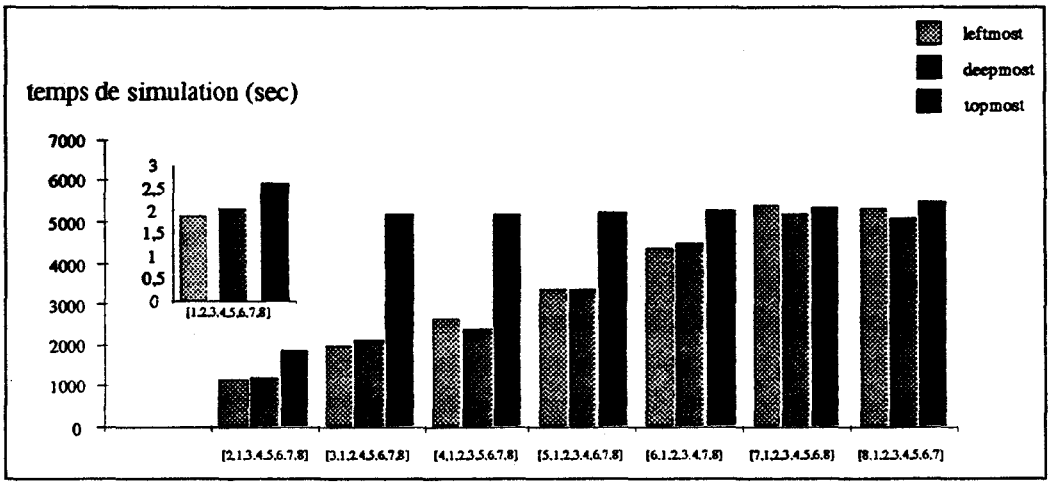



figure 6.33 : Temps des simulations obtenus par les trois stratégies (GP=[1,...],[2,...],...[8,...]).

Pour les trois premières valeurs de *GP*, la stratégie *leftmost* reste plus performante que les deux autres.

Dans les trois solutions suivantes, la stratégie *deepmost* se rapproche de la stratégie *leftmost*, mais l'écart entre les deux est aléatoire.

Malgré un nombre de résolutions plus élevé obtenu par la stratégie *topmost* pour les deux dernières valeurs de *GP*, les temps de simulations correspondant à celles-ci sont proches de ceux obtenus par le stratégie *leftmost*, mais restent supérieurs à ceux enregistrés par la stratégie deepmost. Ceci s'explique par le fait qu'au fur et à mesure que la solution *GP* se déplace à droite, d'une part, la quantité du travail spéculatif diminue, et d'autre part, l'espace de recherche devient plus important. Ce dernier point a entraîné une augmentation des changements de tâches dans la stratégie *leftmost*, ce qui a amoindri le gain obtenu par une quantité de travail spéculatif moins élevée.

Afin de comparer les taux de changements de tâches obtenus par les trois stratégies, nous avons considéré la dernière solution (*GP*=[8,7,6,5,4,3,2,1]) fournie par la stratégie séquentielle.

Le nombre de branches pouvant faire l'objet d'un travail spéculatif est quasiment nul, pour une telle exécution.

Les figures 6.34 et 6.35 donnent respectivement le temps de simulation mis par chacune des stratégies, ainsi que le nombre total de changements de tâches effectués par les dix processeurs considérés.

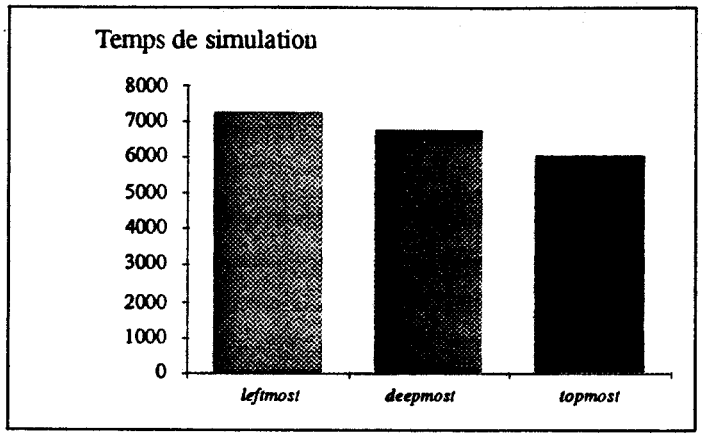



figure 6.34 : Performances des trois stratégies (GP=[8,7,6,5,4,3,2,1]).

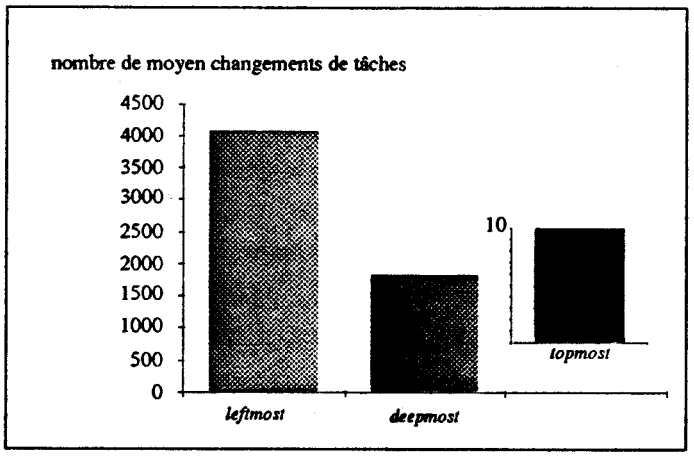



figure 6.35 : Nombres moyens des changements de tâches enregistrés dans les trois stratégies (GP=[8,7,6,5,4,3,2,1])

La stratégie *topmost* offre la meilleure performance, car le grain de parallélisme obtenu introduit moins de changements de tâches que les deux autres stratégies (on retrouve ici la même conclusion que celle obtenue aprs test des programmes sans effets de bord).

Malgré l'écart très important entre le nombre de changements de tâches obtenu par la stratégie *topmost*, et celui obtenu par l'une des deux autres stratégies, les temps de simulation, pour chacune des trois stratégies, ne présentent pas de différence aussi importante. Ceci montre bien que les changements de tâches dans notre modèle n'est pas un facteur critique.

### 2.4.2.2 - Travail spéculatif dans certaines applications réelles :

Après avoir étudié le comportement des stratégies de recherche des travaux, en modélisant le travail spéculatif, nous nous sommes intéressés à de réelles applications qui peuvent engendrer le travail spéculatif à cause de l'utilisation du parallélisme.

Les programmes précédents ont été réutilisés dans le cadre de cette évaluation, et réécrits pour ne fournir qu'une solution à la question initiale. Ce type de programmation est fréquemment utilisé en Prolog : parfois, seule une solution à un problème dont l'espace de recherche est très important, est suffisante pour l'utilisateur. Les branches se trouvant à droite de la branche donnant cette solution, constituent un travail spéculatif.

Le tableau ci-dessous donne les performances, en terme de temps de simulation, pour chacune des stratégies, et pour une configuration de dix processeurs.

| Programmes testés | *leftmost* | *deepmost* | *topmost* |
|---|---|---|---|
| map | 2.6 | 3.5 | 4 |
| mutation | 5.25 | 5 | 9 |
| ham | 19 | 21 | 24.5 |
| interrogation 1 | 118 | 135 | 125 |
| interrogation 2 | 5365 | 10132 | 9976 |
| 8-reines | 169 | 363 | 604 |
| moyenne | 5678.8 | 10659.5 | 10742.5 |

Table 6.11 : performances des 10 processeurs cherchant la première solution.

Les résultats présentés dans le tableau 6.11, confirment l'efficacité de la stratégie *leftmost*, dans les programmes introduisant le travail spéculatif. La stratégie *deepmost* reste en moyenne meilleure que la stratégie *topmost*.

Il est donc intéressant d'utiliser, dans les programmes de tailles importantes, et dont le but est de trouver une seule solution, une stratégie comme *leftmost* qui permet d'accélérer la recherche de cette première solution.

## 3 - Conclusion :

Dans la première partie de chapitre, nous avons décrit le simulateur de la machine abstraite parallèle. Nous avons montré ses limites et la façon dont les mesures ont été relevées. Trois versions du simulateur ont été réalisées, qui sont basées sur les méthodes de liaisons (table de hachage, vecteurs de liaisons et tableaux de liaisons) permises par le modèle du calcul.

Nous avons ensuite proposé trois stratégies de recherche des travaux, appliquées aux processeurs oisifs. La première (*leftmost*) consiste à favoriser le travail situé le plus à gauche dans l'arbre de recherche, la deuxième (*deepmost*) a pour but de répartir la recherche dans le fond de l'arbre de recherche, et enfin, la troisième (topmost), consiste à concentrer la recherche des travaux à proximité de la racine de l'arbre de recherche.

Dans la première partie des évaluations effectuées, nous nous sommes penchés sur l'étude des trois méthodes de liaisons proposées. Les résultats obtenus ont permis d'écarter la méthode basée sur le hachage à cause de son taux élevé de déperditions du aux appels fréquents du mécanisme de hachage.

Les résultats concernant la méthode basée sur l'utilisation des vecteurs de liaisons ont mis en évidence le besoin de synchroniser les créations de vecteurs; ceci s'est traduit dans notre simulateur, par des temps de simulations plus lents à cause des appels du mécanisme d'exclusion mutuelle à chaque création de vecteur.

La méthode basée sur l'utilisation des tableaux de liaisons a montré de meilleures performances, du fait de l'accès aux liaisons profondes, qui est plus rapide, et ne nécessite aucune synchronisation entre les processeurs. Nous avons sélectionné celle-ci pour le reste des évaluations.

Dans cette même série d'évaluations, nous avons donné le taux de déréférencement nécessitant la recherche d'une liaison profonde valide, ainsi que le nombre moyen d'accès aux liaisons profondes lors d'une recherche. Les résultats obtenus pour les programmes testés, ont montré que ces deux chiffres sont assez faibles. Ceci est principalement dû à l'utilisation d'une stratégie *topmost*, qui, comme nous l'avons vu, permet d'augmenter le grain du parallélisme, et par conséquent, offre une localité des références aux processeurs.

La deuxième partie des évaluations avait pour but d'étudier le comportement des trois stratégies dans les programmes sans effets de bord, et dans ceux utilisant les prédicats extra-logiques *write* et *cut*.

Les résultats obtenus pour la première catégorie de programmes, ont montré l'efficacité de la stratégie *topmost*. Celle-ci introduit moins de changements de tâches, et exploite mieux la granularité des tâches.

Nous avons ensuite évalué le gain de performances en utilisant la stratégie *topmost* : ce gain est linéaire dans les programmes massivement parallèles.

Les résultats obtenus, lorsque des effets de bord sont introduits dans les programmes, ont montré la situation inverse. Ainsi, une stratégie comme *leftmost* ou *deepmost*, offrant une grain de parallélisme plus fin, permet de mieux optimiser les activités des processeurs, en accélérant la réactivation des processeurs suspendus suite à un effet de bord, et en évitant des quantités importantes de travaux spéculatifs engendrés par l'utilisation du coupe-choix dans un environnement parallèle.

# CONCLUSION GENERALE

L'exploitation du parallélisme en programmation logique a soulevé, ces dix dernières années, un vif intérêt dans la communauté scientifique. De nombreux projets ont été lancés dans diverses directions, afin de découvrir quelles étaient les sources potentielles et les sources réellement exploitables de parallélisme.

A ce jour, il semble irréfutable que l'exploitation du parallélisme-OU présent dans de nombreuses application de type intelligence artificielle, soit la source de parallélisme la plus prometteuse en terme d'accélération d'exécution.

L'approche OU-multiséquentielle est actuellement l'approche qui a montré au mieux son aptitude et sa capacité à s'implanter efficacement sur les architectures parallèles présentes sur le marché (Sequent-Balance, Encore-Alliant).

Pour cette raison, nous avons réexaminé l'un des points fondamentaux d'une implantation du langage Prolog sur un multiprocesseur, à savoir la gestion des liaisons multiples induites par le parallélisme-OU. Nous avons pu montrer que les méthodes utilisées dans ce domaine (PEPSys/Argonne, Versions-Vectors, SRI) pouvaient être synthétisées de façon à garder les avantages respectifs de chacune, c'est à dire un temps d'accès aux liaisons qui soit borné, et un temps d'installation de tâches parallèles ne nécessitant aucune recopie d'environnement.

Nous avons proposé un modèle OU-parallèle multi-séquentiel dont les principales caractéristiques sont, en premier lieu, le mécanisme de liaison utilisé. Ce dernier répond aux objectifs précités avec une contrainte mineure limitant, à un instant donné, le nombre de liaison d'une même variable à une et une seule par processeur.

D'autre part, la gestion par pile des tâches entreprises par un processeur, a permis d'utiliser toutes les techniques efficaces de gestion de l'espace, utilisées dans une implantation séquentielle.

Nous avons défini une machine abstraite parallèle pour ce modèle. Une émulation de celle-ci sur une machine séquentielle (station de travail Sun) a permis, dans un premier temps, de valider le modèle de calcul et le mécanisme de liaison utilisé, et dans un deuxième temps, d'évaluer les performances lors de l'exécution de programmes réels.

Nous nous sommes ensuite intéressé aux stratégies de recherche du travail disponible par un processeur (*scheduling*), et trois algorithmes de recherche de travail ont été étudiés puis évalués. Les résultats de simulation ont mis en évidence que pour les programmes sans effets de bord, la stratégie (*topmost*) favorisant le travail disponible en haut de l'arbre de recherche, est la meilleure en terme de temps d'exécution, de part sa granularité plus importante.

Par contre, pour des programmes introduisant des effets de bord, la stratégie (*leftmost*) qui consiste à favoriser le travail disponible se trouvant dans la partie la plus à gauche de l'arbre de recherche, offre les meilleurs performances en réduisant la quantité du travail spéculatif et en accélérant la réactivation des processeurs suspendus momentanément suite à un effet de bord.

Le coût réduit (et constant) de l'installation d'une tâche parallèle, offert par le modèle, laisse envisager une solution mixte combinant les deux stratégies : l'une utilisée pour mieux exploiter la granularité des tâches parallèles, l'autre, au contraire, permettant d'éclater le grain de parallélisme afin de minimiser les déperditions introduites par les effets de bords (synchronisation et travail spéculatif).

Les résultats de simulation qui ont été obtenus ne permettent pas de comparer nos performances à d'autres systèmes existants (Aurora, PEPSys, Muse), mais sont encourageant pour envisager une réelle mise en oeuvre et nous allons, dans les mois à venir, entreprendre l'implantation du modèle sur un multiprocesseur.

# Annexe

## -=- Programmes de test -=-

Programme ***map*** [Per 90] :

```
$map:-carte(A,B,C,blue,red,F,G,H).

$carte(A,B,C,D,E,F,G,H) :- next(A,B),next(A,C),next(A,D),next(B,C),
                           next(C,D),next(C,E),next(D,E),next(D,F),
                           next(E,F),next(E,G),next(F,G),next(F,H),next(G,H).

$next(blue,yellow).
$next(blue,red).
$next(blue,green).
$next(yellox,blue).
$next(yellow,red).
$next(yellow,green).
$next(red,blue).
$next(red,yellow).
$next(red,green).
$next(green,blue).
$next(green,red).
$next(green,yellow).
                                   :- map.
```

Programme ***mutation*** :

```
$mutation(Z):- animal(X),animal(Y),append(A,B,X),dif(B,[]),
               append(B,C,Y),dif(C,[]),append(X,C,Z).

$append([],L,L).
$append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

$animal([a|[l|[l|[i|[g|[a|[t|[o|[r]]]]]]]]]).
$animal([t|[o|[r|[t|[u|[e]]]]]]).
$animal([c|[a|[r|[i|[b|[o|[u]]]]]]]).
$animal([c|[h|[e|[v|[a|[l]]]]]]).
$animal([o|[u|[r|[s]]]]).
$animal([v|[a|[c|[h|[e]]]]]).
$animal([l|[a|[p|[i|[n]]]]]).

                                   :-mutation(X).
```

Programme ***permutation*** :

```
$permutation([],[]).
$permutation(L,[E|P]):-del(L,E,D),permutation(D,P).

$del([H|T],H,T).
$del([H|T],E,[H|L]):-del(T,E,L).
                                    :-permutation([0,1,2,3,4,5],L).
```

Programme ***Mu*** :

```
$theorem(Depth, [m, i]).
$theorem(Depth, []) :- fail.
$theorem(Depth, R) :-dif(Depth ,0),minus( Depth ,1,D),theorem(D, S),rules(S, R).

$rules(S, R) :- rule3(S,R).
$rules(S, R) :- rule4(S,R).
$rules(S, R) :- rule1(S,R).
$rules(S, R) :- rule2(S,R).

$rule1(S,R) :-append(X, [i], S),append(X, [i,u], R).

$rule2([m | T], [m | R]) :- append(T, T, R).

$rule3([], X) :- fail.
$rule3(R, T) :-append([i,i,i], S, R),append([u], S, T).
$rule3([H | T], [H | R]) :- rule3(T, R).

$rule4([], X) :- fail.
$rule4(R, T) :- append([u, u], T, R).
$rule4([H | T], [H | R]) :- rule4(T, R).

                                    :-theorem(5,T).
```

Programme ***ham*** :

```
$ham(X):-cycle_ham([c,d,e,f,g,k,l,m,n,o,p,q,r,t],X).
$chain_ham([X],L,[X|L]).
$chain_ham([X|Y],K,L):-delete(Z,Y,T),edge(X,Z),chain_ham([Z|T],[X|K],L).

$edge(X,Y):-connect(X,L),el1(Y,L).

$connect(c,[t,d,l]).
$connect(d,[c,e,q]).
$connect(e,[d,f,m]).
$connect(f,[e,g,r]).
```

```
$connect(g,[f,p,n]).
$connect(k,[o,l,t]).
$connect(l,[k,m,c]).
$connect(m,[l,n,e]).
$connect(n,[m,o,g]).
$connect(o,[n,k,r]).
$connect(p,[g,q,t]).
$connect(q,[p,r,d]).
$connect(r,[q,o,f]).
$connect(t,[p,k,c]).
                                        :-ham(X).
```

Programme ***8-reines*** :

```
$go8r(X1,X2,X3,X4,X5,X6,X7,X8):- sol8([c(1,X1),c(2,X2),c(3,X3),c(4,X4),c(5,X5)c(6,X6),
                                        c(7,X7),c(8,X8)]).

$ delete(X,[X|L],L).
$ delete(X,[Y|L],[Y|R]) :- delete(X,L,R).
; % genere toutes les listes de couples possibles
$ genere([],[],[]).
$ genere(Lig,Col,[c(X,Y)|L]) :-delete(X,Lig,Lig1),delete(Y,Col,Col1), genere(Lig1,Col1,L).

;% verifie qu'il n'y a pas d'attaque entre la nouvelle piece et celles placees
$ pasdattaque(X,[]).
$pasdattaque(c(X,Y),[c(X1,Y1)|Autres]) :- dif(X,X1),dif(Y,Y1),minus(X1,X,X2),ab-
s(X2,Rx),minus(Y1,Y,Y2), abs(Y2,Ry),dif(Rx,Ry),pasdattaque(c(X,Y),Autres).
```

Programme ***interrogation*** [Gia 85] adapté dans [Per 90] :

```
$analyse(Q,X) :- query(Q,[],E),transform(E,L),eval(L,X).
```

grammaire :

```
<query> ::= qui<verb_phrase>|<who_is_the><noun_1><noun_phrase>
<who_is_the>::=quel est le | quels sont les | quelle est la | quelles sont les
<verb_phrase>::=<verb><prediacte>
<prediacte>::=<infinite_article><noun_0> | <definite_article><noun_1><noun_phrase>
<noun_phrase>::=<preposition><goper_noun>|<prep_article><noun_0>|
<preposition><noun_1><noun_phrase>
<verb>::=est | sont
<prep_article>::=des | <prep_article><infinite_article>
<indefinite_article>::=un | une | des
<definite_article>::=le | la | les
<preposition>::=de | des
```

```
$query([qui|X],Y,qui(VP)) :- verb_phrase(X,Y,GENRE,NOMBRE,VP).


$query(X,Y,quel(nc(N1),NP)):-who_is_the(X,Z,GENRE,NOMBRE),noun_1(Z,T,-
GENRE,NOMBRE,N1),noun_phrase(T,Y,NP).

$verb_phrase(X,Y,GENRE,NOMBRE,gv(verb(V),P)) :- verb(X,Z,GENRE,NOM-
BRE,V),predicate(Z,Y,GENRE,NOMBRE,P).

$noun_phrase(X,Y,gn(P,PN)) :-preposition(X,Z,GENRE,NOMBRE,P),prop-
er_noun(Z,Y,GENRE,NOMBRE,PN).

$noun_phrase(X,Y,gn(PA,nc(N0))) :-prep_article(X,Z,GENRE,NOMBRE,PA),-
noun_0(Z,Y,GENRE,NOMBRE,N0).

$noun_phrase(X,Y,gn(PA,nc(N1),NP)) :-prep_article(X,Z,GENRE,NOMBRE,PA),-
noun_1(Z,T,GENRE,NOMBRE,N1),noun_phrase(T,Y,NP).

$predicate(X,Y,GENRE,NOMBRE,att(art(A),nc(N0))) :-indefinite_article(X,Z,GEN-
RE,NOMBRE,A),noun_0(Z,Y,GENRE,NOMBRE,N0).

$predicate(X,Y,GENRE,NOMBRE,att(art(A),nc(N1),NP)) :-definite_article(X,Z,GEN-
RE,NOMBRE,A),noun_1(Z,T,GENRE,NOMBRE,N1),noun_phrase(T,Y,NP).

$proper_noun(X,Y,masculin,singulier,np(MPN)) :-masculine_proper_noun(X,Y,MPN).
$proper_noun(X,Y,feminin,singulier,np(FPN)) :-feminine_proper_noun(X,Y,FPN).

$who_is_the([quel|[est|[le|X]]],X,masculin,singulier).
$who_is_the([quels|[sont|[les|X]]],X,masculin,pluriel).
$who_is_the([quelle|[est|[la|X]]],X,feminin,singulier).
$who_is_the([quelles|[sont|[les|X]]],X,feminin,pluriel).

$noun_0([homme|X],X,masculin,singulier,homme).
$noun_0([femme|X],X,feminin,singulier,femme).

$noun_1([fils|X],X,masculin,singulier,fils).
$noun_1([fils|X],X,masculin,pluriel,fils).
$noun_1([fille|X],X,feminin,singulier,fille).
$noun_1([filles|X],X,feminin,pluriel,filles).
$noun_1([frere|X],X,masculin,singulier,frere).
$noun_1([freres|X],X,masculin,pluriel,freres).
$noun_1([soeur|X],X,feminin,singulier,soeur).
$noun_1([soeurs|X],X,feminin,pluriel,soeurs).
$noun_1([oncle|X],X,masculin,singulier,oncle).
$noun_1([oncles|X],X,masculin,pluriel,oncles).
$noun_1([tante|X],X,feminin,singulier,tante).
```

```
$noun_1([tantes|X],X,feminin,pluriel,tantes).
$noun_1([cousin|X],X,masculin,singulier,cousin).
$noun_1([cousins|X],X,masculin,pluriel,cousins).

$masculine_proper_noun([X|L],L,X) :- male(X).
$feminine_proper_noun([X|L],L,X) :- female(X).

$prep_article([des|X],X,GENRE,pluriel,des).
$prep_article(X,Y,GENRE,NOMBRE,art(A)) :-preposition(X,Z,GENRE,NOMBRE,P),in-
definite_article(Z,Y,GENRE,NOMBRE,A).

$preposition([de|X],X,GENRE,singulier,de).

$indefinite_article([un|X],X,masculin,singulier,un).
$indefinite_article([une|X],X,feminin,singulier,une).
$indefinite_article([des|X],X,GENRE,pluriel,des).

$definite_article([le|X],X,masculin,singulier,le).
$definite_article([la|X],X,feminin,singulier,la).
$definite_article([les|X],X,GENRE,pluriel,les).

$transform(qui(VP),L) :- transform(VP,L).

$transform(quel(N,NP),L) :- transform(N,L1),transform(NP,L2),build_1(L1,L2,L).

$transform(ph(PN,VP),L) :- transform(PN,L1),transform(VP,L2),build_1(L1,L2,L).

$transform(gv(verb(V),P),L) :- transform(P,L).

$transform(gn(PA,PN),L) :- transform(PN,L).
$transform(gn(PA,N,NP),L) :- transform(N,L1),transform(NP,L2),build_1(L1,L2,L).

$transform(att(art(A),N),L) :- transform(N,L).
$transform(att(art(A),N,NP),L) :- transform(N,L1),transform(NP,L2),build_2(L1,L2,L).

$build_1(fils,L2,quel(X,fils(X,L2))).
$build_1(filles,L2,quel(X,filles(X,L2))).
$build_1(freres,L2,quel(X,freres(L2,X))).
$build_1(soeurs,L2,quel(X,soeurs(L2,X))).
$build_1(oncles,L2,quel(X,oncles(X,L2))).
$build_1(tantes,L2,quel(X,tantes(X,L2))).
$build_1(cousins,L2,quel(X,cousins(L2,X))).

$build_2(oncles,homme,quel(X,et(oncles(X,Y),homme(Y)))).
$build_2(oncles,femme,quel(X,et(oncles(X,Y),femme(Y)))).
$build_2(tantes,homme,quel(X,et(oncles(X,Y),homme(Y)))).
```

```
$build_2(tantes,femme,quel(X,et(oncles(X,Y),femme(Y)))).

$eval(quel(X,freres(Y,X)),X) :- brother(Y,X).
$eval(quel(X,soeurs(Y,X)),X) :- sister(Y,X).
$eval(quel(X,oncles(Y,X)),X) :- uncle(Y,X).
$eval(quel(X,tantes(Y,X)),X) :- aunt(Y,X).
$eval(quel(X,cousins(Y,X)),X) :- cousin(Y,X).

$eval(quel(X,fils(X,quel(Y,oncles(Z,Y)))),X) :- uncle(Y,Z),son(X,Y).
$eval(quel(X,filles(X,quel(Y,oncles(Z,Y)))),X) :- uncle(Y,Z),daughter(X,Y).
$eval(quel(X,fils(X,quel(Y,cousins(Z,Y)))),X) :- cousin(Y,Z),son(X,Y).
$eval(quel(X,filles(X,quel(Y,cousins(Z,Y)))),X) :- cousin(Y,Z),daughter(X,Y).
$eval(quel(X,et(oncles(X,Y),homme(Y))),X) :- uncle(X,Y),male(Y).
$eval(quel(X,et(oncles(X,Y),femme(Y))),X) :- uncle(X,Y),female(Y).
$eval(quel(X,et(tantes(X,Y),homme(Y))),X) :- aunt(X,Y),male(Y).
$eval(quel(X,et(tantes(X,Y),femme(Y))),X) :- aunt(X,Y),female(Y).

$grand_parent(X,Y) :- parent(X,Z),parent(Z,Y).
$grand_father(X,Y) :- grand_parent(X,Y),male(X).
$grand_mother(X,Y) :- grand_parent(X,Y),female(X).

$parent(X,Y) :- father(X,Y).
$parent(X,Y) :- mother(X,Y).

$child(X,Y) :- parent(Y,X).

$son(X,Y) :- child(X,Y), male(X).
$daughter(X,Y) :- child(X,Y),female(X).

$brother_or_sister(X,Y) :- father(P,X),father(P,Y),mother(M,X),mother(M,Y),dif(X,Y).
$brother(X,Y) :- brother_or_sister(X,Y),male(X).
$sister(X,Y) :- brother_or_sister(X,Y),female(X).

$uncle_or_aunt(X,Y) :- parent(Z,Y),brother_or_sister(X,Z).

$uncle(X,Y) :- uncle_or_aunt(X,Y),male(X).
$aunt(X,Y) :- uncle_or_aunt(X,Y),female(X).

$cousin(X,Y) :- uncle_or_aunt(Z,X),child(Y,Z).

$$transform(nc(N),N).
$transform(np(N),N).

$build_1(fils,L2,quel(X,fils(X,L2))).
$build_1(filles,L2,quel(X,filles(X,L2))).
$build_1(freres,L2,quel(X,freres(L2,X))).
```

```
$build_1(soeurs,L2,quel(X,soeurs(L2,X))).
$build_1(oncles,L2,quel(X,oncles(X,L2))).
$build_1(tantes,L2,quel(X,tantes(X,L2))).
$build_1(cousins,L2,quel(X,cousins(L2,X))).

$build_2(oncles,homme,quel(X,et(oncles(X,Y),homme(Y)))).
$build_2(oncles,femme,quel(X,et(oncles(X,Y),femme(Y)))).
$build_2(tantes,homme,quel(X,et(oncles(X,Y),homme(Y)))).
$build_2(tantes,femme,quel(X,et(oncles(X,Y),femme(Y)))).
$eval(quel(X,freres(Y,X)),X) :- brother(Y,X).
$eval(quel(X,soeurs(Y,X)),X) :- sister(Y,X).
$eval(quel(X,oncles(Y,X)),X) :- uncle(Y,X).
$eval(quel(X,tantes(Y,X)),X) :- aunt(Y,X).
$eval(quel(X,cousins(Y,X)),X) :- cousin(Y,X).

$eval(quel(X,fils(X,quel(Y,oncles(Z,Y)))),X) :- uncle(Y,Z),son(X,Y).
$eval(quel(X,filles(X,quel(Y,oncles(Z,Y)))),X) :- uncle(Y,Z),daughter(X,Y).
$eval(quel(X,fils(X,quel(Y,cousins(Z,Y)))),X) :- cousin(Y,Z),son(X,Y).
$eval(quel(X,filles(X,quel(Y,cousins(Z,Y)))),X) :- cousin(Y,Z),daughter(X,Y).

$eval(quel(X,et(oncles(X,Y),homme(Y))),X) :- uncle(X,Y),male(Y).
$eval(quel(X,et(oncles(X,Y),femme(Y))),X) :- uncle(X,Y),female(Y).
$eval(quel(X,et(tantes(X,Y),homme(Y))),X) :- aunt(X,Y),male(Y).
$eval(quel(X,et(tantes(X,Y),femme(Y))),X) :- aunt(X,Y),female(Y).

$grand_parent(X,Y) :- parent(X,Z),parent(Z,Y).
$grand_father(X,Y) :- grand_parent(X,Y),male(X).
$grand_mother(X,Y) :- grand_parent(X,Y),female(X).

$parent(X,Y) :- father(X,Y).
$parent(X,Y) :- mother(X,Y).

$child(X,Y) :- parent(Y,X).

$son(X,Y) :- child(X,Y), male(X).
$daughter(X,Y) :- child(X,Y),female(X).

$brother_or_sister(X,Y) :- father(P,X),father(P,Y),mother(M,X),mother(M,Y),dif(X,Y).

$brother(X,Y) :- brother_or_sister(X,Y),male(X).
$sister(X,Y) :- brother_or_sister(X,Y),female(X).

$uncle_or_aunt(X,Y) :- parent(Z,Y),brother_or_sister(X,Z).

$uncle(X,Y) :- uncle_or_aunt(X,Y),male(X).
$aunt(X,Y) :- uncle_or_aunt(X,Y),female(X).
```

```
$cousin(X,Y) :- uncle_or_aunt(Z,X),child(Y,Z).

$mother(marcelle,louis).
$mother(marcelle,joseph).
$mother(raymonde,stephen).
$mother(raymonde,yves).
$mother(raymonde,judith).
$mother(raymonde,huguette).
$mother(huguette,claudine).
$mother(claudine,laurent).
$mother(helene,joseph).
$mother(helene,jean_louis).
$mother(helene,m_therese).
$mother(helene,daniel).
$mother(helene,francoise).
$mother(helene,monique).
$mother(helene,bernadette).
$mother(helene,pierre).
$mother(helene,genevieve).
$mother(judith,michel).
$mother(valentine,alain).
$mother(noelie,bernard).
$mother(martine,olivier).
$mother(m_therese,lionel).
$mother(m_therese,sebastien).
$mother(m_therese,agnes).
$mother(martine,celine).
$mother(monique,francois).
$mother(dominique,elise).
$mother(dominique,elodie).
$mother(chantal,laurent).

$mother(madeleine,yvonne).
$mother(madeleine,jean).
$mother(madeleine,michel).
$mother(elodie,yvette).
$mother(elodie,odette).
$mother(elodie,gilbert).
$mother(yvette,christian).
$mother(yvette,m_francoise).
$mother(yvette,jean_marc).
$mother(odette,martine).
$mother(yvonne,brice).
$mother(yvonne,virginie).
$mother(yvonne,roger).
```

$mother(yvonne,dominique).
$mother(yvonne,veronique).
$mother(m_francoise,laure).
$mother(martine,nicolas).
$mother(martine,celine).
$mother(dany,aline).
$mother(dany,eric).
$mother(dominique,stephane).
$mother(dominique,vanessa).

$father(jules,louis).
$father(jules,joseph).
$father(jules,stephen).
$father(jules,yves).
$father(jules,judith).
$father(jules,huguette).
$father(louis,claudine).
$father(christian,laurent).
$father(yves,joseph).
$father(yves,jean_louis).
$father(yves,marie_therese).
$father(yves,daniel).
$father(yves,francoise).
$father(yves,monique).
$father(yves,bernadette).
$father(yves,pierre).
$father(yves,genevieve).
$father(louis,michel).
$father(stephen,alain).
$father(joseph,bernard).
$father(jean_louis,olivier).
$father(guy,lionel).
$father(guy,sebastien).
$father(guy,agnes).
$father(daniel,celine).
$father(daniel,francois).
$father(bernard,elise).
$father(bernard,elodie).
$father(alain,laurent).

$father(henri,yvonne).
$father(henri,jean).
$father(henri,michel).
$father(leopold,yvette).
$father(leopold,odette).
$father(leopold,gilbert).

```
$father(jean,christian).
$father(jean,m_francoise).
$father(jean,jean_marc).
$father(louis,martine).

$father(michel,brice).
$father(michel,virginie).
$father(jean,roger).
$father(jean,dominique).
$father(jean,veronique).
$father(francois,laure).
$father(philippe,nicolas).
$father(philippe,celine).
$father(roger,aline).
$father(roger,eric).
$father(antonio,stephane).
$father(antonio,vanessa).

$female(agnes).
$female(aline).
$female(bernadette).
$female(celine).
$female(chantal).
$female(claudine).
$female(dany).
$female(dominique).
$female(elise).
$female(elodie).
$female(francoise).
$female(genevieve).
$female(helene).
$female(huguette).
$female(jackie).
$female(judith).
$female(laure).
$female(madeleine).
$female(m_francoise).
$female(m_therese).
$female(martine).
$female(monique).
$female(marcelle).
$female(noelie).
$female(odette).
$female(raymonde).
$female(valentine).
$female(vanessa).
```

```
$female(veronique).
$female(virginie).
$female(yvette).
$female(yvonne).

$male(alain).
$male(antonio).
$male(bernard).
$male(brice).
$male(christian).
$male(daniel).
$male(eric).
$male(francois).
$male(gilbert).
$male(guy).
$male(henri).
$male(jean).
$male(jean_marc).
$male(jean_louis).
$male(joseph).
$male(jules).
$male(laurent).
$male(leopold).
$male(lionel).
$male(louis).
$male(michel).
$male(nicolas).
$male(olivier).
$male(pierre).
$male(philippe).
$male(roger).
$male(sebastien).
$male(stephane).
$male(stephen).
$male(thierry).
$male(yves).

;/* interrogation-q1*/
$q1(X):-analyse([quels|[sont|[les|[cousins|[de|[claudine]]]]]],X).

;/* interrogation-q2 */
$q2(X):-analyse([quels|[sont|[les|[fils|[des|[cousins|[de|[claudine]]]]]]]],X).
```

Programme ***fibonacci*** :

```
$fib(1,1).
$fib(2,2).
$fib(X,R):-lt(2,X),minus(X,1,X1),minus(X,2,X2),fib(X1,R1),fib(X2,R2),plus(R1,R2,R).
```

Programme ***nreverse*** :

```
$nreverse([],[]).
$nreverse([X|L],R):-nreverse(L,LL),append(LL,[X],R).
```

Programme ***qsort*** :

```
$ pa([],N,[],[]).
$ pa([X|L],Y,[X|L1],L2):-le(X,Y),!,pa(L,Y,L1,L2).
$ pa([X|L],Y,L1,[X|L2]):-pa(L,Y,L1,L2).

$ qsort([],R,R).
$ qsort([X|L],R,R0):- pa(L,X,L1,L2),qsort(L2,R1,R0),qsort(L1,R,[X|R1]).
```

# *Références bibliographiques*

[Ali87] K. Ali.
"A method for Implenting Cut in Parallel Execution of Prolog"
Proceedings of the 4th Symposium on Logic Programming.
San Francisco, August 1987.

[Ali88] K. Ali.
" OR-parallel Execution of Prolog on BC-machine"
Proceedings of the Fifth International Conference and Symposium on Logic Programming. Seattle, August, 1988, 1531-1545.

[Ali90] K. Ali, R. Karlsson.
"The Muse approach to OR-Parallel Prolog".
Int. Journal of Parallel Programming, Vol 19, no2, 1990, pp 129-162.

[Ali91] K. Ali, R. Karlsson.
"Performance of Muse on the BBN Butterfly TC2000"
Proceeding ICLP'91 Pre-conference Workshop on Parallel Execution of logic programs. Paris juin 1991.

[Bar88] U. Baron, J. Chassin De Kergommeaux, M. Hailperin, M. Ratcliffe, P. Pobert.
"The Parallel ECRC Prolog System PEPSys : an overview and evaluation results".
FGCS'88, Nov-Dec 88, pp 841-850

[Bea91] A. Beaumont "scheduling Strategies and Speculative Work"
Proceeding ICLP'91 Pre-conference Workshop on Parallel Execution of logic programs. Paris juin 1991.

[Boi88] P. Boizumault.
"PROLOG L'implantation"
Masson 88.

[Bou90] H. Bourzoufi, G. Goncalves, I. Hannequin, P. Lecouffe, B. Toursel.
"Depth First Or Parallelism in the LOG-ARCH Project".
Parallel and Distributed Computing, and Systems, October 1990, pp42-45.

[Bou91] H. Bourzoufi, G. Goncalves, P. Lecouffe, B. Toursel.
"an efficient bindin management in OR-parallel model"
Proceeding ICLP'91 Pre-conference Workshop on Parallel Execution of logic programs. Paris juin 1991.

[Bri90] J. Briat, M. Favre, C. Geyer.
"OPERA: OU Parallélisme et Régulation Adaptative en Prolog. SPLT 90.

[But86] R. Butler, E. L. Lusk, R. Olson, R.A Overbeek.
"ANLWAM, a parallel implementation of the Warren Abstrac t Machine".
International report, Argonne National Laboratory, 1986.

[But88] R. Butler, T. Disk, E. L. Lusk, R. Olson, R. Overbeek and R. Stevens.
"Scheduling OR parallelism: an Argonne Perspective"
Fifth International Logic Programming Conference and Fifth Symposium on
Logic Programming 1988.

[Cal88] A. Calderwood.
" Cut, Commit and side Effects in Or-Parallel Prolog"
Internal Report, Gigalips Project, 1988.

[Cal89] A. Calderwood and P. Szeredi.
"Scheduling OR Parallelism in Aurora - The Manchester Scheduler".
6th International Conference on Logic Programming, Portugal, pp 419-435, June 1989

[Cha73] Chin-Liang Chang and Richard Char-Tung Lee.
"Symbolic Logic and Mechanical Theorem Proving"
Academic Press, 1973.

[Cha89a] J. Chassin de Kergommeaux, P. Codognet, P. Robert, JC. Syre.
"Une programmation logique parallèle : Les langages gardés"
TSI, 1989.

[Cha89b] J. Chassin de Kergommeaux, P. Codognet, P. Robert, JC. Syre.
"Une programmation logique parallèle : Systèmes non déterministes"
TSI, 1989.

[Col 72] A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel
"Un système de communication homme-machine en français"
G.I.A, Université d'Aix-marseille II, 1972.

[Clo88] W.F. CLOCKSIN, H. ALSHAWI.
"A method for efficiently Executing Horn Clause Programs using Multiple Processors"
New Generation Computing, 5 (1988), pp 361-376.

[Con83] J. S. Conery.
"The AND/OR Process Model for Parallel Execution of Logic Programes"
Ph D Thesis, University of California, IRvine, 1983.

[Del86] J.P Delahaye.
"Outils logiques pour l'intelligence artificielle" Eyrolles, 1986.

[Del87] J.P. Delahaye.
"Sémantique logique et dénotationnelle des Iterpréteurs Prolog"
Theoretical Informatics and Apllications, 1987.

[Del88] J.P Delahaye.
"Cours de Prolog avec Turbo Prolog"
Eyrolles, 88.

[Deg84] D. Degroot.
"Restricted And-Parallelism"
Proc of FGCS'84, ICOT, November 1984, pp 471-478.

[Dur86] I. Durand.
"Un modèle d'interprétation Répartie pour une Architecture Multiprocesseurs Prolog"
Thèse de Doctorat, Université P. Sabatier Toulouse, Oct 1986.

[Dwo84] Cynthia Dwork, Paris C. Kanellakis, John C. Mitchell.
"On the sequential Nature of Unification".
Journal of logic programming (1) : pages 35-50 , 1984.

[Gia85] F. Giannesini, H. Kanoui, R. Pasero, M. Van Caneghem.
PROLOG - interEditions (1985).

[Han89] I. Hannequin, G. Goncalves, P.Lecouffe, B. Toursel
"A new Parallel Evaluation Scheme"
Euromincro 89, Proceedings of the 13th International Symposium on Microprocessing and Microprogramming, September 1989.

[Hau87] B. Hausman, A. Ciepielewski, S. Haridi .
"OR-Parallel Prolog made efficient on shared memory multiprocessor ".
4th symp on Logic Programming, San Francisco, Sept 87, pp 69-79

[Hau90] B. Hausman.
"Handling of speculative work in OR-Parallel PROLOG Evaluation Results"
Proceeding of NACLP 1990, pp 721-736

[Her 86] M.V. Hermengildo.
"An Abstract Machine for Restricted AND-parallel execution of logic programs"
In 3rd Int. Conf. on Logic Programming, pages 25-39. London, July, 1986.

[Kha 88] M. Kharoune.
" MIPAP Un modèle d'interprétation parallèle pour Prolog"
Thèse de doctorat, Université de Rennes 1, 1986.

[Lin88] Y. J. Lin, V. Kumar.
"Performance of AND-Parallel execution of Logic programs on a shared-memory multiprocessor".
FGCS'88, Nov-Dec 88, pp 851-860.

[Lus88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R Stevens, D. H. Warren, A. Calderwoodd, P. Szeridi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, B. Hausman
"The Aurora OR-Parallel Prolog System".
FGCS'88, Nov-Dec 88, pp 819-830.

[Mel82] C.S. Mellish.
" An alternative tou structure-sharing in the implémentation of Prolog interpreter"
[Workshop'80], pp21-32, 1980.

[Per90] C. Percebois.
"Définition et Evaluation d'un Modèle d'Exécution Répartie pour les Systèmes Logiques Non-déterministes"
Thèse de Doctorat d'Etat, soutenue le 11 décembre 1990 à l'Université Paul Sabatier de Toulouse

[Rob 88] P. Robert.
" Une machine Abstraite pour la mise en oeuvre du parallélisme OU/ET en Programmation Logique"
Thèse soutenue en juin 1988 à l'ENSAE.

[Rob65] J.A Robinson.
" A machine oriented logic based on the resolution principle"
Journal of the ACM 12, pp23-44, 1965.

[Soh85] Yukio Sohma, Ken Satoh, Koichi Kumon, Hideo Masuzawa, Akihiro Itashiki.
"A New Parallel Inference Mecanism Based on sequential Processing"
IFIP TC-10 Working Conf. On fifth Generation Computer Architecture.
Manchester, July 15-18, 1985.

[Sar87] V.A. SARASWAT "GHC: Operational Semantic, Problems, and Relationship with CP"
In 4th symp on Logic Programming, San Fransisco, Septembre 1987, pp 347-359.

[Ste86] L. Sterling, A. Ehud Shapiro.
" The Art of Prolog"
Advanced Programming Techniques, MIT press, 1986.

[Tic83] E. Tick.
"An overlapped Prolog processor"
Technical Note 308, SRI international, Menlo Park, 1983.

[War80] D. H. D. Warren.
"An improved Prolog implementation which optimizes tail recursion"
[Workshop'80], pp1-11, 1980.

[War83] D. H. D. Warren.
"An abstract Prolog instruction set"
Technical Note 309, SRI International, Menlo Park, 1983.

[War87a] D. H. D. Warren.
"OR-Parallel Execution Models of Prolog".
TAPSOFT 87, Pisa, March 87, pp 243-255.

[War87b] D. H. D. Warren "The SRI model for OR_Parallel execution of Prolog- Abstract design and implementation issues"
4th symp on Logic Programming, San Francisco, Sept 87, pp 92-102

## Résumé :

La réalisation technique des super-ordinateurs parallèles est maintenant possible à des coûts raisonnables, mais pour qu'ils soient exploitables encore faut-il qu'on sache les programmer.

Les travaux réalisés dans le cadre de cette thèse répondent à un double objectif, d'une part, ils montrent que le langage PROLOG peut être utilisé comme un langage de programmation des machines parallèles, en exploitant uniquement le parallélisme intrinsèque au langage, et d'autre part ils s'inscrivent parmi les efforts visant à répondre au besoin accru de puissance de calcul dans le domaine du traitement symbolique, en particulier dans les applications d'Intelligence Artificielle.

Pour cela, un modèle de calcul exploitant le *parallélisme-OU* offert par PROLOG, a été élaboré en tenant compte de toutes les propriétés non-logiques du langage, et plus particulièrement les effets de bord. La principale caractéristique du modèle est d'éviter toute recopie d'environnement de calcul entre processeurs. Un système de simulation basé sur une extension d'un interpréteur PROLOG, a permis d'évaluer les performances du modèle d'exécution. Le gain de performances étant linéaire dans les programmes offrant beaucoup de parallélisme. Les résultats obtenus ont montré également que l'efficacité du *parallélisme-OU* (multiséquentiel) dépend fortement de la stratégie d'allocation des travaux au processeurs : si dans les programmes sans effets de bord, de meilleures performances sont obtenues par l'utilisation de stratégies augmentant la granularité du parallélisme en faisant éviter aux processeurs des changements de tâches fréquents, dans les programmes introduisant des effets de bord, la stratégie favorisant les branches les plus à gauche dans l'arbre de recherche permet de réduire à la fois la quantité du travail spéculatif introduit par le prédicat CUT, ainsi que les déperditions dues aux suspensions des processeurs effectuant des effets de bord. Le modèle proposé, évitant toute recopie d'environnement permet de supporter ces différentes stratégies.



## Mots-clés :

Programmation logique, Parallélisme-OU, Modèle multiséquentiel, Effets de bord, Travail spéculatif.