LAIOS : un réseau multiprocesseur orienté vers des applications d'intelligence artificielle

Jean Duprat

THESE

présentée par

Jean DUPRAT,
professeur agrégé de mathématiques

pour obtenir le titre de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**
(arrêté ministériel du 5 juillet 1984)

(Spécialité : microélectronique)

==========

LAIOS : UN RESEAU MULTIPROCESSEUR ORIENTE VERS DES APPLICATIONS D'INTELLIGENCE ARTIFICIELLE.

==========

Date de soutenance : 22 juillet 1988

Composition du jury : M. J. DELLA DORA, président,
M. B. COURTOIS,
M. P. JORRAND,
M. B. LECUSSAN,
M. J. P. SANSONNET.

Thèse préparée au sein du laboratoire TIM3

"L'auteur, géomètre intrépide et théologien lumineux, résout, par le secours seul de la règle et du compas, toutes les questions théologiques, principalement celles qui concernent le libre arbitre et la grâce..."

D'Alembert.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

sident: Georges LESPINARD

Année 1988

Professeurs des Universités

RIBAUD	Michel	ENSERG
RRAUD	Alain	ENSIEG
UDELET	Bernard	ENSPG
AUFILS	Jean-Pierre	ENSEEG
IMAN	Samuel	ENSERG
OCH	Daniel	ENSPG
IS	Philippe	ENSHMG
NNETAIN	Lucien	ENSEEG
UVARD	Maurice	ENSHMG
ISSONNEAU	Pierre	ENSIEG
UNET	Yves	IUFA
ILLERIE	Denis	ENSHMG
VAIGNAC	Jean-François	ENSPG
ARTIER	Germain	ENSPG
ENEVIER	Pierre	ENSERG
ERADAME	Herve	UFR PGP
OVET	Alain	ENSERG
HEN	Joseph	ENSERG
UMES	André	ENSERG
RVE	Félix	ENSHMG
LLA-DORA	Jean-François	ENSIMAG
PORTES	Jacques	ENSPG
LMAZON	Jean-Marc	ENSERG
RAND	Francis	ENSEEG
RAND	Jean-Louis	ENSIEG
GGIA	Albert	ENSIEG
NLUPT	Jean	ENSIMAG
ULARD	Claude	ENSIEG
NDINI	Alessandro	UFR PGP
UBERT	Claude	ENSPG
NTIL	Pierre	ENSERG
EVEN	Hélène	IUFA
ERIN	Bernard	ENSERG
YOT	Pierre	ENSEEG
NES	Marcel	ENSIEG
USSAUD	Pierre	ENSIEG
JOUBERT	Jean-Claude	ENSPG
JOURDAIN	Geneviève	ENSIEG
LACOUME	Jean-Louis	ENSIEG
LESIEUR	Marcel	ENSHMG
LESPINARD	Georges	ENSHMG
LONGEQUEUE	Jean-Pierre	ENSPG
LOUCHET	François	ENSIEG
MASSE	Philippe	ENSIEG
MASSELOT	Christian	ENSIEG
MAZARE	Guy	ENSIMAG
MOREAU	René	ENSHMG
MORET	Roger	ENSIEG
MOSSIERE	Jacques	ENSIMAG
OBLED	Charles	ENSHMG
OZIL	Patrick	ENSEEG
PARIAUD	Jean-Charles	ENSEEG
PERRET	René	ENSIEG
PERRET	Robert	ENSIEG
PIAU	Jean-Michel	ENSHMG
POUPOT	Christian	ENSERG
RAMEAU	Jean-Jacques	ENSEEG
RENAUD	Maurice	UFR PGP
ROBERT	André	UFR PGP
ROBERT	Francois	ENSIMAG
SABONNADIERE	Jean-Claude	ENSIEG
SAUCIER	Gabrielle	ENSIMAG
SCHLENKER	Claire	ENSPG
SCHLENKER	Michel	ENSPG
SILVY	Jacques	UFR PGP
SIRYES	Pierre	ENSHMG
SOHM	Jean-Claude	ENSEEG
SOLER	Jean-Louis	ENSIMAG
SOUQUET	Jean-Louis	ENSEEG
TROMPETTE	Philippe	ENSHMG
VEILLON	Gérard	ENSIMAG
ZADWORNY	François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

LLIET	Louis

Personnes ayant obtenu le diplôme

d'HABILITATION A DIRIGER DES RECHERCHES

CKER	Monique
DER	Zdenek
ASSERY	Jean-Marc
OLLET	Jean-Pierre
EY	John
INET	Catherine
MMAULT	Christian
RNUEJOLS	Gérard
JLOMB	Jean-Louis
ARD	Francis
NES	Florin
DEROO	Daniel
DIARD	Jean-Paul
DION	Jean-Michel
DUGARD	Luc
DURAND	Madeleine
DURAND	Robert
GALERIE	Alain
GAUTHIER	Jean-Paul
GENTIL	Sylviane
GHIBAUDO	Gérard
HAMAR	Sylvaine
HAMAR	Roger
LADET	Pierre
LATOMBE	Claudine
LE GORREC	Bernard
MADAR	Roland
MULLER	Jean
NGUYEN TRONG	Bernadette
PASTUREL	Alain
PLA	Fernand
ROUGER	Jean
TCHUENTE	Maurice
VINCENT	Henri

CHERCHEURS DU C.N.R.S

Directeurs de recherche 1ère Classe

CARRE	René	LANDAU	Ioan
FRUCHART	Robert	VACHAUD	Georges
HOPFINGER	Emile	VERJUS	Jean-Pierre
JORRAND	Philippe		

Directeurs de recherche 2ème Classe

ALEMANY	Antoine	KLEITZ	Michel
ALLIBERT	Colette	KOFMAN	Walter
ALLIBERT	Michel	KAMARINOS	Georges
ANSARA	Ibrahim	LEJEUNE	Gérard
ARMAND	Michel	LE PROVOST	Christian
BERNARD	Claude	MADAR	Roland
BINDER	Gilbert	MERMET	Jean
BONNET	Roland	MICHEL	Jean-Marie
BORNARD	Guy	MUNIER	Jacques
CAILLET	Marcel	PIAU	Monique
CALMET	Jacques	SENATEUR	Jean-Pierre
COURTOIS	Bernard	SIFAKIS	Joseph
DAVID	René	SIMON	Jean-Paul
DRIOLE	Jean	SUERY	Michel
ESCUDIER	Pierre	TEODOSIU	Christian
EUSTATHOPOULOS	Nicolas	VAUCLIN	Michel
GUELIN	Pierre	WACK	Bernard
JOUD	Jean-Charles		

Personnalités agréées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

ENSEEG

CHATILLON	Christian	SARRAZIN	Pierre
HAMMOU	Abdelkader	SIMON	Jean-Paul
MARTIN GARIN	Régina		

ENSERG

BOREL	Joseph

ENSIEG

DESCHIZEAUX	Pierre	PERARD	Jacques
GLANGEAUD	François	REINISCH	Raymond

ENSHMG

ROWE	Alain

ENSIMAG

COURTIN	Jacques

EFP

CHARUEL	Robert

C.E.N.G

CADET	Jean	NIFENECKER	Hervé
COEURE	Philippe	PERROUD	Paul
DELHAYE	Jean-Marc	PEUZIN	Jean-Claude
DUPUY	Michel	TAIEB	Maurice
JOUVE	Hubert	VINCENDON	Marc
NICOLAU	Yvan		

Laboratoires extérieurs :

C.N.E.T

DEVINE	Rodericq	MERCKEL	Gérard
GERBER	Roland	PAULEAU	Yves

****

****

***

REMERCIEMENTS.

Ce travail n'aurait pas vu le jour sans:

M. JEAN DELLA DORA, qui me fait l'honneur de présider mon jury, dont l'affabilité et la simplicité réchauffe le cœur,

M. PHILIPPE JORRAND, qui a accepté d'être rapporteur, et son enthousiasme communicatif pour l'intelligence artificielle,

M. BERNARD LECUSSAN, qui a accepté d'être membre du jury, dont les précieux conseils ont contribué à l'amélioration de ce texte,

M. JEAN-PAUL SANSONNET, qui a accepté d'être rapporteur et qui a su, à plusieurs reprises, m'encourager dans mon travail,

M. BERNARD COURTOIS, mon directeur, dont l'attention toujours bienveillante mais jamais pesante à mes travaux fut une motivation de tous les instants,

M. MICHEL COSNARD, qui m'a fait l'honneur de m'inviter dans la sympathique équipe enseignant l'informatique à l'Ecole Normale Supérieure de Lyon dont il a la charge, et dont la gentillesse y est devenu proverbiale,

M. GERARD NOGUEZ et tous les Enseignants d'Informatique de PARIS VI, qui ont guidé mes premiers pas et m'ont inoculé le virus, avec la qualité de leur enseignement,

M. FRANCIS JUTAND et son équipe de l'E.N.S.T., M. JEAN-JACQUES GIRARDOT de l'E.M.S.E., et leur vision enthousiasmante de l'informatique et de son enseignement,

M. JEAN-MICHEL MULLER, qui est bien plus que mon correcteur orthographique préféré,

Mme DOMINIQUE MERMET PASTRE, binôme des premières années, dont la rigueur et l'exigeance du travail bien fait est un exemple,

Toute l'équipe du laboratoire d'architecture de TIM3, où il fait si bon travailler, où je compte tant d'amis, avec une mention particulière pour les secrétaires et "tuquettes" qui sont la providence du chercheur perdu dans les arcanes de l'administration,

Toute l'équipe d'algorithmique parallèle de TIM3, le climat qui règne dans leur laboratoire est un mélange particulièrement réussi de travail et de joie de vivre,

Mes parents, mes sœurs, ma famille, la compréhension et l'intérêt qu'ils ont apportés à mon entreprise de "recyclage",

Mes amis, mes collègues enseignants, leur réconfort dans les moments difficiles,

Mes anciens élèves enfin, qui ont maintenu mon goût d'apprendre et à qui j'offre ce travail à titre d'illustration d'un propos que je leur ai souvent tenu: "la vie n'est pas jouée à vingt ans",

Que tous trouvent ici mes plus sincères remerciements.

A la mémoire de Fernando Pereira,

photographe,

LAIOS : UN RESEAU MULTIPROCESSEUR ORIENTE VERS DES APPLICATIONS D'INTELLIGENCE ARTIFICIELLE.

RESUME

Ce travail présente une architecture massivement multiprocesseur pour l'exécution parallèle d'applications d'intelligence artificielle appelée LAIOS. Il se divise en quatre parties.

La première présente le langage de programmation. LAIOS se programme avec un dialecte de PROLOG inspiré de LISLOG-C. Le format interne utilise des objets autodéfinis structurés en blocs.

La seconde partie définit le modèle. L'arbre de recherche ET/OU de PROLOG est déroulé dynamiquement. Le ET pipe line et le OU parallélisme sont assurés par recopie de données et synchronisés par un mécanisme d'anticipation. Le traitement des données est effectué par un opérateur universel: l'unificateur/évaluateur.

La troisième partie décrit l'architecture. Les programmes sont compilés et rangés dans une base de connaissances, les modules opératoires accèdent à cette base par un réseau de bus et de mémoires cache. Les modules sont placés sur un maillage hexagonal et communiquent entre voisins par des mémoires à double accès.

La dernière partie présente les résultats de simulation. Un algorithme qui prend en compte les contraintes de communications entre modules, offre une répartition correcte de la charge sur le réseau. L'opérateur universel est validé et le ramasse miettes démontre son efficacité. La simulation du réseau démontre, qu'en l'absence d'un parallélisme suffisant, la recopie des données handicape les performances. Des solutions sont envisagées afin de s'affranchir de ce problème.

MOTS CLES:

Intelligence artificielle.
PROLOG.
Architecture modulaire.
Multiprocesseur massif.
Parallélisme.
Exécution dynamique.

LAIOS : A MULTIPROCESSOR NETWORK FOR ARTIFICIAL INTELLIGENCE APPLICATIONS.

ABSTRACT

This work presents a massively multiprocessor network, named LAIOS, for a parallel execution of artificial intelligence applications. It is divided in four parts.

The first one shows the programmation language. LAIOS'language is a PROLOG dialect near of LISLOG-C. The internal format uses self-defined objects structured in bloks.

The second part defines the model. The research AND/OR tree of PROLOG is dynamically spread. The AND pipe-line and the OR parallelism are ensured by the data copy and synchronized by the anticipation mechanism. Data are computed by an universal operator: the unifier/evaluator.

The third part describes the architecture. The programs are compiled and memorized in a knowledge base, the operating modules have access to it by buses and cache memories. The modules are set on an hexagonal lattice and communicate between neighbours by dual ports memories.

The fourth part gives the simulations results. An algorithm taking communication constraints into account, offers a correct load balancing onto the network. The universal operator is valided and the garbage collection is efficient. The network simulation demonstrates that, without a sufficient parallelism, the multiple data copies handicap the performances. Propositions are considered to overcome this problem.

KEY WORDS:

Artificial intelligence.
PROLOG.
Modular architecture.
Massively multiprocessor.
Parallelism.
Dynamical execution.

PROLOGUE

L'intelligence artificielle est un vaste domaine de la science informatique [RECH_85]. Elle connaît un essor rapide ces dernières années. Les travaux du Projet Japonais de Cinquième Génération [ICOT_82], [KAWA_84], [MOTO_85], [FUC&_87], y apportent une contribution importante; les autres nations industrialisées fournissent également un effort sérieux; citons "the Strategic Computing Initiative" aux U.S.A. ou une partie du programme ESPRIT de la Communauté Européenne [TREL_85]. Actuellement, certaines des applications envisagées (vision artificielle, reconnaissance des formes, reconnaissance de la parole, calcul formel, démonstrations de théorèmes ...) sont encore, pour l'essentiel, objet de recherches, d'autres, et tout particulièrement les systèmes experts, ont trouvé leur créneau sur le marché des progiciels dans des domaines aussi variés que la finance, la médecine ou la conception de circuits intégrés V.L.S.I. [NAU_83], [SCHI_84], [WILL_84].

D'ores et déjà, les travaux effectués permettent de cerner les besoins matériels nécessaires à l'exécution performante des applications de l'intelligence artificielle [HAYE_85], [TRE&_86], [WAH&_86], [CATI_87]. Les notions de calcul formel, de bases de connaissances et de moteurs d'inférence ont des exigences différentes de celles de calculs numériques ou de gestion de fichiers. La relative inadaptation des machines classiques à ces nouvelles exigences conduise à des performances modestes en regard de l'importance des travaux demandés.

De nouvelles architectures sont à l'étude pour améliorer cet état de fait. La présence d'un fort potentiel de parallélisme dans les applications envisagées permet d'espérer une accélération du temps d'exécution sur des machines à parallélisme massif . Ces architectures, mettant en jeu un grand nombre de processeurs et de mémoires de grande capacité, ne sont devenues réalisables que grâce aux extraordinaires progrès effectués dans le domaine de l'intégration sur le silicium de fonctions complexes. Aujourd'hui, le développement et la mise au point de ces machines est dans une phase particulièrement attractive; plusieurs solutions sont envisagées face aux problèmes de mise en oeuvre sans que l'on sache à priori lesquelles s'imposeront.

Le travail présenté ici est une contribution à ces recherches. LAIOS (Lattice for Artificial Intelligence Oriented System) est un réseau hexagonal modulaire extensible de processeurs sur lequel se déroulent des arbres de recherche en parallèle. Le parallélisme implicite contenu dans les langages qui admettent un modèle d'exécution arborescent y est exploitable. Les langages de programmation en logique (PROLOG et certains de ses dialectes) s'y exécutent quasi directement. L'extension aux langages fonctionnels y est étudiée.

Des notions nouvelles, fort éloignées du schéma de Von Neumann, sont mises en oeuvre dans LAIOS. Les données autodéfinies permettent de confiner l'usage du jeu d'instruction du processeur utilisé à la seule écriture du système d'exploitation du module; du fait des possibilités sans cesse croissantes d'intégration V.L.S.I., il est raisonnable d'envisager le décodage et le traitement des données autodéfinies entièrement cablés. Par sa proximité avec les langages de haut niveau, la structuration de la base de connaissances en noeud-modèles offre un environnement de travail agréable et efficace. Enfin, le contrôle est assuré par des automates programmables, les classes de noeuds, ce qui permet une implantation évolutive des langages.

La présentation de ce travail suit le principe d'une conception descendante. La première partie

présente les langages de haut et de bas niveaux, la seconde le modèle d'exécution et le parallélisme mis en oeuvre, la troisième l'architecture du réseau et de chaque module et la dernière les simulations effectuées et les performances espérées.

Le programme de simulation détaille tous les transferts d'information et leur méthode d'accès. Dans l'optique d'une réalisation à base de circuits commerciaux, cela facilitera l'écriture du logiciel système. Ce programme est structuré en couches, ce qui, dans l'hypothèse de la réalisation de circuits spécialisés, propose une séparation entre les fonctionnalités matérielles et les fonctionnalités logicielles.

Ce travail a été réalisé au sein de l'équipe d'architecture des ordinateurs du laboratoire TIM3 de l'INPG/IMAG de Grenoble.

LIVRE PREMIER :

LE LANGAGE.

CHAPITRE I. LES LANGAGES DE HAUT NIVEAU.

Le langage informatique est l'interface entre deux mondes, l'homme et la machine. S'il est présomptueux de définir à la fois rigoureusement et brièvement le terme de langage de haut niveau, le sens commun l'accepte comme un langage indépendant de la machine sur laquelle il est implanté et non spécialisé sur une application.

Ce chapitre passe en revue les grandes familles de langages et examine pour chacune d'elles les facilités offertes pour des applications de type intelligence artificielle et pour l'exploitation du parallélisme.

I. 1. TOUR D'HORIZON DU PROBLEME.

Trois questions se posent: quel langage? pour quelles applications? avec quel mode de fonctionnement?

I. 1. 1. Classification des langages.

Une classification couramment employée [SHAO_85], [TREL_86], distingue quatre classes de langages:

- les langages **impératifs**,
- les langages **fonctionnels**,
- les langages de **programmation en logique**,
- les langages **orientés objets**.

La machine virtuelle d'un langage impératif est d'architecture **VonNeumann**. Ce sont les langages les plus utilisés aujourd'hui; il en existe de nombreuses familles.

Un langage fonctionnel est basé sur la théorie mathématique des fonctions. Il contient sa propre machine universelle, la fonction d'évaluation. **LISP** [GIRA_85] est l'exemple le plus connu de langage fonctionnel, mais d'autres langages plus formels ont été définis comme FP par Backus [MAGO_ 85].

Un langage de programmation en logique est basé sur la logique des prédicats. Il utilise comme moteur le raisonnement par inférence. **PROLOG** [KOWA_74], [WARR_78], [WAR&_84] , [GIA&_85] et ses dialectes constitue la famille de cette classe de langages.

Enfin, un langage de programmation objet a pour souci premier la cohérence. L'ensemble des objets est structuré hiérarchiquement en classes, les objets communiquant entre eux par messages. Un exemple type de ces langages est **SMALLTALK** [MEV&_87].

I. 1. 2. Les besoins de l'intelligence artificielle.

Un recencement des besoins informatiques spécifiques à l'intelligence artificielle a été fait par

Wah [WAH_87]. Une autre vision de ces problèmes est exposée dans le programme de recherche concertée en Intelligence Artificielle [CHO&_ 85].

Dans les applications de l'intelligence artificielle, le **traitement symbolique de données formelles** est souvent plus important que le calcul effectif de données numériques. Aux opérations arithmétiques et logiques classiques, il faut ajouter des opérations symboliques primitives telles que la comparaison, la sélection, le tri, la reconnaissance ou la fermeture transitive.

Un manque de savoir faire ou des connaissances incomplètes sur le problème traité rendent impossible une planification préalable des procédures utiles. De ce fait, les algorithmes d'intelligence artificielle sont souvent **non déterministes**. Ils conduisent alors à une énumération plus ou moins exhaustive de toutes les solutions possibles.

La grande diversité des exécutions possibles d'un algorithme conduit naturellement à donner la préférence à des **structures dynamiques**; celles-ci seront créées et détruites au fur et à mesure des besoins durant l'exécution. Il en résulte l'impossibilité de gérer de façon préalable la totalité de l'espace mémoire.

De plus, ces algorithmes reposent sur la connaissance du problème. Une machine intelligente se doit d'être capable d'**apprentissage** et de faire évoluer ses connaissances. La machine contient un moteur capable de raisonner sur les connaissances; ces connaissances forment un système ouvert.

La gestion de connaissances est confiée à une base disposant de fonctionnalités classiques d'ajout, de retrait ou de modifications. De plus, cette base doit offrir des possibilités de vérifications de cohérence entre informations ou de degré de confiance à accorder aux diverses informations. C'est le domaine difficile à maitriser des **ensembles flous** dans lequel la relation d'appartenance n'est plus booléenne.

I. 1. 3. Le parallélisme.

Le principe élémentaire consiste à découper un programme en **tâches** exécutables simultanément qui sont confiées à des processeurs différents. Lorsque ce découpage est défini par le programmeur à l'aide de primitives de contrôle, le parallélisme est dit **explicite**. Lorsqu'il est défini à la compilation par exemple par extraction du **graphe d'ordonnancement**, le parallélisme est **implicite**.

La finesse du découpage est généralement liée à la granularité du réseau de processeurs. De ce fait, on est souvent amené à effectuer un découpage plus grossier que celui du graphe d'ordonnancement pour obtenir des tâches plus conséquentes. Des dépendances entre tâches apparaissent qui nécessitent une synchronisation pour le **partage de ressources communes**.

Lorsqu'il s'agit de partager des données entre tâches concurrentes, il est fondamental de savoir quelles tâches seront susceptibles de modifier ces données afin de maintenir la cohérence du programme. Des **synchronisations de type producteur/consommateur** sont alors employées.

Enfin, il ne faut pas perdre de vue que le nombre de processeurs physiques est limité. En utilisant N processeurs, le meilleur cas ne peut être qu'une division par N du temps de calcul. Un problème de complexité exponentielle le restera même si l'algorithme parallèle utilisé a une complexité polynomiale sur un nombre illimité de processeurs. Il faut donc concevoir le parallélisme comme le moyen d'adapter le temps de calcul aux conditions d'utilisation. Par exemple, le gain d'un facteur cent sur un programme de vision artificielle et de reconnaissance

des formes pourra rendre intéressante la vitesse de déplacement d'un robot autonome.

I. 2. ELEMENTS DE CHOIX.

Bien qu'aucune réponse ne soit satisfaisante sur tous les points, les différentes classes de langages évoquées ci-dessus répondent de façon inégale aux exigences du cahier des charges.

I. 2. 1. Les langages impératifs.

Ces langages ont été définis pour réaliser des calculs effectifs, ils sont mal adaptés au calcul formel. Par exemple, les seules variables formelles admises sont généralement les paramètres des procédures.

Les premiers langages ayant été conçus à une époque où la mémoire était rare et chère, cette dernière était gérée avec application à la compilation. De ce fait, son utilisation était entièrement contrôlée par le programmeur lors de ses déclarations de types. Il en résultait une rigidité du cadre dans lequel se déroulait le programme. Cet état de fait s'est modifié dans les langages apparus plus récemment et PASCAL par exemple intègre des possibilités de création (new) et de destruction (dispose) d'objets dynamiques dans une zone mémoire généralement nommée tas.

Le parallélisme est apparu très tôt en FORTRAN mais uniquement sous un aspect de calcul vectoriel dans des boucles DO parallélisées sur de gros ordinateurs scientifiques. Cette classe de parallélisme **SIMD** (Single Instruction Multiple Data) est inadaptée aux problème d'intelligence artificielle. Il faut attendre les derniers langages impératifs apparus pour trouver un parallélisme plus souple appelé **MIMD** (Multiple Instruction Multiple Data) avec ADA, langage dans lequel les processus communiquent par une mémoire partagée, ou OCCAM dans lequel les processus échangent des messages.

Enfin le découpage strict entre données et programmes cadre mal avec la notion de base de connaissances dans laquelle les notions de savoir et de savoir faire sont unifiées.

Ainsi se trouvent rapidement résumées quelques raisons essentielles qui font que les langages impératifs sont peu employés en intelligence artificielle.

I. 2. 2. Les langages fonctionnels.

Ce sont généralement des langages **déclaratifs**, c'est-à-dire dans lesquels l'ordre d'écriture des fonctions est indifférent pour l'exécution. Il s'agit là d'un aspect important à la fois pour l'intelligence artificielle en apportant de la souplesse dans la gestion de la base de connaissances et pour l'exécution parallèle en associant une tâche à chaque appel de fonction.

Les objets manipulés par ces langages sont généralement des listes, ce sont des objets génériques pour des structures plus générales qui pourront être facilement construites. De plus, il y a unité de structure entre les objets et un seul opérateur d'exécution, la fonction récursive d'évaluation qui calcule la valeur de chaque objet [SAIN_86]. Cette vision unificatrice, proche de la notion de machine universelle, facilite grandement l'écriture des programmes de calcul formel. L'unicité des objets offre également de l'intérêt pour l'écriture des connaissances.

Ces langages ont un déroulement dynamique qui fait un large appel à la récursion. Les créations d'objets sont nombreuses en cours d'exécution. LISP est ainsi l'un des premiers langages à avoir posé concrètement le problème de la récupération de la place afin de traiter des problèmes importants sans saturation mémoire.

Les langages fonctionnels et tout particulièrement LISP sont très utilisés par la communauté américaine des chercheurs en intelligence artificielle.

Toutefois, ces langages ne comportent aucun mécanisme de raisonnement, de recherche, de déduction ou de choix de stratégie. De tels mécanismes généraux doivent être intégrés à un niveau supérieur dans la hiérarchie des langages. Ainsi apparaissent des générateurs de systèmes experts écrits en LISP, contenant un moteur d'inférence et des mécanismes de contrôle, qui permettent l'écriture rapide de systèmes experts dans des domaines divers.

I. 2. 3. Les langages de programmation en logique.

L'origine de PROLOG étant l'étude des **grammaires formelles**, PROLOG et la famille de ses dérivés est naturellement apte au calcul formel. Sa faiblesse résiderait plutôt dans le calcul effectif dont les opérations s'intègrent avec plus ou moins de bonheur selon les versions dans la syntaxe.

De même que les langages fonctionnels, les langages de programmation en logique sont par nature des **langages déclaratifs** avec les avantages inhérents: ergonomie pour l'utilisateur, souplesse de gestion des connaissances, découpage naturel en tâches parallélisables. Toutefois, PROLOG utilise l'ordre des déclarations des clauses pour le contrôle du déroulement, cette restriction contenant à la fois des avantages et des inconvénients. Ce point sera discuté lors de l'établissement du modèle d'exécution.

Ces langages font aussi un large appel à la récursion et à la création dynamique d'objets avec les avantages que celà contient en clarté et en facilité de programmation et les sophistications que celà nécessite pour le système qui le supporte en gestion d'appels et gestion mémoire.

Le déroulement d'un programme PROLOG réalise, sauf demande contraire explicite, la recherche exhaustive de toutes les solutions. Cette démarche se retrouve très souvent dans les applications d'intelligence artificielle. Toutefois, le contrôle de cette recherche est limité à une seule stratégie et un seul opérateur de **coupure** (pruning), ce qui sera souvent insuffisant et les applications importantes devront programmer leurs propres stratégies.

Cette recherche exhaustive se modélise en un arbre de recherche, ce qui amène naturellement à désirer l'exploration des branches en parallèle. Le succès de cet objectif sera un pas décisif dans l'implantation de langages de programmation en logique, en effet, la situation actuelle souffre d'un déroulement linéaire ralenti par la gestion lourde des **retours en arrière** (backtracking).

Cette analyse sommaire justifie le point de vue des chercheurs de l'I.C.O.T. [FUC&_87] pour lesquels les langages de programmation en logique sont un pont naturel entre l'intelligence artificielle et l'exécution parallèle.

I. 2. 4. Les langages orientés objets.

Ces langages demandent un mode de pensée différent et une vision originale des problèmes.

Le modèle d'un ensemble d'objets communiquant par échange de **messages** autorise la prise en compte simultanée de plusieurs messages différents par des objets différents. Le problème du maintien de la cohérence exige alors une synchronisation extraite du programme. Il faut penser le parallélisme en termes de réseau d'acteurs et non en termes de tâches concurrentes.

Le mécanisme d'**héritage** est susceptible de remplacer celui d'inférence dans les

raisonnements. Schématiquement, l'implication logique "l'autruche est un oiseau donc l'autruche a deux ailes" est remplacée par "l'autruche étant une instance de la classe des oiseaux, l'autruche hérite de la propriété générique de la classe des oiseaux qui est d'avoir deux ailes". Là encore une vision différente de l'enregistrement des connaissances et des mécanismes de raisonnement s'impose.

Enfin, les objets sont créés dynamiquement, mais dans le cadre strict de la hiérarchie, chaque objet étant typé. Augmenter les connaissances dans la résolution d'un problème ne s'exprimera donc pas dans la construction d'une structure mais dans la descente de la hiérachie du général au particulier.

Du fait de son originalité et de sa cohérence, cette classe de langages influence de nombreux travaux tant en intelligence artificielle qu'en exécution parallèle.

I. 3. TENDANCES ACTUELLES D'EVOLUTION.

Les trois familles, langages fonctionnels, langages de programmation en logique et langages orientés objets contiennent chacunes des notions intéressantes. La tendance actuelle consiste souvent à chercher une structure unificatrice qui les intègrerait.

I. 3. 1. Langages fonctionnels et de programmation en logique.

Ces deux familles ont en commun l'aspect déclaratif du programme, la gestion dynamique de la mémoire, l'usage généralisé de la récursion et la structuration des données en listes. Ils offrent donc des facilités d'interpénétration qui se sont effectuées de plusieurs façons différentes.

Certains interpréteurs PROLOG ont été écrits en LISP comme dans MAIA [SANS_85]. De ce fait, il est possible de passer en cours d'exécution d'un langage à l'autre au sein du même programme.

L'intégration de LISP dans PROLOG est davantage recherchée dans LISLOG [BOU&_87]; les fonctions évaluables de LISP sont des objets du langage et apparaissent en tant que telles comme paramètres des prédicats. Deux mécanismes d'évaluation peuvent être mis en jeu, soit une évaluation lors de l'appel, soit une évaluation par nécessité retardée jusqu'à l'utilisation.

Une approche ensembliste de cette intégration est formalisée par Cras [CRAS_86] qui intègre le typage des objets .

I. 3. 2. L'influence de la programmation orientée objet.

Les intégrations sont multiples car le sujet est riche de possibilités; la programmation orientée objet apporte la structuration et PROLOG les mécanismes de déduction. Cette présentation ne prétend pas à l'exhaustivité.

L'utilisation des fonctions et des clauses comme méthodes pour les objets est proposée dans PHOCUS [CHA&_87]. La structuration de l'ensemble des clauses en objet est réalisée dans MODULOG [DEW&_87]. Une définition d'objet a été réalisée en CONCURRENT PROLOG [SHA&_83] ou pour l'écriture d'un système expert en PROLOG [ARN&_87].

Le groupe BULL a développé sur sa gamme d'ordinateurs un noyau appelé KOOL [VIDE_87]. KOOL (Knowledge representation Object Oriented Language) est présenté comme un "shell" associant modèles objets, "frames", règles de production et programmation

fonctionnelle.

L'évolution la plus probable reste un langage orienté objet qui forme le cadre englobant de parties déclaratives soit sous forme de fonctions soit sous forme de clauses.

CHAPITRE II.
LE PROLOG DE LAIOS.

Cette version de PROLOG reprend intégralement la structure des clauses de Horn formées de prédicats logiques. Il y intègre les fonctions évaluables et les contraintes. Il est donc très voisin de LISLOG-C [BOU&_87] bien que plus restrictif.

II. 1. LA SYNTAXE.

Ce chapître présente les règles de grammaire de la syntaxe sous leur forme normale de Backus Naur (BNF). La compréhension de ce chapître, nécessairement technique, peut être facilitée par une lecture parallèle du chapître suivant qui présente la sémantique du langage; pour faciliter une telle lecture, les titres de paragraphes sont identiques dans ces deux chapîtres.

II. 1. 1. Les clauses.

Elles contiennent une tête et un corps. De façon classique, on appelle **fait** une clause dont le corps est vide, **but** une clause dont la tête est vide et **règle** une clause dont ni la tête ni le corps ne sont vides.

<clause> ::= <fait> | <but> | <règle>
<fait> ::= <tête> .
<but> ::= :- <corps> .
<règle> ::= <tête> :- <corps> .

Une tête ne peut contenir qu'un seul **prédicat** alors que le corps est une liste finie de prédicats.

<tête> ::= <prédicat>
<corps> ::= <liste de prédicats>
<liste de prédicats> ::= <prédicat> | <prédicat> , <liste de prédicats>

Les espaces entourant les signes de ponctuation sont ignorés. Des commentaires peuvent être insérés entre les clauses.

<signes de ponctuation> ::= :- | . | , | (|) | : | "
<commentaires> ::= " <suite de caractères> "

II. 1. 2. Les prédicats.

Ils sont formés d'un identificateur formel et d'une liste de paramètres éventuellement vide. La coupure (/) est un prédicat particulier.

<prédicat> ::= <identificateur formel> | <identificateur formel> (<liste de paramètres>) | /
<liste de paramètres> ::= <paramètre> | <paramètre> , <liste de paramètres>

Un identificateur formel est une chaîne de caractères alphanumériques dont le premier est une lettre minuscule; par contre, un identificateur évaluable est une chaîne de caractères alphanumériques dont le premier est une majuscule.

<minuscule> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<minuscule> ::= à | â | ä | é | è | ê | ë | î | ï | ô | ö | ù | û | ü
<majuscule> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
<majuscule> ::= O | P | Q | R | S | T | U | V | W | X | Y | Z
<chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<souligné> ::= _
<alphanumérique> ::= <minuscule> | <majuscule> | <chiffre> | <souligné>
<chaîne> ::= <> | <alphanumérique> <chaîne>
<identificateur formel> ::= <minuscule> <chaîne>
<identificateur évaluable> ::= <majuscule> <chaîne>

Un paramètre est soit une constante, soit une structure formelle, soit une structure évaluable.
<paramètre> ::= <constante> | <structure formelle> | <structure évaluable>

II. 1. 3. Les constantes.

Les constantes sont de quatre types classiques.

<constante> ::= <booléen> | <entier> | <réel> | <chaîne de caractères>
<booléen> ::= true | false
<entier> ::= <suite de chiffres> | <signe> <suite de chiffres>
<signe> ::= + | -
<suite de chiffres> ::= <chiffre> | <chiffre> <suite de chiffres>
<réel> ::= <entier> . <suite de chiffres> | <entier> . <suite de chiffres> <exposant>
<exposant> ::= e <entier> | E <entier>
<quote> ::= '
<caractère> ::= *tout caractère ayant un code ASCII sauf la quote et le guillemet*
<suite de caractères> ::= <> | <caractère> <suite de caractères>
<chaîne de caractères> ::= <quote> <suite de caractère> <quote>
<chaîne de caractères> ::= <quote> <suite de caractères> <quote> <chaîne de caractères>

Classiquement, le parenthésage des chaînes de caractères étant effectué par des quotes, une quote présente dans la chaîne devra être redoublée pour être reconnue; il en va de même pour le parenthésage des commentaires par des guillemets.

II. 1. 4. Les variables.

Une variable est définie par son identificateur évaluable. Elle est suivie d'une liste de contraintes éventuellement vide.

<variable> ::= <identificateur évaluable> <liste de contraintes>
<liste de contraintes> ::= <> | : <contrainte> <liste de contraintes>

Les contraintes peuvent être de trois types.

<contrainte> ::= <contrainte de liaison> | <contrainte d'appartenance>
<contrainte> ::= <contrainte d'inégalité>
<contrainte de liaison> ::= ? | !
<contrainte d'appartenance> ::= in <ensemble> | out <ensemble>
<ensemble> ::= boolean | integer | real | char | <liste>

<contrainte d'inégalité> ::= <signe d'inégalité> <paramètre d'inégalité>
<signe d'inégalité> ::= < | > | ≤ | ≥ | ≠
<paramètre d'inégalité> ::= <constante> | <identificateur de variable>

Les espaces entourant les signes d'inègalités ou d'appartenance sont ignorés.
Le test de différence peut être généralisé aux structures formelles.

<contrainte d'inégalité> ::= ≠ <structure formelle>

II. 1. 5. Les structures formelles.

Les structures formelles se caractérisent par un identificateur formel et une liste de paramètres éventuellement vide. Le nombre de paramètres est appelé l'**arité** de la structure. Deux structures formelles particulières sont distinguées, les atomes qui sont les structures formelles d'arité nulle et les listes dont l'identificateur est vide. Les listes peuvent être d'arité quelconque.

<structure formelle> ::= <atome> | <liste> | <fonction formelle>
<atome> ::= <identificateur formel>
<liste> ::= () | (<liste de paramètres>)
<fonction formelle> ::= <identificateur formel> (<liste de paramètres>)

Par compatibilité avec les autres idiomes PROLOG, les écritures suivantes sont équivalentes:
() *et* [] *et* nil
(a,(b,())) *et* [a | [b]] *et* [a.b.nil].
Cependant, afin d'éviter toute ambiguïté, le point est toujours l'opérateur de suite binaire lorsqu'il apparait entre crochets. Il faut obligatoirement utiliser l'écriture parenthèsée lorsque des réels apparaissent dans la suite.

II. 1. 6. Les structures évaluables.

Les structures évaluables sont soit des variables, soit des fonctions évaluables. Remarquons qu'il n'y a pas de différence entre une variable et une fonction évaluable d'arité nulle.
La version actuelle ne présentant ni mode de définition, ni mode d'appel de fonctions évaluables, seules les fonctions prédéfinies, appelées opérateurs, sont utilisées. La liste de ces fonctions peut facilement être enrichie en fonction des possibilités du matériel. Leur nom peut être un caractère non alphanumérique (+, -, ...) ou respecter la syntaxe des identificateurs de variables (Mod, Div, ...).
La notation préfixée est systématiquement utilisée.
Il est possible de mettre des contraintes sur une fonction évaluable; celle-ci seront vérifiées sur le résultat de la fonction.
Le résultat d'une fonction est une variable implicite ou explicite.

<fonction évaluable> ::= <opérateur> (<liste de paramètres>) <liste de contraintes>
<fonction évaluable> ::= <opérateur> (<liste de paramètres>) = <variable>
<opérateur> ::= + | - | * | ÷ | Mod | Div | Abs | Trunc | Ou | And | Not
<opérateur> ::= *autres fonctions usuelles implantées*

Les espaces entourant les opérateurs sont ignorés.

II. 2. LA SEMANTIQUE.

II. 2. 1. Les clauses.

Une clause s'interprète comme l'implication logique du corps sur la tête. Dans le monde d'un programme, un fait est satisfait pour toute assignation des variables, la tête d'une règle est satisfaite pour toute assignation de variables qui satisfait tous les prédicats du corps de la règle, un but s'interprète comme la question de l'existence d'assignation qui le satisfasse.

Exemple :

père (jacques, jean).
se traduit par jacques est le père de jean.
:- père (X, jean).
signifie qui est le père de jean?
Evidemment, l'exécution d'un tel programme fait apparaitre la solution:
X = jacques.

Deux clauses ayant le même identificateur pour leur prédicat de tête s'interprètent comme une disjonction des corps de ces clauses.

Exemple:
parent (X, Y) :- père (X, Y).
parent (X, Y) :- mère (X, Y).
signifient qu'un parent de Y est son père ou sa mère.

Un corps de clause contenant plusieurs prédicats s'interprète comme la conjonction de ces prédicats.

Exemple:
grand_père (X, Y) :- père (X, Z), parent (Z, Y).
définit le grand-père de Y comme le père d'un de ses parents.

II. 2. 2. Les prédicats.

Selon les assignations affectées à ses variables, un prédicat peut être satisfait ou non. L'ensemble des prédicats d'un programme forme un monde clos. Pour un but donné, l'ensemble des assignations qui le satisfont peut être vide ou au contraire contenir plusieurs solutions voire une infinité.

Exemple:
père (jacques, jean).
mère (marie, jean).
parent (X, Y) :- père (X, Y).
parent (X, Y) :- mère (X, Y).

Le but
:- parent (X, jean).
reçoit les deux réponses
X = jacques.
X = marie.

alors que le but

:- parent (X, jacques).
n'a aucune solution dans le programme.

La coupure est un prédicat toujours satisfait dont le rôle est uniquement d'offrir un contrôle sur l'exécution du programme.

II. 2. 3. Les constantes.

Les quatre types n'appellent pas de commentaires particuliers.
L'intervalle de définition des entiers et des réels dépend de l'implantation.

II. 2. 4. Les variables.

La nature d'une variable de PROLOG est de pouvoir être liée à une constante , à une autre variable ou à une structure. La célèbre instruction des langages impératifs **n:=n+1** n'a pas de sens ici.

La portée d'une variable est la clause dans laquelle elle apparait. Il n'y a pas de variable globale. Il ne peut y avoir d'effet de bord entre clauses.

Les contraintes englobent sous la même écriture des objets différents proposés dans plusieurs idiomes PROLOG. Une liste de contraintes sur une variable se traduit par la conjonction (ET logique) de ces contraintes.

Les contraintes d'inégalité remplacent les prédicats prédéfinis de PROLOG.

Exemple:

Le programme
mère (marie, jean).
mère (marie, alain).
frère (X, Y) :- mère (Z, X), mère (Z, Y).
:- frère (X, Y).

propose quatre solutions
X = jean, Y = jean.
X = jean, Y = alain.
X = alain, Y = jean.
X = alain, Y = alain.

alors que le même programme avec une définition de frère incluant la contrainte de différence
frère (X, Y) :- mère (Z, X), mère (Z, Y:≠X).
ne propose plus que les deux solutions
X = jean, Y = alain.
X = alain, Y = jean.

Les contraintes d'appartenance permettent un typage des variables.

Exemple:

Le programme calculant la moyenne d'une liste de notes pourra tenir compte des absences.
moyenne ((), 0, 0).
moyenne ((X: in integer, L), + (S, X), + (N, 1)) :- moyenne (L, S, N).
moyenne((abs, L), S, N) :- moyenne (L, S, N).

Les contraintes de liaisons apparaissent sous le nom de modes dans CONCURRENT PROLOG[SHA&_83]. Elles imposent l'utilisation de variables d'entrée ou de sortie. Elles sont

reprises de manière facultative dans PARLOG [CLA&_84] et dans certains PROLOG [WARR_78]. Dans LAIOS, elles sont aussi laissées à l'initiative du programmeur. Elles s'interprètent de la façon suivante: ? exige que la variable soit libre au moment de l'unification (ce point sera détaillé davantage lors de l'exposé du modèle d'exécution), ! exige au contraire qu'elle soit liée à autre chose qu'une variable libre.

Exemple:

La règle
grand_père (X, Y) :- père (X, Z), parent (Z, Y).
permet de demander aussi bien les grand-pères de jean que les petit-fils de jacques; mais si le programme contient de nombreux faits pour la relation père, la recherche des grand-pères de jean s'effectuera en les examinant tous.

Les contraintes de liaisons permettent d'optimiser la recherche en fonction du sens dans lequel elle est demandée en écrivant
grand_père (X:!, Y:?) :- père (X, Z), parent (Z, Y).
grand_père (X:?, Y:!) :- parent (Z, Y), père (X, Z).

Il est également possible d'éviter certaines branches infinies. Ainsi, le programme
long ((), 0).
long ((X, L), + (N, 1)) :- long (L, N).
:- long (L, N).

boucle indéfiniment en fournissant les réponses suivantes
L = (), N = 0.
L = (_, ()), N = 1.
L = (_, (_, ())), N = 2...
où _ signifie que la variable est restée libre.

Il est facile d'interdire cette boucle en réécrivant la règle de récursion avec une contrainte de liaison sur la liste:
long ((X, L:!), + (N, 1)) :- long (L, N).

Les contraintes, bien qu'assez semblables aux gardes des idiomes comme GHC [UEDA_86], en diffèrent dans la philosophie. Les gardes sont destinées à déterminer quelle clause doit être utilisée, les gardes des différentes clauses candidates étant évaluées en parallèle et la première clause dont les gardes s'achèvent sur un succès élimine les autres candidates. Les contraintes offrent au programmeur la possibilité de restreindre le choix, éventuellement même systématiquement sur une seule clause en mettant des contraintes mutuellement exclusives, mais elles ne l'imposent pas et n'ont pas à priori un rôle de contrôle sur le déroulement.

II. 2. 5. Les structures formelles.

Les identificateurs formels, qu'ils soient utilisés pour les prédicats, les fonctions formelles ou les atomes, permettent de manipuler des concepts dont le sens échappe à la machine par la seule définition de règles de manipulation.

Les exemples précédents ont déjà utilisé des atomes -jacques, jean, marie- et des listes binaires. Les exemples ci-dessous montrent l'utilisation de listes ternaires et de fonctions formelles.

Exemple:

Insertion d'un nombre dans un tri binaire:

```
insert (N, (), ((), N, ())).

insert (N, (Left_tree, N, Right_tree), (Left_tree, N, Rigth_tree)).

insert (N, (Left_tree, P :> N, Right_tree), (New_tree, P, Right_tree)) :-
    insert (N, Left_tree, New_tree).

insert (N, (Left_tree, P :< N, Right_tree), (Left_tree, P, New_tree)) :-
    insert (N, Right_tree, New_tree).
```

Exemple:

Factorisation à l'aide d'une identité remarquable:

```
fact (somme (carré (X), produit (2, X, Y), carré (Y)),
        carré (somme (X, Y)))  :- fact (somme (X, Y)).
```

Cette règle utilise deux fonctions formelles d'identificateur somme, l'une d'arité 2 et l'autre d'arité 3; ce sont, pour le programme, deux fonctions distinctes au même titre que si leurs identificateurs étaient différents.

II. 2. 6. Les fonctions évaluables.

Ces fonctions sont indispensables dans un langage de programmation. Dans les premières versions de PROLOG, elles étaient assez mal intégrées parmi les prédicats. Des versions plus récentes utilisent un prédicat prédéfini souvent nommé eval qui admet deux paramètres, le premier étant l'expression à évaluer et le second la variable résultat [PROL_87].

LAIOS utilise une autre solution qui consiste à étendre la notion de variable à celle de fonction évaluable. Une fonction évaluable peut être vue comme sa variable résultat avec des paramètres et une méthode de calcul ou si l'on préfère une variable peut être considérée comme le résultat d'une fonction évaluable qui étant sans paramètre ni méthode reste libre d'être liée à tout objet.

Exemple:

Le calcul de la suite de Fibonacci s'écrit en PROLOG avec deux prédicats prédéfinis eval et superieur:

```
fib (0, 1).
fib (1, 1).
fib (N, X) :- superieur (N, 2), eval (- (N, 1), N1), fib (N1, Y),
                    eval (- (N, 2), N2), fib (N2, Z), eval (+ (Y, Z), X).
```

Dans le PROLOG de LAIOS, la dernière règle s'écrit

```
fib (N :≥2, + (X, Y)) :- fib (- (N, 1), X), fib (- (N, 2), Y).
```

La possibilité de rendre explicite la variable résultat est illustrée dans l'écriture équivalente de cette règle:

```
fib (N, + (X, Y)) :- fib (- (N, 1) = N1 :≥1, X), fib (- (N1, 1), Y).
```

Une extension future de ce langage consistera à autoriser l'utilisation de fonctions évaluables définies par l'utilisateur lui-même. Cette extension nécessitera la mise au point d'une syntaxe

permettant de définir ces fonctions et d'une méthode régissant l'appel de ces fonctions lors de l'exécution.

II. 3. LE DEROULEMENT.

Le principe fondamental d'exécution est la **réécriture** du but jusqu'à l'obtention d'un but vide. Lorsqu'un but vide est obtenu au bout d'une suite finie de réécritures, il existe une assignation de toutes les variables rencontrées qui satisfait tous les buts de la suite. L'**historique** des réécritures détermine alors cette assignation des variables.

II. 3. 1. Réécriture d'un but.

Soit un but B non vide, formé d'une conjonction de prédicats P_i, $1 \leq i \leq n$. L'historique de ce but est synthétisé en un ensemble de liaisons appelé **environnement** E du but B. Dans B, on distingue un prédicat P_i complètement évalué; la stratégie et la définition d'un prédicat complètement évalué seront précisés à la fin de ce paragraphe. La réécriture de B selon P_i se déroule en deux étapes.

Le prédicat P_i est comparé aux prédicats de tête de toutes les clauses. Cette comparaison particulière est appelée unification [MAR&_82]; elle peut réussir ou échouer.
L'unification de deux objets satisfait aux règles suivantes:

- l'unification de deux constantes réussit si et seulement si les constantes sont égales;
- l'unification de deux structures formelles réussit si et seulement si les structures ont même identificateur, même arité et si leurs paramètres respectifs de même rang s'unifient;
- l'unification d'une variable libre avec un objet réussit si la liaison de la variable avec l'objet est autorisée. Une liaison entre une variable et un objet est autorisée si l'objet satisfait les contraintes de la variable. Une liaison autorisée est reportée sur toutes les occurences de la variable qui est désormais réputée liée;
- l'unification d'une variable liée à un objet A avec un objet B réussit si et seulement si l'unification de l'objet A et de l'objet B réussit;
- une fonction évaluable se comporte pour l'unification comme sa variable résultat.

S'il existe une clause C_l dont la tête T_l unifie avec le prédicat P_i alors la réécriture de B se poursuit sinon elle échoue. La poursuite de la réécriture consiste à construire le nouveau but B' et son nouvel environnement E', réécriture de B et E; cette étape est appelée évaluation de B en B'.
Le but B' est formé à partir du but B dans lequel le prédicat P_i a été remplacé par le corps K_l de la clause C_l. Les variables sont renommées pour éviter toute homonymie. E' est initialement formé de E.
Pour chaque occurence d'une variable liée dans B' ou E', l'objet auquel cette variable a été liée se substitue à la variable. Si une substitution ne satisfait pas une contrainte attachée à la variable, l'évaluation échoue. Toute fonction évaluable dont les paramètres ne contiennent plus de variable est calculée. Le résultat de l'opération est lié à la variable résultat. Si la liaison échoue du fait des contraintes, l'évaluation échoue, sinon le résultat est substitué à la fonction; de plus si le résultat est explicite, les liaisons sont reportées sur toutes les occurences de la variable qui sera remplacée par sa valeur.

Si l'évaluation réussit, la réécriture a réussi. Dans le but B', seront réputés complètement évalués des prédicats dont les paramètres ne comportent plus de fonction évaluable.

Exemple:

Soit le programme
"clause 1"
fib (0, 1).
"clause 2"
fib (1, 1).
"clause 3"
fib (N :≥2, + (X, Y)) :- fib (- (N, 1), X), fib (- (N, 2), Y).
"but"
:- fib (2, X).

Il se déroule comme suit:

But initial et environnement initial:
"but 0"
:- fib (2, X0).
{env 0}
{X0=X}.

Réécriture du but 0:

- Unification du but 0 et de la clause 1 : échec car 2≠0;
- Unification du but 0 et de la clause 2 : échec car 2≠1;
- Unification du but 0 et de la clause 3 : succès;

- Liaisons: 2 avec N1 autorisée car 2≥2, X0 avec + (X1, Y1);

- Substitution du premier prédicat du but :
:- fib (- (N1, 1), X1), fib (- (N1, 2), Y1).
{X0=X}.

- Substitution des variables liées:
:- fib (- (2, 1), X1), fib (- (2, 2), Y1).
{+ (X1, Y1)=X}.

- Calcul des fonctions évaluables :
"but 1"
:- fib(1, X1), fib (0, Y1).
{env 1}
{+ (X1, Y1)=X}.

Réécriture du but 1:

- Unification du but 1 et de la clause 1 : échec car 1≠0;
- Unification du but 1 et de la clause 2 : succès;
- Unification du but 1 et de la clause 3 : échec car 1<2;

- Liaisons: X1 avec 1;

- Substitution du premier prédicat du but :
:- fib (0, Y1).
{+ (X1, Y1)=X}.

- Substitution des variables liées:
:- fib (0, Y1).
{+ (1, Y1)=X}.

- Calcul des fonctions évaluables :
"but 2"
:- fib (0, Y1).
{env 2}
{+ (1, Y1)=X}.

Réécriture du but 2:

- Unification du but 2 et de la clause 1 : succès;
- Unification du but 2 et de la clause 2 : échec car 0≠1;
- Unification du but 2 et de la clause 3 : échec car 0<2;

- Liaisons: Y1 avec 1;

- Substitution du premier prédicat du but :
:- .
{+ (X1, Y1)=X}.

- Substitution des variables liées:
:- .
{+ (1, 1)=X}.

- Calcul des fonctions évaluables :
"but 3"
:- .
{env 3}
{2=X}.

Le but 3 étant vide, le résultat X=2 est affiché.

II. 3. 2. Stratégie de déroulement.

Le principe de PROLOG est d'explorer toutes les solutions possibles. Aussi le choix du prédicat du but candidat à l'unification est-il toujours le prédicat le plus à gauche (stratégie de gauche à droite). Chaque fois qu'un prédicat du but unifie avec le prédicat de tête d'un fait, comme le corps du fait est vide, la substitution devient un effacement ce qui permet de passer au prédicat suivant.

Un succès est obtenu lorsque après un nombre fini de réécritures, le but obtenu est vide. L'environnement permet de mettre en exergue la solution trouvée. La présentation de cette solution ne contient que les variables du but initial et les objets auxquels elles sont liées. Une variable peut rester libre, elle apparaitra alors sous forme d'un souligné.

Lors de la recherche de la clause dont la tête unifie avec le prédicat le plus à gauche du but, les clauses sont examinées dans l'ordre dans lequel elles ont été écrites. Lorsqu'une réécriture réussit, un point de choix est alors créé avec le but avant réécriture, son environnement avant réécriture et la clause suivante dans la liste des clauses. Les points de choix sont empilés au fur et à mesure des réécritures (stratégie en profondeur d'abord).

Lorsqu'il n'y a plus de réécriture possible du but, soit parce que celui-ci est vide (succès), soit parce que la tentative de réécriture du prédicat le plus à gauche avec toutes les clauses a échoué (échec), il y a remontée (backtracking) au point de choix le plus récent. Le but du point

de choix sert de nouveau but avec son environnement conservé avec lui et les tentatives de réécritures sont reprises à partir de la clause mémorisée dans le point de choix.

Cette stratégie exige l'implantation de trois ou quatre piles pour pouvoir être exécutée sur une machine conventionnelle [CHAS_86].

Ce principe de déroulement peut conduire à des bouclages infinis, éventuellement en laissant inexaminées des solutions.

Exemple:

Le programme de concaténation de deux listes présenté ci-dessous est inhabituel dans sa forme, mais il met mieux en évidence la stratégie du déroulement de PROLOG.

"clause 1"
conc ((X, L), M, (X, N)) :- conc (L, M, N).
"clause 2"
conc ((), L, L).
"but"
:- conc (X, Y, (a, (b, ())))).

Déroulement:

"but 0"
:- conc (X0, Y0, (a, (b, ())))).
{env 0}
{X0=X, Y0=Y}.
{pile des points de retour}
ø .

Réécriture du but 0:

- Unification du but 0 et de la clause 1 : succès;

- Liaisons :
X0 et (X1, L1), Y0 et M1,
(a, (b, ())) et (X1, N1) soit X1 et a, N1 et (b, ()).

- Evaluation:
"but 1"
:- conc (L1, M1, (b, ())).
{env 1}
{(a, L1)=X, M1=Y}.
{pile des points de retour}
(but 0, clause 2).

Réécriture du but 1:

- Unification du but 1 et de la clause 1 : succès;

- Liaisons :
L1 et (X2, L2), M1 et M2, (b, ()) et (X2, N2) soit X2 et b, N2 et ().

- Evaluation:
"but 2"
:- conc (L2, M2, ()).
{env 2}
{(a, (b, L2))=X, M2=Y}.
{pile des points de retour}
(but 0, clause 2), (but 1, clause 2).

Réécriture du but 2:

- Unification du but 2 et de la clause 1 : échec à cause de () et (X3, N3);
- Unification du but 2 et de la clause 2 : succès;

- Liaisons :
L2 et (), M2 et L4, () et L4;

- Evaluation:
"but 4"
:-.
{env 4}
{(a, (b, ()))=X, ()=Y}.
{pile des points de retour}
(but 0, clause 2), (but 1, clause 2).

Le but 4 est vide donc le premier résultat est trouvé:
X= (a, (b, ())), Y= ().

Retour arrière:

- Unification du but 1 et de la clause 2 : succès;

- Liaisons :
L1 et (), M1 et L5, (b, ()) et L5;

- Evaluation :
"but 5"
:-.
{env 5}
{(a, ())=X, (b, ())=Y}.
{pile des points de retour}
(but 0, clause 2).

Le but 5 est vide donc le second résultat est trouvé:
X= (a, ()), Y= (b, ()).

Retour arrière:

- Unification du but 0 et de la clause 2: succès;

- Liaisons :
X0 et (), Y0 et L6, (a, (b, ())) et L6;

- Evaluation :
"but 6"
:-.
{env 6}
{()=X, (a, (b, ()))=Y}.
{pile des points de retour}
ø .

Le but 6 est vide donc le troisième résultat est trouvé:
X= (), Y= (a, (b, ())).

La pile est vide, le déroulement est terminé.

II. 3. 3. Contrôle du retour arrière par la coupure.

Le déroulement exposé dans le paragraphe ci-dessus consiste en une exploration exhaustive de toutes les éventualités. Pour offrir au programmeur la possibilité d'interdire l'exploration de certaines de ces éventualités, un prédicat de contrôle du déroulement, la coupure, a été installé dans PROLOG. L'utilisation de ce prédicat est également autorisée dans le PROLOG de LAIOS bien que son mécanisme soit peu compatible avec le parallélisme. Ce point sera discuté avec l'exposé du parallélisme dans LAIOS.

Le mécanisme de la coupure est que son franchissement fait disparaitre les points de choix accumulés dans la pile après l'appel de la clause contenant la coupure franchie.

Exemple:

Le programme
père (jacques, jean).
mère (marie, jean).
parent (X, Y) :- père (X, Y), /.
parent (X, Y) :- mère (X, Y).

répondra à la requête
:- parent (X, jean).
par
X = jacques.

Pour la réécriture du but initial, les deux dernières clauses étaient candidates, l'examen de la première se faisait alors que la seconde était empilée en attente. Le succès du prédicat père permettait de franchir la coupure et entrainait la disparition de la pile de la clause permettant de trouver la mère .

Par contre, à la requête
:- parent (marie, X).

la réponse ***X=jean*** *apparaitra car l'échec de* ***père (marie, Y)*** *ne permettra pas de franchir la coupure et la dernière clause candidate sera examinée.*

Exemple:

Le programme:
mère (marie, jean).
mère (marie, alain).
frère (X, Y) :- mère (Z, X), /, mère (Z, Y :≠ X).
:- frère (X, Y).

ne fera apparaitre que la solution
X= jean, Y= alain.

car le premier succès du prédicat ***mère (Z, X).*** *obtenu avec le premier fait permet de franchir la coupure et interdit d'examiner le second. Par contre, ces deux faits seront examinés pour le dernier prédicat du corps de la règle et sans la contrainte* ***:≠X,*** *deux solutions apparaitraient* ***X= jean, Y= jean.*** *et* ***X= jean, Y= alain.***

CHAPITRE III.
LE FORMAT INTERNE DE LAIOS.

Ce chapitre présente la traduction de la syntaxe présentée au chapitre précédent en objets directement utilisables par la machine. Cette traduction en objets reste très proche du PROLOG de LAIOS ce qui a deux conséquences, le compilateur sera simple à écrire et il sera possible d'écrire un programme performant d'aide à la correction (debugger) qui fera partie de l'environnement de programmation de LAIOS.

III. 1. PRINCIPES FONDAMENTAUX.

La représentation interne des objets de LAIOS permet de s'affranchir du langage machine du processeur utilisé dans le module. Cette représentation est en harmonie avec l'architecture multiprocesseur.

III. 1. 1. Les objets autodéfinis.

La difficulté d'écrire un compilateur de PROLOG en langage machine tient au fait que le traitement à effectuer dépend de la nature de l'objet traité et que cette nature n'est connue que lors du déroulement.

Il est donc nécessaire que l'objet contienne des informations sur sa nature car ces informations ne peuvent être contenues dans l'instruction. C'est ce que Myers [MYER_82] appelle des **objets autodéfinis**. Il prend l'exemple de l'addition pour illustrer son propos; dans un processeur classique, les différentes instructions selon qu'il s'agisse d'additions d'entiers naturels, d'entiers signés, d'entiers en décimal codé binaire, de réels en flottant, de réels double précision en flottant, etc... permettent d'interpréter les deux termes de la somme qui ne sont que des suites de bits connues par leurs adresses, le choix étant fait à la compilation. Une machine travaillant avec des objets autodéfinis ne contient qu'une instruction générique d'addition et à l'adresse de chacun des termes de la somme se trouve un objet avec une **étiquette** (tag) indiquant sa nature.

Les avantages des données étiquetées sont multiples. Tout d'abord, un contrôle est effectué sur les objets ce qui permet d'éviter des opérations absurdes. Ensuite, le jeu d'instruction est réduit à un petit nombre d'instructions génériques; avec l'essor de la microprogrammation, il est possible d'accélérer l'interprétation de ces instructions en fonction de l'étiquette de l'objet [KOYA_86]. De plus, Myers montre que le fait d'indiquer la nature de l'objet sur l'objet et non sur l'instruction qui le concerne n'entraîne pas un surcoût notable de l'occupation mémoire.

Dans LAIOS, il est fait un très large appel aux données autodéfinies et les instructions génériques sont réduites au minimum.

III. 1. 2. La structuration en blocs.

Une manière naturelle d'implanter les structures dynamiques est l'utilisation des **pointeurs**. Dans une architecture monomémoire, - avec un processeur ou plusieurs se la partageant -, l'utilisation de pointeurs est la façon la plus économique en temps de réaliser les modifications successives de structures. Ce n'est pas systématiquement la meilleure solution pour minimiser

l'encombrement mémoire.

Surtout, cette implantation des structures est très malcommode à déplacer d'un emplacement mémoire à l'autre puisqu'elle contient des adresses. Or, dans les architectures multiprocesseurs à mémoire distribuée comme LAIOS, des données inamovibles entrainent un surcoût de communication. Une structuration des données facilitant les déplacements a été recherchée.

Dans LAIOS, les structures sont implantées sous forme de **blocs**, un bloc étant un ensemble borné de mots mémoire consécutifs. La linéarisation de la structure exige un parenthésage, LAIOS utilise celui proposé par Shobatake [SHO&_86] pour son unificateur cellulaire; chaque objet contient comme information son arité, l'arité d'un objet étant le nombre d'objets introduits.

Une telle structure est évidemment déplaçable facilement par simple **copie** et il suffit de connaître sa première adresse pour y accéder. La structure de bloc favorise les algorithmes ayant un accès séquentiel aux objets du bloc et défavorise ceux qui ont un accès aléatoire. La structure de bloc ne favorise pas les modifications de structure, c'est pourquoi les nombreuses copies pour déplacement seront mises à profit pour effectuer ces modifications, une mémorisation des modifications à effectuer étant réalisée de façon temporaire sous forme de pointeurs.

III. 2. LES CELLULES DE LAIOS.

LAIOS est une machine à **mots** de 32 bits. Chaque objet se traduit par une **cellule**; celle-ci occupe un ou plusieurs mots.

III. 2. 1. Les champs d'une cellule.

Pour les raisons exposées au chapitre précédent, une cellule contient trois champs. Le premier est son **étiquette** qui précise la nature de l'objet. Le second est son **arité** qui traduit la structure de l'objet. Le dernier est la **valeur** de l'objet..

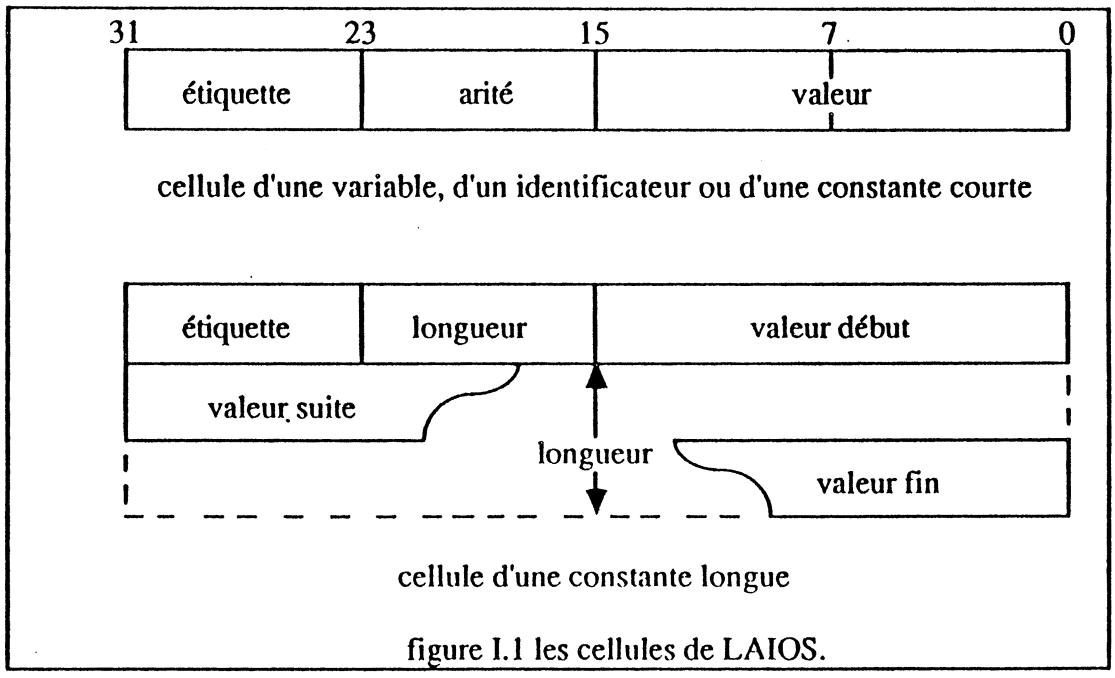


figure I.1 les cellules de LAIOS.

L'étiquette occupe les huit bits de poids fort du premier mot de la cellule, l'arité les huit suivants. La valeur occupe d'abord les seize bits restants du mot puis, si celà est insuffisant, elle

peut se prolonger sur un nombre entier de mots consécutifs de 32 bits

III. 2. 2. Les étiquettes d'une cellule.

Chaque bit de l'étiquette a un sens propre, aussi, dans ce pargraphe, les étiquettes seront-elles écrites en binaire.

Les quatre bits de poids fort désignent la nature de l'objet. Ils s'interprètent comme suit:

- 0000 : pour une variable libre,
- 0100 : pour une variable liée qui n'est pas un résultat de fonction,
- 0110 : pour un résultat de fonction,
- 1000 : pour une constante courte,
- 1001 : pour une constante longue,
- 1010 : pour un identificateur formel utilisateur,
- 1011 : pour un identificateur formel système,
- 1101 : pour un identificateur de fonction évaluable,
- 1111 : pour un identificateur de contrainte.

Ce codage est basé sur les principes suivants:

- le bit 31 sépare les variables du reste,
- le bit 30 code la liberté d'une variable,
- le bit 29 sépare les résultats de fonctions des autres variables,
- les bits 30 et 29 codent la nature des objets non variables: constantes, identificateur formel, fonction évaluable ou contrainte,
- le bit 28 est actuellement inutilisé pour les variables, il distingue les constantes courtes des constantes longues et les identificateurs utilisateurs des identificateurs prédéfinis.

Les autres codes sont réservés pour des extensions futures. Il est envisagé de leur donner le sens suivant:

- 0xx1: pour des variables en mémoire globale partagée dont l'adresse serait sur 32 ou 48 bits,
- 1100: pour des identificateurs de fonctions évaluables utilisateur,
- 1110: pour des identificateurs de contraintes utilisateur.

Les quatre bits de poids faible apportent des informations supplémentaires pour certaines catégories. Les constantes sont précisées ainsi:

- 0111 : constante booléenne,
- 1011 : constante entière,
- 1101 : constante réelle,
- 1110 : chaîne de caractères.

La même interprétation des bits sert à traduire les contraintes de typage prédéfini des variables, le zéro autorisant le type et le un l'interdisant. Par exemple, le code 0000 autorise toute liaison, le code 1001 n'autorise que les liaisons avec des rééls ou des entiers, il traduit soit **: in (integer,real)**, soit **: out boolean : out char**.

Les identificateurs systèmes utilisés sont également nommés par ces quatre bits:

- 0000 : introduit un bloc,
- 0010 : introduit un en-tête de modèle,
- 0011 : introduit un en-tête de processus,

- 0100 : introduit un tableau de variables,
- 0101 : introduit un tableau de constantes.

La raison essentielle de ce codage est de libérer le champ valeur pour un autre usage. Même avec des développements futurs, le nombre de ces identificateurs doit rester suffisament faible pour admettre le codage sur quatre bits.

III. 2. 3. L'arité d'une cellule.

L'arité étant un nombre entier positif défini sur huit bits, elle est inférieure à 256. Selon l'étiquette, elle s'interprète différement.

L'arité d'une variable indique le nombre de contraintes attachée à la variable; elle ne tient pas compte des contraintes d'appartenance exprimées dans l'étiquette..

L'arité d'une constante indique le nombre de mots supplémentaires nécessaires pour l'écriture de la constante. Il faut noter que dans ce cas, le terme d'arité est impropre et devrait être remplacé par longueur. Cette information a cependant la même utilité car c'est elle qui permet de parcourir l'objet. Toutefois, à la différence d'une véritable arité, les mots introduits ne doivent pas être considérés comme premiers mots de nouvelles cellules dont il faut déchiffrer de nouveau l'arité mais comme continuations de la même cellule.

L'arité d'un identificateur formel indique son nombre de paramètres. Si l'arité est nulle , il s'agit d'un atome.

L'arité d'une fonction évaluable est égale au nombre de ses paramètres augmenté d'une unité pour la variable résultat qui apparaitra à la fin. Ainsi, une fonction évaluable est complètement parenthésée entre son identificateur et son résultat.

L'arité d'un identificateur de contrainte indique son nombre de paramètres.

III. 2. 4. La valeur d'une cellule.

Dans ce paragraphe, toutes les valeurs sont écrites en hexadécimal.

La valeur d'une variable libre est généralement 0000. Toutefois, ce champ peut être momentanément utilisé pour noter le nouveau nom de la variable lorsque, pour éviter des conflits de nom entre variables de clauses différentes, les variables d'une clause sont renommées.

La valeur d'une variable liée est toujours une adresse locale sur 16 bits. Cette indirection est utilisée pour mémoriser un lien avec un objet, éventuellement en attendant de substituer l'objet à la variable; le champ valeur contient l'adresse de l'objet. Comme il apparaitra dans l'architecture des modules, chaque module possède dans une zone mémoire appelée brouillon, deux tableaux agencés de telle sorte qu'une adresse dans un de ces tableaux soit équivalente à un nom de variable. L'indirection est alors utilisée pour traduire le lien que représente le nom d'une variable, deux occurrences d'une même variable correspondent à deux variables liées à un même mot du tableau.

Il n'y a que deux constantes booléennes, codées sur un mot (constante courte) avec pour valeur FFFF pour vrai et 0000 pour faux.

Les entiers courts (arité zéro) et les entiers longs (arité un) sont codés en binaire complément à deux respectivement sur 16 bits et sur 48 bits dans le champ valeur.

Les réels courts (arité un) et les réels longs (arité deux) sont en format flottant IEEE respectivement sur 48 bits et 80 bits.

Une chaîne de caractères courte ne code qu'un caractère sur un mot (arité zéro); son champ valeur est formé du code ASCII du caractère suivi du code ASCII de la quote. Une chaîne de caractères longue code un nombre de caractères N supérieur à un, (arité égale à la partie entière de la division de N+2 par 4) sur arité+1 mots. Le champ valeur est la suite des codes ASCII des caractères de la chaîne terminé par le code ASCII de la quote; le dernier mot est complété, si besoin est, par des codes ASCII du retour chariot.

La valeur d'un identificateur formel utilisateur est un nombre codé sur 16 bits. La valeur 0000 étant réservée pour la liste, celà autorise l'utilisation maximale de 65535 identificateurs différents pour chaque valeur de l'arité. De même, la valeur d'une fonction évaluable ou d'une contrainte est son nom codé en un nombre de 16 bits.

Par contre, la valeur d'un identificateur formel système est égale à la longueur du bloc qu'il introduit; c'est la raison qui nous avait conduit à coder le nom de ces identificateurs dans l'étiquette. Cette information sur la taille du bloc est utilisée dans les opérations d'allocation et de désallocation de mémoire.

III. 3. LES STRUCTURES DE DONNEES.

Dans LAIOS, les cellules sont structurées en blocs, pour les raisons exposées ci-dessus, ou en tableaux.

III. 3. 1. Les concepts généraux.

Physiquement, un bloc est une suite de cellules consécutives. La première de ces cellules est un identificateur système de bloc dont l'arité détermine le nombre de structures du bloc et la valeur indique la longueur en mots du bloc.

Le parcours d'un bloc se fait à l'aide d'un compteur. Celui-ci est initialisé avec l'arité du bloc. Le passage d'une cellule à la suivante provoque une décrémentation du compteur, le décodage d'une cellule augmente le compteur de l'arité de la cellule. La dernière cellule est atteinte lorsque, après décodage, le compteur est nul.

Un tableau est un bloc particulier dans lequel toutes les structures sont identiques. L'arité du tableau est égale au nombre de structures du tableau. L'accès direct par indice est possible dès que la longueur d'une structure est connue; celle-ci peut, si elle n'est pas connue, être facilement calculée en fonction de l'arité et de la longueur. Par la suite, les tableaux qui apparaitront seront surtout des tableaux de variables.

III. 3. 2. Exemples de blocs.

Dans les exemples ci-dessous, les codages sont explicités -et non sous écriture binaire- pour une meilleure lisibilité. En plus des trois champs, étiquette, arité, valeur, apparaissent deux autres colonnes l'une de commentaires et l'autre simulant la valeur du compteur de parcours.

Les numérotations des identificateurs et des variables sont données à titre d'illustration car elles dépendent bien évidemment du contexte. La numérotation des variables fait apparaitre la page de brouillon -page 10- et l'indice dans le tableau.

Le prédicat **fib** (N : ≥2, +(X, Y)) donnera le bloc:

étiquette	arité	valeur	commentaire	compteur
bloc	1	9		1
ident. formel	2	n° 1	fib	2
var. liée	1	10-1	N	2
contrainte	1	n° 8	≥	2
entier court	0	2		1
fonct. éval.	3	n° 1	+	3
var. liée	0	10-2	X	2
var. liée	0	10-3	Y	1
résultat	0	10-4		0

Le prédicat **conc** ((X,L), (a,(b,(c,()))), (X,N)) se traduit par:

étiquette	arité	valeur	commentaire	compteur
bloc	1	15		1
ident. formel	3	n° 4	conc	3
ident. formel	2	n° 0	liste bin.	4
var. liée	0	10-1	X	3
var. liée	0	10-2	L	2
ident. formel	2	n° 0	liste bin.	3
ident. formel	0	n° 1	a	2
ident. formel	2	n° 0	liste bin.	3
ident. formel	0	n° 2	b	2
ident. formel	2	n° 0	liste bin.	3
ident. formel	0	n° 3	c	2
ident. formel	0	n° 0	liste vide	1
ident. formel	2	n° 0	liste bin.	2
var. liée	0	10-1	X	1
var. liée	0	10-3	N	0

L'apparition de la variable X deux fois dans le prédicat se traduit par l'apparition de la même adresse deux fois dans le champs de valeur.

III. 3. 3. Exemples de tableaux.

Le tableau de variables du prédicat **fib** (N : ≥2, +(X,Y)) est:

étiquette	arité	valeur	commentaire	compteur
tableau	4	5		4
var. liée	0	10-1	N	3
var. liée	0	10-2	X	2
var. liée	0	10-3	Y	1
résultat	0	10-4	somme	0

A titre d'exemple, le tableau contenant deux rééls longs: 1.0 et 3.14 s'écrit:

étiquette	arité	valeur	commentaire	compteur
tableau	2	7		2
réél long	2	1.0	3 mots	3
		-		2
		-		1
réél long	2	3.14	3 mots	2
		-		1
		-		0

Le compteur n'est pas nécessaire pour parcourir un tableau.

LIVRE SECOND.

LE MODELE.

CHAPITRE I.
LES MODELES ARBORESCENTS.

Dans le chapitre sur la sémantique du langage, a été présenté le modèle de réécriture des buts. Avec sa stratégie en profondeur d'abord et de gauche à droite, avec l'empilement des points de choix et le retour en arrière, ce modèle satisfait une architecture monoprocesseur [WARR_83]. Avec des architectures multiprocesseurs,un modèle arborescent est présenté ci-dessous qui est plus naturel à PROLOG. Plusieurs stratégies de déroulement sont développées sur ce modèle [SYRE_85].

I. 1. LES ARBRES DE RECHERCHE.

Les arbres de recherche, également nommés arbres ET/OU modélisent une situation où alternent à chaque étape des conjonctions et des disjonctions. C'est typiquement le cas de PROLOG [WISE_86].

I. 1. 1. L'arbre théorique d'un programme.

Un programme est une liste de clauses de Horn $(C_i)_{1\leq i\leq n}$, chaque clause C_i est de la forme T_i:-K_i., la tête T_i est formée d'un prédicat ou est vide, le corps K_i est une suite éventuellement vide de prédicats.

La forêt théorique d'un programme est construite à partir d'un ensemble de buts avec les règles suivantes:
- les buts du programme sont des buts de la forêt,
- à chaque prédicat d'un but est associé un **nœud OU**,
- un nœud OU associé à un prédicat P_j a autant de fils qu'il existe de clauses C_i dont la tête a le même identificateur que P_j,
- tout fils d'un nœud OU est un **nœud ET** associé à une clause C_i; si le corps K_i de cette clause n'est pas vide, il y a création d'un nouveau but pour la forêt, but formé du corps de C_i sans tête.

Il faut noter que si le programme ne contient qu'un seul but initial, la forêt se réduit à un arbre, ce qui est le cas courant. Il existe des nœuds ET particuliers n'ayant pas de fils, ceux qui sont associés aux faits, ces nœuds forment les feuilles de l'arbre. Dès qu'une récursion est présente dans le programme, l'ensemble des buts créés devient infini, l'arbre est aussi dit **infini**.

I. 1. 2. Exemples.

Les arbres théoriques présentés ci-dessous sont obtenus en applicant les règles de construction sur des programmes déjà vus dans le livre précedent.

Le premier exemple choisi est le programme de base de données familiale qui donne naissance à un arbre théorique fini.

```
"clause 1"
père (jacques, jean).
"clause 2"
père (jules, jacques).
"clause 3"
père (maurice, marie).
"clause 4"
mère (marie, jean).
"clause 5"
parent (X, Y) :- père (X, Y).
"clause 6"
parent (X, Y) :- mère (X, Y).
"clause 7"
grand_père (X, Y) :- père (X, Z), parent (Z, Y).
"clause 8"
:- grand_père (X, jean).
```

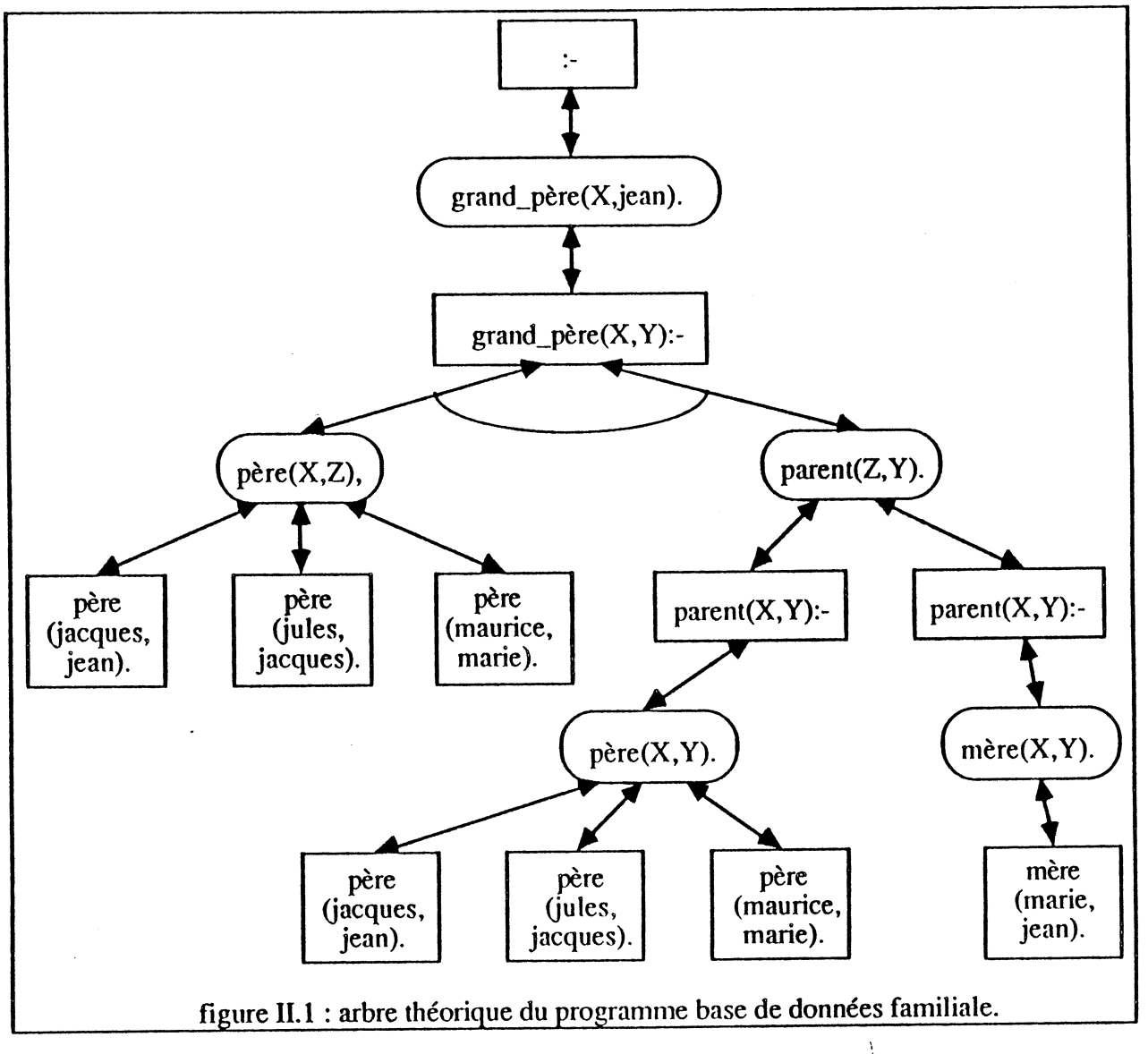


figure II.1 : arbre théorique du programme base de données familiale.

Le second exemple est le programme de calcul de la suite de Fibonnacci qui donne naissance à un arbre infini.

```
"clause 1"
fib (0, 1).
"clause 2"
fib (1, 1).
"clause 3"
fib (N :≥2, +(X,Y) ) :- fib ( -(N,1), X), fib ( -(N,2), Y).
"clause 4"
:- fib (3, X).
```

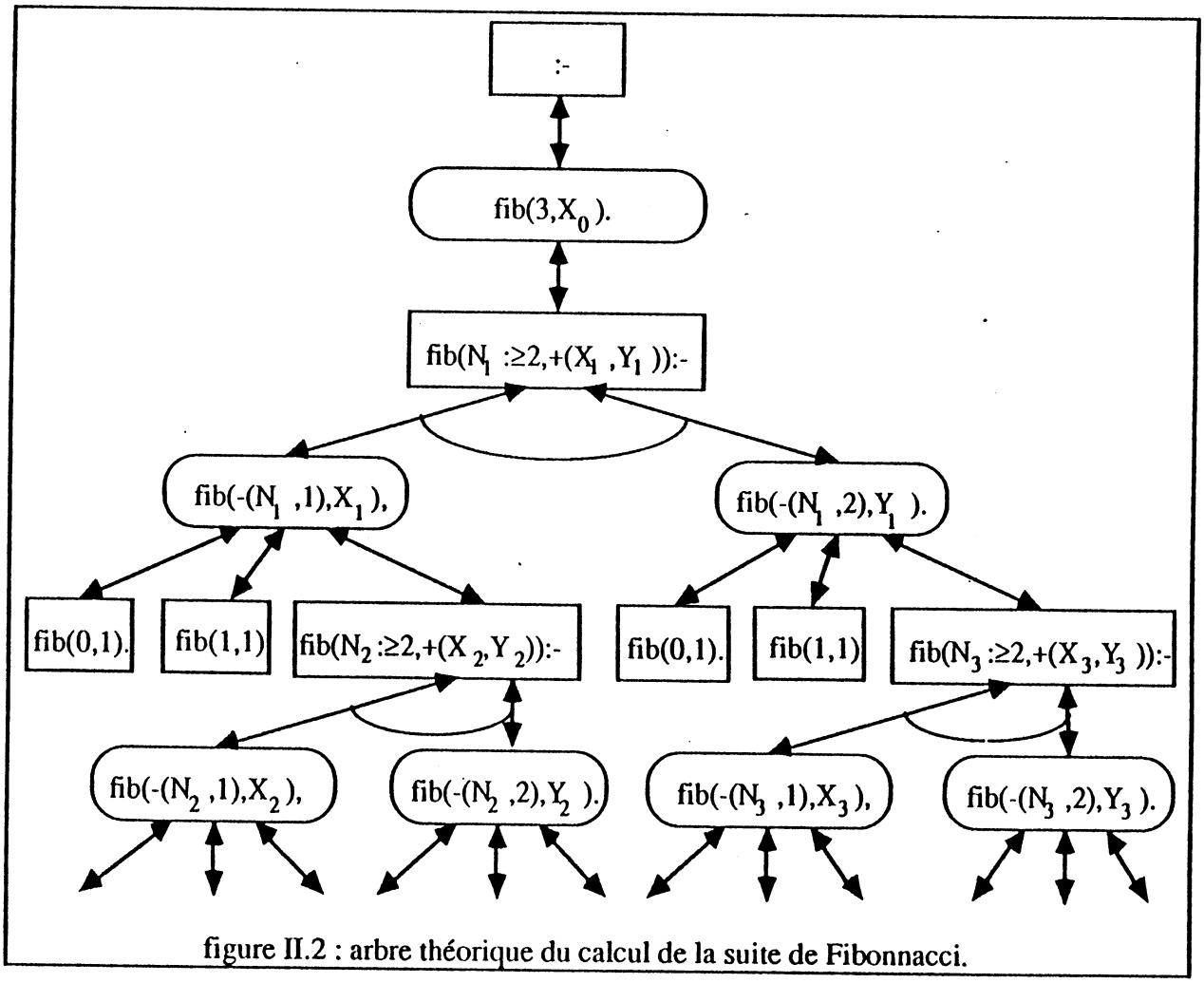


figure II.2 : arbre théorique du calcul de la suite de Fibonnacci.

Les variables sont renumérotées pour éviter tout problème d'homonymie.

I. 1. 3. Arbre déroulé.

La trace de l'exécution du programme est la répercussion sur les nœuds de l'arbre des assignations sur les variables. Dès lors que certains prédicats sont satisfaits ou non, les nœuds correspondants seront réputés avoir réussi ou échoué. Les branches issues d'un nœud qui a échoué sont des branches mortes. L'**arbre de recherche** proprement dit (arbre déroulé) est obtenu par élagage des branches mortes, il peut être fini ou infini.

Par exemple, l'arbre de recherche du programme de la suite de Fibonnacci ci-dessus est fini.

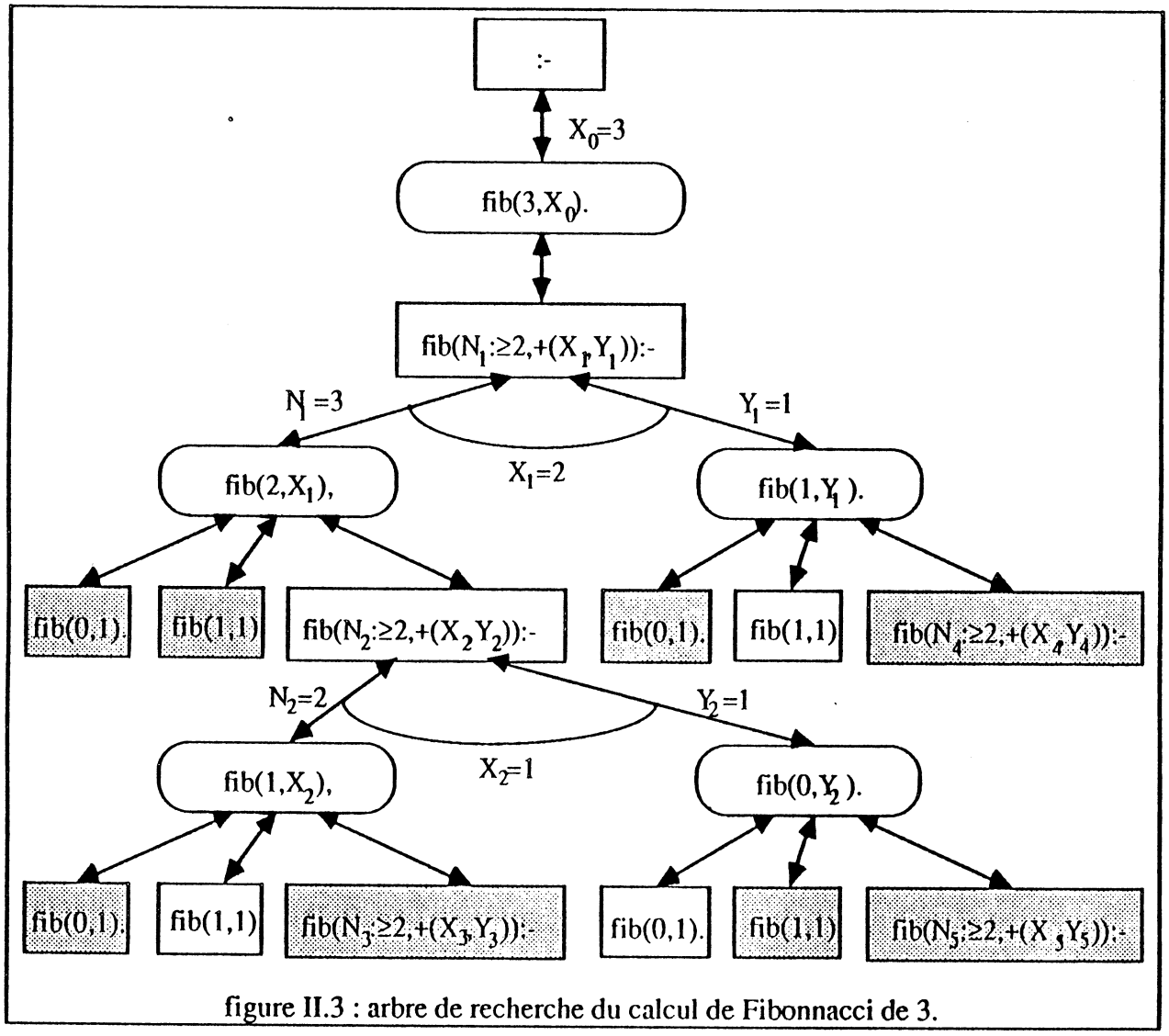


figure II.3 : arbre de recherche du calcul de Fibonnacci de 3.

Ce petit exemple met en évidence une notion importante. Pour obtenir le résultat, il faut balayer l'arbre à la descente (de la racine aux feuilles) et à la remontée (des feuilles à la racine):
$X_0 = +(X_1,Y_1) = +(+(X_2,Y_2), 1) = +(+(1,1), 1) = +(2,1) = 3.$

Une autre notion apparait ici; dans les nœuds ET, le calcul des fonctions évaluables (ici l'addition de X et Y) est retardée jusqu'à la remontée des données alors que dans les nœuds OU, ce calcul (ici les soustractions N-1 et N-2) est effectué dès la descente des données. Ce mode de fonctionnement est celui de LAIOS. D'autres choix peuvent être faits, qui retardent systématiquement les évaluations jusqu'à ce qu'elles soient nécessaires -évaluations retardées- ou au contraire qui les effectuent dès que leurs paramètres sont complètement instanciés [CRAS_86], [BOU&_87]. Ces choix imposent de véhiculer les fonctions évaluables dans l'arbre et sont, de ce fait, plus complexes à mettre en œuvre.

Le calcul est encore plus complexe sur l'arbre de la base de donnée familiale.

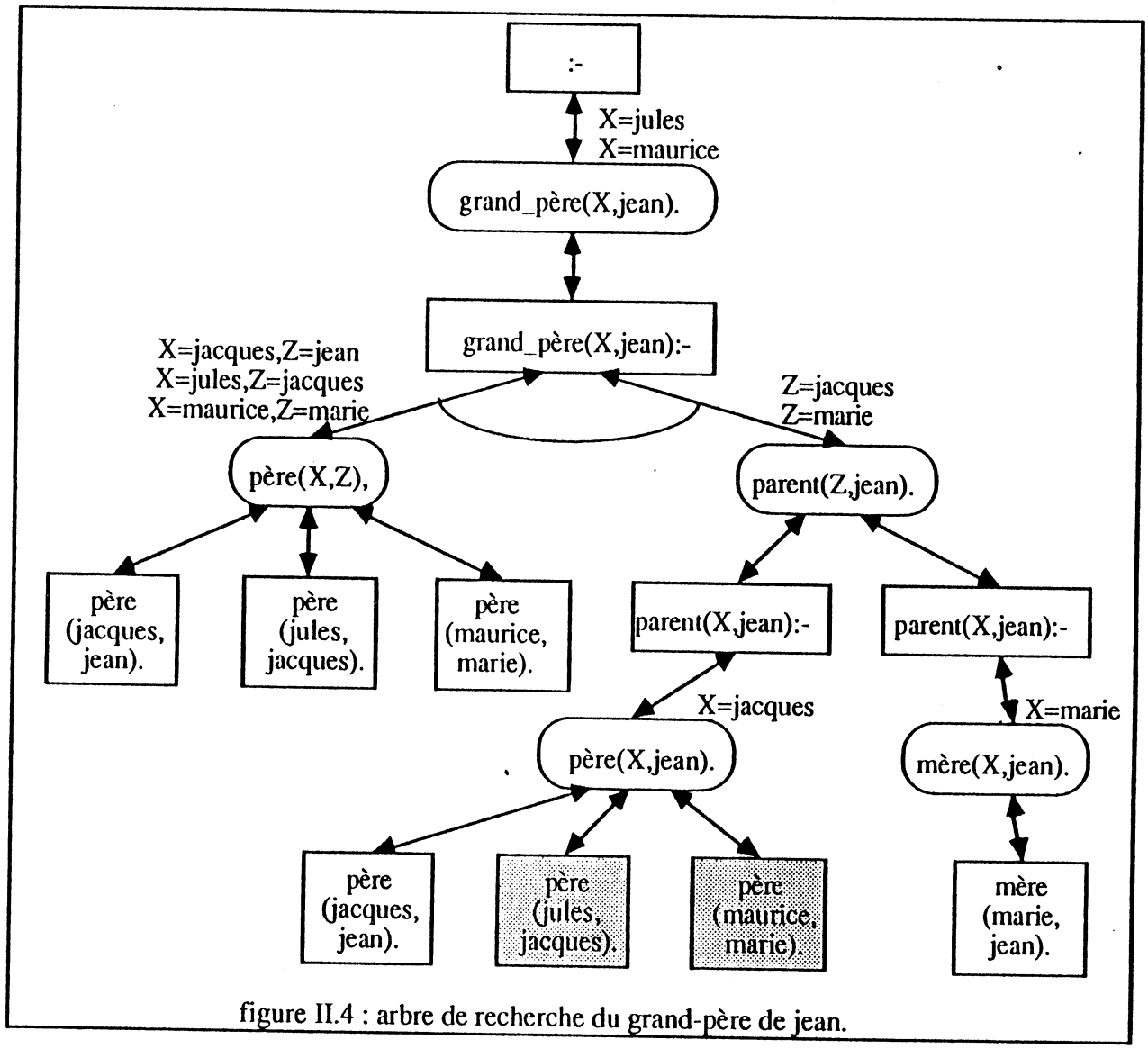


figure II.4 : arbre de recherche du grand-père de jean.

Du fait du non déterminisme, une même variable peut avoir des assignations multiples. De plus le rôle de filtre du nœud ET est ici mis en évidence sur le nœud grand-père qui assure la cohérence des assignations de la variable Z.

Le parcours des branches en parallèle sur un arbre de recherche a des conséquences différentes selon que les branches sont issues d'un nœud OU ou d'un nœud ET, on parlera de parallélisme OU ou de parallélisme ET.

I. 2. LE PARALLELISME OU.

C'est l'exploration parallèle des branches de l'arbre de recherche issues des nœuds OU; cette stratégie s'interprète aussi comme la réécriture concomittante des buts à partir des divers points de choix.

I. 2. 1. Exécution.

La caractéristique principale du parallélisme OU est l'indépendance des branches issues d'un nœud OU. L'avantage est qu'aucune synchronisation n'est nécessaire. Par contre, il est indispensable de garantir que les liaisons effectuées dans une branche ne se répercutent pas dans d'autres branches. Les deux solutions les plus couramment proposées sont la copie de buts ou les répertoires d'environnements.

Les variables d'un nœud OU peuvent recevoir plusieurs assignations distinctes, un mécanisme doit être mis en place afin de gérer les assignations multiples [LIND_84].

Enfin, le modèle d'arbre de recherche n'est pas conforme au parallélisme OU total au sens de Kalé [KALE_87]. Ce dernier exige que les différentes branches du OU n'interfèrent pas entre elles dans la résolution du but entier. Un modèle de réécritures multiples du but (assez proche de la méthode de construction de l'arbre théorique) formalise mieux ce parallélisme.

Les performances du parallélisme OU de l'arbre de recherche sont étroitement corrélées aux programmes exécutés. Un programme entièrement déterministe n'offre aucun parallélisme de ce type et aucune amélioration n'est espérée. A l'inverse, un programme combinatoire offre le maximum de parallélisme OU et son exécution peut, à l'optimum, être aussi rapide que la recherche de la première solution dans une résolution séquentielle. C'est pourquoi le parallélisme OU est intéressant dans les problèmes d'intelligence artificielle pour lesquels l'apparition de solutions est souvent due à l'exploration de nombreuses possibilités.

I. 2. 2. Implantation.

Plusieurs solutions ont été proposées et simulées.

Le **sac de jetons** est proposé par Ciepielewski [CIEP_84], [CIE&_84]. Chaque nœud non terminal prend la forme d'un jeton (token) contenant le but à résoudre et son environnement propre. Ces jetons sont réunis en une mémoire partagée par tous les processeurs. Chaque processeur peut prendre un jeton dans le sac, unifier l'unificande et en cas de succès renvoyer dans le sac toutes les solutions non terminales (ni échec, ni succès dû à un but vide).

L'arbre dynamique de processus est proposé par Conery [CON&_81]. Chaque nœud est un processus. Ces processus sont pris en charge par des processeurs communiquant entre eux. Les processus échangent des messages contenant les assignations des variables. Le modèle de LAIOS exposé plus loin dérive de ce modèle; il est également la base de l'architecture de PIM_R [ONA&_85].

I. 3. LE PARALLELISME ET.

C'est l'exploration parallèle des branches issues d'un nœud ET ce qui s'interprète comme la résolution parallèle des prédicats d'un but.

I. 3. 1. Mise en œuvre.

Si les prédicats du but sont indépendants (c'est le cas de la suite de Fibonnacci), leurs résolutions ne nécessitent aucune synchronisation. L'ensemble des solutions du but est alors le produit cartésien des ensembles de solutions de chaque prédicat. Si le problème est déterministe, ce parallélisme est bien connu sous le nom "divide and conquer".

Si les prédicats ne sont pas indépendants, ils partagent des variables. Différents modèles

peuvent être envisagés pour maintenir la cohérence sur les variables. L'efficacité de ces diverses solutions est largement dépendante du type de programme exécuté.

I. 3. 2. L'opérateur de jointure.

La première solution consiste à explorer les branches indépendamment, quitte à maintenir la cohérence sur les solutions qui remontent sur chacune des branches au nœud ET. Le nœud réalise alors l'**opération de jointure** des bases de données relationnelles [SHI&_84]. Cette solution est proposée dans EPILOG [WISE_86] sous le nom de back unification. C'est pourtant une solution coûteuse en place mémoire qui sur la plupart des problèmes, conduit à l'exécution de beaucoup de travail inutile; par exemple, sur le problème de la recherche des grand-pères dans la base de données familiale, la première branche remonte toutes les relations de pères de la base, ce qui peut faire beaucoup, lorsque deux seulement seront solutions.

Une solution plus éloignée de la philosophie de PROLOG est celle du langage GHC (**Guarded Horn Clauses**) de l'ICOT [UEDA_86], lui-même dérivé de CONCURRENT PROLOG dont le mécanisme est voisin [BEL&_86], [SUZU_85]. Dans cet idiome, le corps d'une clause est divisé en deux parties, la garde et le corps proprement dit. La garde ne contient que des prédicats ne réalisant pas de liaison sur les variables (notion voisine de la contrainte du PROLOG de LAIOS). Le nœud ET efface en parallèle les gardes de toutes les clauses unificatrices (à l'image du parallélisme OU). La première garde dont l'effacement se termine par un succès provoque l'élection de sa propre clause comme clause unificatrice et l'éviction des autres clauses. Il s'ensuit qu'au plus une solution peut être exhibée pour un prédicat donné. Il est donc possible de lancer la résolution parallèle de tous les prédicats d'un but, la jointure étant, de ce fait, simplifiée à l'extrême. Cette implantation permet la programmation efficace des algorithmes de type divide and conquer mais s'écarte résolument du non déterminisme de PROLOG car elle rend les problèmes déterministes.

Une autre solution restrictive est celle du **ET pipe line** proposée par [LIN&_84], [TAM&_84]. Pour chaque prédicat d'un but, au fur et à mesure que des solutions apparaissent, elles sont communiquées au prédicat suivant (dans l'ordre de lecture) du but. A l'inverse de la solution précédente, celle-ci ne provoque aucune accélération sur un programme déterministe, en particulier, les stratégies divide and conquer ne peuvent être programmées. Par contre, l'opération de jointure étant réalisée en pipe line, dans l'hypothèse de solutions multiples, un facteur important d'accélération apparait par rapport à la solution séquentielle. De plus, ce mécanisme ne nécessite aucune analyse de la situation ni la mise en place de synchronisation complexe. Cette solution a été retenue pour LAIOS. Une version bidirectionnelle de ce mécanisme est présentée par Nakagawa [NAKA_84].

I. 3. 3. Les synchronisations producteur-consommateurs.

L'autre grande famille de solutions pour le parallélisme ET consiste en l'établissement de synchronisations de type producteur-consommateurs sur les tâches se partageant des variables.

En l'absence de tout renseignement d'ordre sémantique sur le programme, il est impossible de définir le meilleur producteur. Des heuristiques sont proposées qui vont du plus simple, le premier prédicat dans l'ordre d'écriture du but dans lequel la variable apparait, à plus sophistiqué comme le prédicat contenant le moins de variables non complètement instanciées parmi tous les prédicats se partageant la variable. Quoiqu'il en soit, on ne peut être certain que le prédicat instancie complètement la variable pour laquelle il a été désigné producteur, ce qui peut contraindre à choisir le producteur plusieurs fois.

Un deuxième cas critique pouvant se produire est lorsque deux variables distinctes deviennent liées l'une à l'autre par le jeu des unifications. En fait, le graphe des dépendances

entre variables est modifié dynamiquement au cours de l'exécution et il faut sans cesse l'analyser et le réactualiser.

De nombreuses solutions ont été imaginées pour réaliser une synchronisation de type producteur-consommateurs entre variables partagées. Citons rapidement les principales.

Dans CONCURRENT PROLOG, Shapiro [COD&_86], [SUZU_86] demande au programmeur de l'écrire explicitement, ce qui nuit à la généralité des programmes et demande au programmeur une connaissance du déroulement de son programme.

De Groot [DeGR_85], [CHA&_85] propose de compiler cette synchronisation. Puisque deux variables distinctes apparaissant dans la tête de clause peuvent être liées lors d'une unification, elles doivent être considérées comme dépendantes par le compilateur (pire cas). Cela peut réduire considérablement le parallélisme extrait par cette méthode. Une solution voisine avec gel des variables est proposée par Kharoune [KHA&_86].

Une détection des dépendances entièrement dynamique est proposée dans la thèse de Li [LI_86], [LI&M_86]. Cette solution permet de dégager le maximum de parallélisme mais au prix d'un travail d'analyse des dépendances considérable et permanent. Il faut déterminer le surcoût que cette détection entraîne pour pouvoir estimer le gain.

I. 4. LE MODELE DE LAIOS.

Le modèle choisi est un compromis entre efficacité et simplicité dans l'optique d'une réalisation avec les moyens techniques actuels.

I. 4. 1. L'arbre de recherche dynamique.

L'exécution d'un programme dans LAIOS correspond physiquement au déroulement de l'arbre de recherche sur le réseau de processeurs. Les nœuds de l'arbre sont des processus à l'image de la solution pronée par Conery [CONE_83].

La solution consistant à compiler l'arbre théorique en attribuant à chaque processeur un ensemble [SYRE_86], généralement infini, de nœuds n'a pas été retenue. Bien qu'elle permette de minimiser les déplacements de données, cette solution ne permet pas de répartir la charge de manière satisfaisante. Au moment de la compilation, seul l'arbre théorique est connu, sans qu'il soit possible de tenir compte des branches mortes ou des branches infinies.

La solution adoptée consiste à créer dynamiquement l'arbre de recherche au fur et à mesure des besoins. Toutefois, LAIOS s'est inspiré du fonctionnement classique des microprocesseurs dans lesquels l'instruction suivante est chargée pendant l'exécution de la précédente. La création des nœuds de l'arbre est anticipée d'un étage sur l'activité.

Le principe de la **création dynamique anticipée** est le suivant: dès qu'un nœud est activé sur un calcul, ses propres fils deviennent générateurs. Pour celà, chaque processeur possède un processus système résidant appelé **processus mère**. Ces processus ont une file d'attente dans laquelle se trouvent les nœuds processus générateurs. Les processus mères se chargent entièrement de l'implantation des fils des processus générateurs. Cette phase de création s'effectue en parallèle avec la phase de calcul de la génération précédente de processus.

Comme il apparaitra plus loin, les premières simulations ont montré que les créations sont assez largement recouvertes par l'activité des parents. Le prix à payer pour cette accélération est la création de processus inutiles lorsque l'activité des parents se termine par un échec.

Cependant, le rapport entre le nombre de nœuds créés et le nombre de nœuds créés inutilement reste modeste (n pour 2^{n-1} pour un arbre de recherche binaire équilibré).

Un mécanisme semblable de décès retardé est mis en place. Ce mécanisme veut qu'un processus ayant achevé sa tâche tue ses fils mais ne se tue pas lui-même(1). Du fait des assignations multiples, un processus ayant terminé sa tâche est susceptible d'être réveillé pour une nouvelle tâche; le mécanisme de décès retardé évite de créer un nouveau processus identique. De ce fait, des processus inutiles resteront physiquement présents un certain temps, mais le gain en rapidité justifie la perte de place.

I. 4. 2. Le parallélisme OU.

Dans LAIOS, le parallélisme OU est implanté à la descente dans l'arbre. Un nœud OU active ses fils en même temps dès qu'il le peut. L'indépendance des branches est assurée par des copies multiples des données à raison d'une par fils.

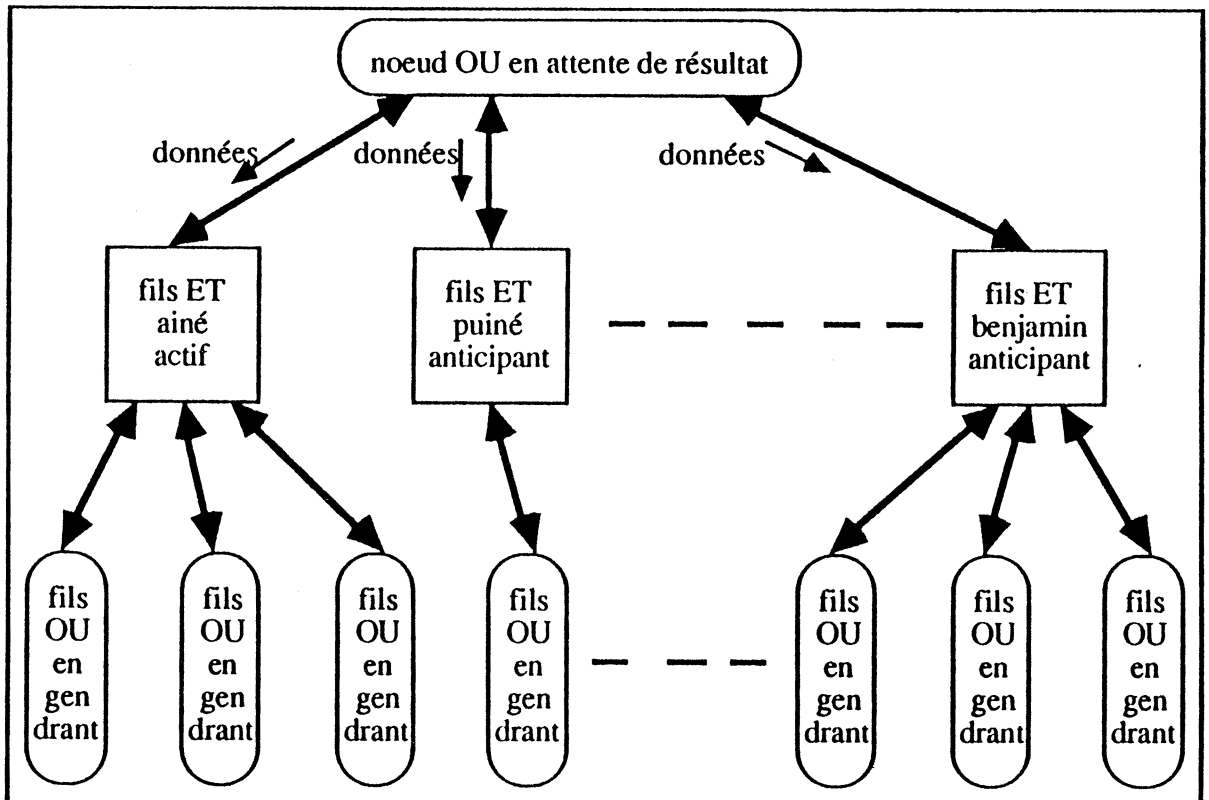


figure II.5 : parallélisme OU.

Par contre, lors de la remontée des résultats, un nœud OU respecte la stratégie de PROLOG classique. Les résultats sont récoltés séquentiellement dans l'ordre du fils ainé au benjamin. Les fils d'un nœud OU doivent donc avoir une file d'attente pour permettre une synchronisation facile de cette récolte.

Cette solution présente deux avantages. Les assignations multiples sont physiquement séparées dans le temps ce qui est simple. Le respect de la stratégie de PROLOG classique permet l'implantation de la coupure. Par contre, elle n'est pas optimale en temps et ne permet pas le parallélisme OU total. En particulier, il est clair que si la branche du fils ainé boucle indéfiniment, les solutions du fils cadet ne pourront jamais remonter, fussent-elles calculées.

(1) Certains y verront le triomphe de l'informatique sur la mythologie: Laïos était le père d'Œdipe!

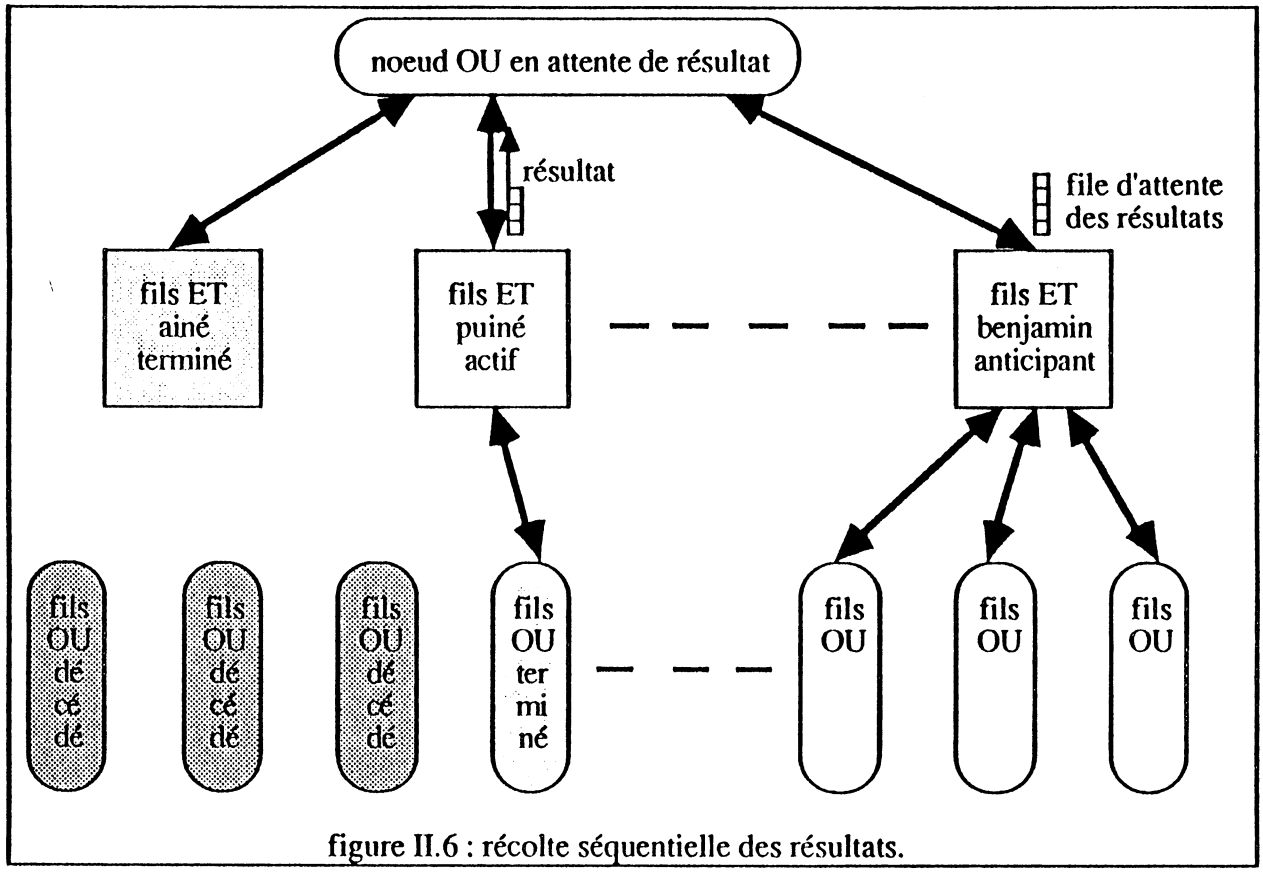


figure II.6 : récolte séquentielle des résultats.

I. 4. 3. Le ET pipe line.

Puisque LAIOS est conçu dans l'optique intelligence artificielle, les solutions multiples doivent pouvoir être examinées. L'opération de jointure est inévitable.

La réalisation de synchronisations producteur-consommateurs apparait délicate à mettre en œuvre; la compilation étant peu optimisable et l'analyse dynamique complexe. En dépit de son intérêt évident, aucune synchronisation de ce type n'est implantée sur LAIOS.

Par contre, le ET pipe line offre à la fois l'intérêt de réaliser la jointure de manière simple, d'être cohérente avec le choix fait précédemment de séparer dans le temps les assignations multiples et de proposer une accélération par rapport à la solution séquentielle non négligeable dans le cas de problèmes non déterministes. Il est donc implanté dans LAIOS.

Les fils d'un nœud ET sont donc reliés par une connexion frère-ainé-frère-cadet. A la descente, le nœud ET amorce le pipe line en transmettant une assignation des variables à son fils ainé. Pour chaque résultat que ce dernier obtiendra, il transmettra une assignation à son frère cadet. A chaque nœud de la chaîne, le flot des assignations s'enrichit des solutions multiples calculées au niveau du nœud, chaque assignation reçue apparaissant comme un problème nouveau pour le nœud. La fin du pipe line est un retour sur le nœud ET qui récupère ainsi toutes les solutions de la jointure les unes derrière les autres.

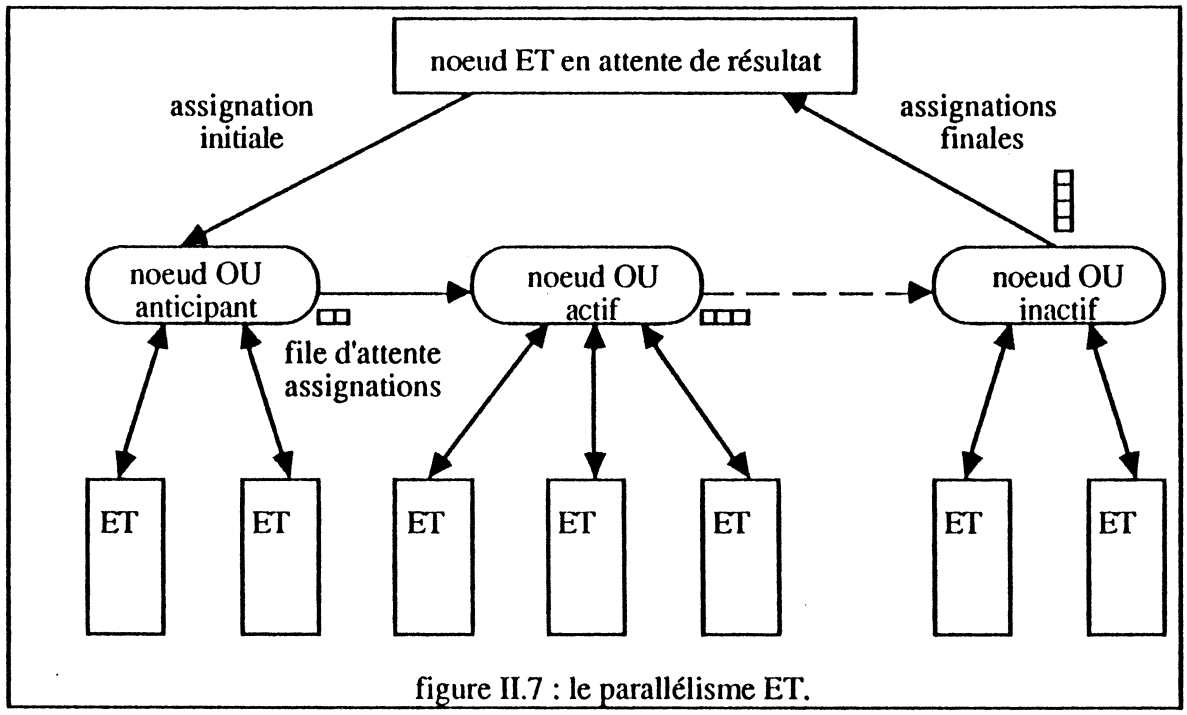


figure II.7 : le parallélisme ET.

Les nœuds le long du pipe line ont une file d'attente permettant de synchroniser le flot. Un petit nombre de messages suffit pour réaliser cette synchronisation.

I. 4. 4. La coupure.

Par souci d'une large compatibilité, la coupure est implantée dans LAIOS. Son fonctionnement dans l'arbre d'exécution se traduit par la taille (pruning) de certaines branches actives de l'arbre.

Par exemple, soit la partie de programme, dans lequel, pour simplifier, les prédicats sont supposés sans paramètres:

```
A :- B.
A :- C / D.
A :- E.
B.
C :- F.
C :- G.
D.
E.
F.
G.
```

Le déroulement séquentiel de ce programme à partir du but A est:

```
:- A.  {s'efface de trois façons, B, C/D ou E};
:- B.  {s'efface d'une seule façon};
:- .   {premier succès et retour arrière pour l'effacement de A};
:- C / D.  {C s'efface de deux façons, F ou G};
:- F / D.  {F s'efface d'une seule façon};
:- / D.  {la coupure se franchit};
```

:- *D.* {D s'efface};
:- . {second et dernier succès car le franchissement de la coupure a provoqué l'abandon de la troisième possibilité d'effacement de A et la seconde d'effacement de C}.

En dessinant la coupure comme un nœud, l'arbre théorique se représente comme suit.

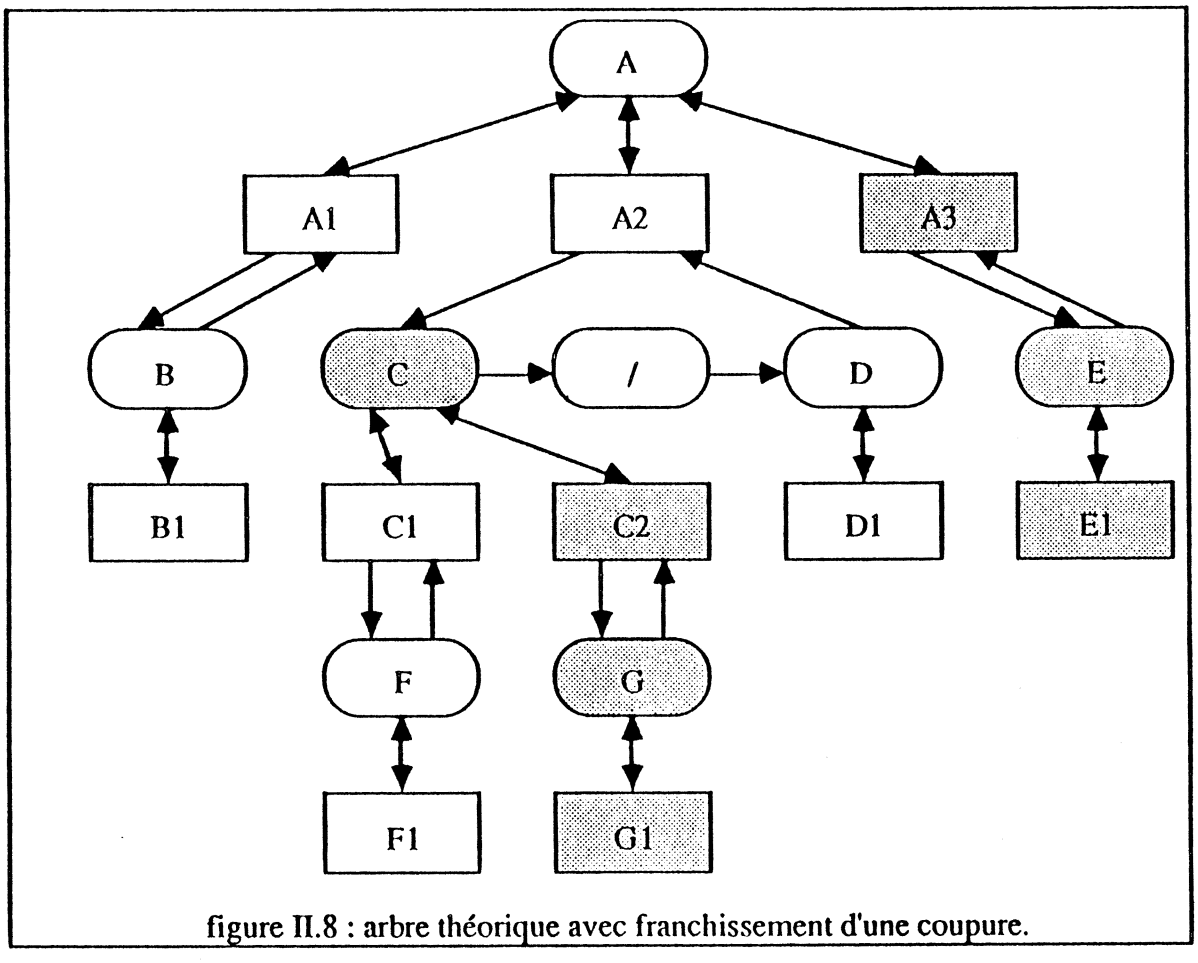


figure II.8 : arbre théorique avec franchissement d'une coupure.

Le franchissement de la coupure provoque l'abandon des branches en grisé.

Dans l'hypothèse du parcours de l'arbre avec la stratégie de LAIOS, parallélisme OU et ET pipe line, ces branches ont débuté leur activité avant le franchissement de la coupure; mais l'ordre de collecte des résultats est tel que les résultats ne peuvent remonter dans ces branches au delà des nœuds en grisé. Il suffit donc de pouvoir provoquer l'abandon dans ces branches quel que soit l'état d'avancement du travail pour obtenir un résultat sémantiquement correct du franchissement de la coupure.

LAIOS garantit donc, de manière simple, un fonctionnement correct de la coupure. Toutefois, aucune optimisation n'est faite pour essayer d'arrêter les calculs inutiles au plus tôt, ce qui serait complexe à mettre en place. Il faut rester conscient du fait que la coupure n'est pas une méthode d'optimisation dans l'exécution parallèle de PROLOG.

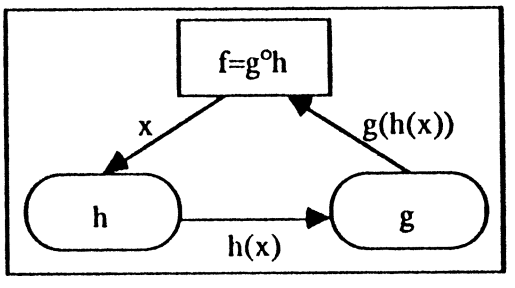
I. 4. 5. Possibilités d'évolution.

Dans sa définition actuelle, une critique pertinente de LAIOS reste l'impossibilité de programmer des stratégies divide and conquer. Certes, elles n'apparaissent pas de façon explicite dans un langage comme PROLOG mais elles demeurent parmi les façons les plus

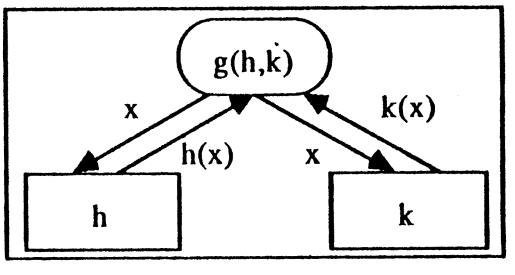
performantes d'utiliser des architectures multiprocesseurs.

L'idée de base pour autoriser ce type de programmation est de l'intégrer dans la définition des fonctions évaluables utilisateurs. Ainsi, dans un même programme pourraient cohabiter des parties non déterministes dont le calcul serait accéléré par le parallélisme OU et le ET pipe line et des parties déterministes dont le calcul pourrait s'effectuer selon des stratégies divide and conquer.

Comme celà a déjà été signalé précédemment, la définition des fonctions évaluables utilisateurs reste à intégrer dans le PROLOG de LAIOS. De même, le modèle reste à définir par rapport à celui défini ci-dessus. Toutefois, en choisissant un modèle arborescent, il est remarquable de noter l'analogie entre l'opération de composition sur les fonctions f(x)=g(h(x)) et le ET pipe line sur les prédicats:



de même que l'analogie entre la projection g(h(x),k(x)) et le parallélisme OU:



C'est cette seconde analogie qui permettrait la programmation de stratégie de type divide and conquer en utilisant des fonctions évaluables.

CHAPITRE II.
LA HIERARCHIE DES NŒUDS.

Les informations utiles pour un nœud de l'arbre de recherche appartiennent à trois niveaux différents, le langage de programmation, le programme ou son exécution. Afin d'introduire une distinction entre ces niveaux, une hiérarchie est créée avec les **classes de nœuds**, les **nœuds modèles** et les **nœuds processus**. Les nœuds modèles sont en correspondance biunivoque avec les prédicats du programme. Les nœuds modèles sont des instances des classes de nœuds et les nœuds processus des instances de nœuds modèles.

II. 1. LES CLASSES DE NŒUDS.

Elles sont caractéristiques du PROLOG de LAIOS. Chaque classe est définie comme un automate d'état fini qui interprète le modèle d'exécution en arbre de recherche.

II. 1. 1. Les différentes classes.

Deux classes ont déjà été citées lors de l'exposé du modèle, la classe **OU** et la classe **ET**. Il faut ajouter, pour pouvoir implanter le PROLOG de LAIOS tel qu'il a été défini dans le livre précédent, trois autres classes: la classe **REQ**, la classe **TERM** et la classe **CUT**.

Par définition, un prédicat associé à un nœud modèle **appartient** à la classe de ce nœud modèle. Ainsi:
- le prédicat d'un fait appartient à la classe TERM,
- le prédicat de la tête d'une règle appartient à la classe ET,
- le prédicat vide, tête d'une requête, appartient à la classe REQ,
- chaque prédicat du corps d'une règle ou d'une requête appartient à la classe OU,
- sauf le prédicat de coupure qui appartient à la classe CUT.

Les automates d'état fini qui définissent les classes sont asynchrones les uns par rapport aux autres. Ils communiquent entre eux par messages. Les messages ne peuvent suivre que les liens de parenté suivants: père-fils, mère-père et frère ainé-frère cadet. Les changements d'état se font soit sur réception d'un message, soit après l'accomplissement d'un calcul. Seul le changement d'un mot d'état est une section critique ininterruptible.

II. 1. 2. Les états et les messages.

La liste des messages ci-dessous contient ceux nécessaires au parallélisme OU et au ET pipe line, elle ne contient pas ceux nécessaires à l'implantation de la coupure qui sera détaillée plus loin.

- **kill** : message du père donnant l'ordre au fils de disparaitre;
- **ack** : message du fils au père ou du frère ainé au cadet ou du cadet au frère ainé signalant sa destruction prochaine;
- **end** : message du père au fils lui intimant l'ordre de cesser toute activité;
- **father** : message du père au fils lui demandant d'engendrer ses propres fils;
- **ready** : message de la mère au père lui indiquant que tous ses fils ont été engendrés;
- **goil** : message du père au fils ou du frère ainé au frère cadet lui demandant d'entamer un

calcul de descente dans l'arbre;

- **goi2** : message du fils au père lui transmettant un résultat et lui enjoignant d'entamer un calcul de remontée;
- **goa** : même message que précédemment avec l'ordre de travailler en anticipation;
- **emit** : message du père au fils ou du frère cadet au frère ainé pour lever l'anticipation et autoriser la transmission d'un résultat;
- **resume** : message du fils au père lui signalant la fin de transmission de résultat.

Le mot d'état d'un automate contient quatre champs:

- les drapeaux de réception de messages impératifs **dec** pour **kill et term** pour **end;**
- le numéro de la **phase** qui note la progression, **0** pour la destruction, **1** pour la génération, **2** pour l'attente d'activité de descente, **3** pour l'activité de descente, **4** pour l'attente de résultat, **5** pour la réception de résultat à la remontée et **6** pour l'épuisement de la file d'attente des résultats à transmettre;
- un drapeau d'état interne **file** traduisant l'occupation ou la vacuité de la file d'attente des résultats, deux compteurs: **fc** pour numéroter le fils courant et **nl** pour décompter le nombre d'acquittements nécessaires avant une disparition;
- les drapeaux de communications qui synchronisent les transmissions de données: **act** pour l'enregistrement des signaux d'activité et **ant** pour noter l'ordre de travail en anticipation.

II. 1. 3. Les automates.

La réception des messages est asynchrone avec l'activité du nœud. Elle s'effectue par modification du mot d'état, chaque modification étant une section critique ininterruptible. Par convention, un drapeau est actif à un et inactif à zéro:

message :	état :	
kill :	dec = 1 .	
ack :	nl = nl+1 .	
end :	term = 1 .	
father :	phase = 1 .	
ready :	phase = 2 .	
goi1 :	act = 1 ,	ant = 0 .
goa :	act = 1 ,	ant = 1 .
goi2 :	act = 1 .	
emit :	ant = 0 .	
resume :	phase = 5 .	

Les actions sont définies en fonction du mot d'état et de la classe du nœud. Les abréviations suivantes sont employées: P pour le père, M pour la mère, A pour le frère ainé, C pour le frère cadet, FS pour tous les fils existants, FSn pour tous les fils sauf le nième, F1 pour le fils ainé, Fc pour le fils courant et Fb pour le benjamin. Le message *mess* envoyé au destinataire *dest* est noté *mess :> dest.*

Les actions impératives sont les mêmes pour tous les nœuds:

état initial :	action :	*messages :*	**état final :**
phase=0, nl=0.	désallocation de l'espace occupé par le processus.	*ack :> P.*	/
dec=1, phase>1.	si A ≠ P, si C ≠ P.	*kill :> FS,* *ack :> A,* *ack :> C.*	**phase=0.**
dec=0,term=1, phase>1.	désallocation de l'espace occupé par les données.	*kill :> FS.*	**term=0, phase=1, file=0,act=0,ant=0.**

L'automate de la classe ET s'écrit, lorsque **dec** et **term** sont inactifs:

état initial :	action :	*messages :*	**état final :**
phase=2, act=1.	calcul de descente, si succès: si échec:	*father :> FS.*	**act=0, phase=3, phase=5.**
phase=3.		*goi1 :> F1.*	**phase=4.**
phase=4, act=1.	calcul de remontée, si succès enregistrement dans la file.	*emit :> Fb.*	**act=0, positionne file**
phase=4, ant=0, file=1.	defile le premier résultat.	*goi2 :> P.*	**ant=1, positionne file.**
phase=5.		*end :> FS.*	**phase=6**
phase=6, ant=0, file=1.	défile le premier résultat.	*goi2 :> P.*	**ant=1, positionne file.**
phase=6, ant=0, file=0.		*resume :> P.*	**phase=2.**

L'automate de la classe OU s'écrit:

état initial :	action :	*messages :*	**état final :**
phase=2, act=1.	calcul de descente, si succès: si échec:	*father :> FS.*	**act=0, phase=3, phase=5.**
phase=3.		*goi1 :> F1, goa :> FS1.*	**phase=4.**
phase=4, act=1.	calcul de remontée, si succès enregistrement dans la file.	*emit :> Fc.*	**act=0, positionne file**
phase=4, ant=0, file=1.	defile le premier résultat, si C ≠ P, si C = P.	*goi1 :> C, goi2 :> P.*	**ant=1, positionne file.**
phase=5, fc≠Fb.	fc = fc+1.	*end :> Fc, emit :> Fc+1.*	**phase=4**
phase=5, fc=Fb.		*end :> Fb.*	**phase=6**
phase=6, ant=0, file=1.	défile le premier résultat, si C ≠ P, si C = P.	*goi1 :> C, goi2 :> P.*	**ant=1, positionne file.**
phase=6, ant=0, file=0.	si A ≠ P, si A = P.	*emit :> A, resume :> P.*	**phase=2.**

L'automate de la classe REQ s'écrit:

état initial :	action :	*messages :*	**état final :**
phase=2.	calcul de descente.	*father :> FS.*	**act=0, phase=3.**
phase=3.		*goi1 :> F1.*	**phase=4.**
phase=4, act=1.	calcul de remontée, si succès affichage du résultat.	*emit :> Fb.*	**act=0.**
phase=5.		*kill :> FS.*	**phase=0**

L'automate de la classe TERM s'écrit:

état initial :	action :	*messages :*	**état final :**
phase=2, act=1.	calcul de descente, si succès: si échec:		**act=0, phase=4, phase=6.**
phase=4.	calcul de remontée, si succès enregistrement dans la file.		**phase=6, positionne file.**
phase=6, ant=0, file=1.	défile le premier résultat.	*goi2 :> P.*	**ant=1, positionne file.**
phase=6, ant=0, file=0.		*resume :> P.*	**phase=2.**

II. 1. 4. La souplesse du modèle.

Les définitions de l'ensemble des états, de l'ensemble des messages et des automates permettent une implantation du langage de manière précise et aisément modifiable. Une illustration de cette souplesse est donnée par l'implantation de la coupure qui n'était pas réalisée au paragraphe précédent.

Une classe supplémentaire est créée, la classe CUT dont on écrira l'automate. De plus d'après le modèle, le franchissement de la coupure provoque deux sortes de tailles. Les nœuds précédant la coupure dans le pipe line doivent abandonner tous les résultats y compris ceux de la file d'attente et le nœud OU contrôlant les points de choix doit abandonner ceux situés après la règle ayant la coupure franchie mais pas les résultats qui sont en attente dans la file. Deux nouveaux signaux sont donc créés **prune1** et **prune2**.

Le signal **prune1** a une action assez semblable au signal **term**, il est enregistré dans un nouveau drapeau impératif **abandon.** Le second signal **prune2** a au contraire, une action assez semblable à celle du **resume**, il sera enregistré par le passage dans une nouvelle phase 7, suivi d'une phase 8 analogues aux phases d'épuisements 5 et 6. En résumé, il vient:

message :	état :
prune1 :	abandon=1.
prune2 :	phase=7.

Le tableau des actions impératives s'enrichit de l'abandon:

état initial :	action :	*messages :*	**état final :**
dec=0, term=0, abandon=1.	désallocation de l'espace des données, si A ≠ P, si A = P.	*kill :> FS, prune1 :> A, prune2 :> P.*	**abandon=0, phase=1, file=0, act=0, ant=0.**

L'automate de la classe CUT s'écrit:

état initial :	action :	*messages :*	**état final :**
phase=2, act=1.	mie en file du résultat reçu.		**phase=6, act=0, positionne file.**
phase=6, ant=0, file=1.	défile le résultat, si C ≠ P, si C = P.	*goi1 :> C, goi2 :> P.*	**ant=1, positionne file.**
phase=6, ant=0, file=0.	si A ≠ P, si A = P.	*prune1 :> A, prune2 :> P.*	**phase =2.**

L'automate de la classe ET s'enrichit des phases 7 et 8:

état initial :	action :	*messages :*	**état final :**
phase=7.		*end :> FS.*	**phase=8.**
phase=8, ant=0, file=1.	défile le premier résultat.	*goi2 :> P.*	**ant=1, positionne file.**
phase=8, ant=0, file=0.		*prune2 :> P.*	**phase=2.**

Le nœud REQ s'enrichit des mêmes actions sauf qu'il n'y a pas de message pour le père et que le résultat doit être envoyé à l'affichage (ce qui peut revenir au même si l'on crée un processus d'affichage père d'un nœud REQ).

Le nœud OU n'ayant pas besoin de transmettre le signal **prune2** au-dessus, il n'est pas nécessaire d'utiliser la phase **8**, la phase **6** suffit, il s'ajoute donc:

état initial :	action :	*messages :*	**état final :**
phase=7.	fc=Fb.	*end :> FS.*	**phase=6.**

Pour clore ce paragraphe, remarquons qu'il serait intéressant de disposer d'un programme réalisant automatiquement la transposition d'un langage en classes de nœuds; un tel programme pourrait être appelé méta-compilateur.

II. 2. LES NŒUDS MODELES.

Ils forment la base de connaissances de LAIOS. Ils sont obtenus par compilation des programmes, chaque prédicat est en correspondance biunivoque avec un nœud modèle. Chaque nœud modèle est une instance de sa classe de nœuds.

II. 2. 1. Le descripteur de nœud-modèle.

Chaque nœud modèle est rangé dans la base de connaissances sous la forme d'un descripteur qui contient toutes les informations utiles. Ce descripteur est lui-même un objet de LAIOS.

A chaque nœud modèle est associé un **numéro d'ordre**. Ce numéro est utilisé par le gestionnaire de la base soit pour utiliser le modèle lors du déroulement du programme, soit pour modifier le noeud modèle lors de modifications sur le programme utilisateur.

Chaque descripteur est un objet de LAIOS, il commence donc par une cellule objet système de type descripteur de nœud-modèle, d'arité comprise entre 2 (nœud de classe CUT) et 6 (nœud de classe OU) et de longueur variable selon le nœud modèle.

La première information est la classe à laquelle appartient le nœud; elle peut être codée sur un entier court.

La seconde information est relative aux variables utilisées dans la clause dont est extrait ce nœud modèle. Le compilateur numérote ces variables. Il distingue les **variables externes** qui apparaissent dans plusieurs prédicats de la clause, les **variables internes** qui n'apparaissent que dans un seul prédicat et les **résultats** de fonctions. Sont numérotés consécutivement dans l'ordre croissant: les variables externes, les variables internes de la tête de clause, les résultats de

la tête de clause, puis les variables internes et les résultats de chaque prédicat du corps de la clause.

Les noeuds de la classe TERM n'ont besoin que d'une information, le nombre de variables; ceux de classe ET ont besoin de deux informations, le nombre de variables externes et le nombre de variables internes; ceux de la classe OU ont besoin de trois informations, le nombre de variables externes, le nombre de variables internes et le numéro de la première variable interne; enfin, les noeuds de classe REQ ou CUT n'ont besoin que du nombre de variables externes.

La troisième information est la **paternité formelle** du nœud. Elle ne concerne que les nœuds de classe REQ, ET et OU qui ont une liste de fils. Chaque fils est représenté par son numéro d'ordre dans la base. La liste des fils est un tableau, il est initialisé par un descripteur de tableau dont l'arité est égale au nombre de fils.

La dernière information apparaissant dans le descripteur est le **bloc des paramètres** du nœud. Cette dernière information n'apparait pas dans les nœuds de classe CUT qui n'effectuent comme travail sur les données que de transmettre la première qui arrive et non les autres.

En résumé, selon leur classe, les nœuds ont des descripteurs sur les modèles suivants:

- modèle de nœud REQ:

en-tête de modèle	4	<longueur>
caract	0	r '
entier	0	<nombre de variables externes>
tableau	<nombre de fils>	<longueur>
...	...	...
bloc	<nombre de paramètres>	<longueur>
...	...	...

- modèle de nœud TERM:

en-tête de modèle	3	<longueur>
caractère	0	t '
entier	0	<nombre de variables>
bloc	<nombre de paramètres>	<longueur>
...	...	...

- modèle de nœud ET:

en-tête de modèle	5	<longueur>
caractère	0	e '
entier	0	<nombre de variables externes>
entier	0	<nombre de variables internes>
tableau	<nombre de fils>	<longueur>
...	...	...
bloc	<nombre de paramètres>	<longueur>
...	...	...

- modèle de nœud OU:

en-tête de modèle	6	<longueur>
caractère	0	o '
entier	0	<nombre de variables externes>
entier	0	<nombre de variables internes>
entier	0	<n° de la 1° variable interne>
tableau	<nombre de fils>	<longueur>
...	...	...
bloc	<nombre de paramètres>	<longueur>
...	...	...

- modèle de nœud CUT:

en-tête de modèle	2	<longueur>
caract	0	r '
entier	0	<nombre de variables externes>

II. 2. 2. Compilation de programmes.

Le programme de calcul de la suite de Fibonnacci :

```
"clause 1"
fib (0, 1).
"clause 2"
fib (1, 1).
"clause 3"
fib (N :≥2, + (X, Y)) :- fib (- (N, 1), X), fib (- (N, 2), Y).
"but"
:- fib (2, X).
```

se compile en sept nœuds modèles:

- nœud 1 (clause 1):

en-tête de modèle	3	6
caractère	0	t '
entier	0	0
bloc	2	3
entier	0	0
entier	0	1

- nœud 2 (clause 2):

en-tête de modèle	3	6
caractère	0	t '
entier	0	0
bloc	2	3
entier	0	1
entier	0	1

Commentaire: le dernier champ de la cellule en-tête de modèle donne la longueur en nombre de cellules du modèle.

- nœud 3 (tête de la clause 3):

en-tête de modèle	5	15
caractère	0	e '
entier	0	3
entier	0	1
tableau	2	3
entier	0	4
entier	0	5
bloc	2	8
variable liée	1	10-1
contrainte	1	≥
entier	0	2
fonction évaluable	3	+
variable liée	0	10-2
variable liée	0	10-3
résultat	0	10-4

Commentaire: ce prédicat utilise les trois variables N, X et Y numérotées respectivement 10-1, 10-2 et 10-3, 10 étant le numéro de la page brouillon qui contient les tableaux de variables, et un résultat interne numéroté 10-4.

- nœud 4 (premier prédicat du corps de la clause 3):

en-tête de modèle	6	15
caractère	0	o '
entier	0	3
entier	0	1
entier	0	5
tableau	3	4
entier	0	1
entier	0	2
entier	0	3
bloc	2	6
fonction évaluable	3	-
variable liée	0	10-1
entier	0	1
résultat	0	10-5
variable liée	0	10-2

Commentaire: ce prédicat n'utilise que deux variables externes N et X (10-1 et 10-2) mais le nœud devra transmettre les trois (le nombre de variables externes est le même pour tous les nœuds d'une même clause). La variable interne est le résultat de la soustraction noté 10-5.

- nœud 5 (deuxième prédicat de la cause 3):

en-tête de modèle	6	15
caractère	0	o '
entier	0	3
entier	0	1
entier	0	6
tableau	3	4
entier	0	1
entier	0	2
entier	0	3
bloc	2	6
fonction évaluable	3	-
variable liée	0	10-1
entier	0	2
résultat	0	10-6
variable liée	0	10-3

- nœud 6 (prédicat vide tête de la clause 4):

en-tête de modèle	4	7
caractère	0	r '
entier	0	1
tableau	1	2
entier	0	7
bloc	1	2
variable liée	0	10-1

Commentaire: ce nœud sert à communiquer à l'extérieur la variable X numérotée 10-1.

- nœud 7 (corps de la clause 4):

en-tête de modèle	6	12
caractère	0	o
entier	0	1
entier	0	0
entier	0	2
tableau	3	4
entier	0	1
entier	0	2
entier	0	3
bloc	2	3
entier	0	3
variable liée	0	10-1

Commentaire: ce nœud n'a pas de variable interne, aussi le numéro de la première variable interne qui est calculée par un compteur n'a pas de sens.

Le compilateur réalise trois tâches différentes. Il opère une traduction du PROLOG de LAIOS en format interne. Il fabrique les liens entre nœuds en numérotant les nœuds et en reportant ces numéros dans la table des fils. Cette deuxième activité assure la cohérence de la base, elle est mise en jeu à chaque modification de la base. Enfin, il analyse la nature des variables, les numérote pour chaque clause et évalue, lorsqu'il y a lieu de le faire le nombre de variables externes, de variables internes et le numéro de la première variable interne pour chaque nœud.

II. 2. 3. Utilisation des nœuds modèles.

Comme il est apparu dans l'arbre de recherche, une même clause est utilisée plusieurs fois lors de l'exécution d'un programme. En conséquence, plusieurs nœuds processus se partageront le même nœud modèle.

Le nœud modèle ne contient que des informations non modifiées au cours de l'exécution, il peut donc être rangé dans la base avec l'accès en lecture seulement pour tous les processus autres que le gestionnaire de la base de connaissances. Des copies du nœud peuvent donc être distribuées aux différents processeurs, à la demande, sans problème de maintien de cohérence.

Il est évident que l'utilisation de prédicats modifiant la base de clauses durant l'exécution (comme assert ou retract) est interdite dans LAIOS. De tels prédicats exigeraient la modification des noeuds modèles durant l'exécution et en interdiraient les copies multiples.

Les nœuds modèles sont utilisés lors de la création de nœuds processus, lors d'un travail sur les données qui utilise le bloc paramètre comme référence et lors de la destruction des fils pour remplacer leur adresse réelle par leur numéro d'ordre. Cette dernière utilisation n'est pas indispensable, elle résulte d'un choix sur les descripteurs de nœuds processus.

II. 3. LES NŒUDS PROCESSUS.

Ce sont les véritables nœuds de l'arbre de recherche. Ils sont créés et détruits dynamiquement. Ce sont des instances de nœuds modèles.

II. 3. 1. Les descripteurs de nœuds processus.

Ce sont des objets de LAIOS. Il sont initialisés par une cellule système de type descripteur de nœud processus. Son arité dépend de la classe du nœud, sa longueur du nœud lui-même. Les informations suivantes sont rangées dans ce descripteur.

La première information est la **classe** du nœud, c'est la même que celle de son modèle.

La seconde information est le **numéro d'ordre du nœud modèle**. Cette information permet de rappeler le modèle à chaque utilisation. Celà autorise à ranger les nœuds modèles dans des mémoires caches, le numéro d'ordre servant d'adresse virtuelle.

La troisième information est l'**état du nœud**. Dans le simulateur, cet état a été condensé en un seul entier, mais il est imaginable de lui donner plusieurs champs pour plus de souplesse.

Les informations suivantes sont les **adresses réelles des parents**. En premier, l'adresse du père, c'est une adresse totale avec le numéro du processeur, le numéro de la page et le numéro de la ligne dans la page. Suivent, pour les nœuds de classe OU et CUT, les adresses, totales également, des frères ainé et cadet. Une adresse totale est une suite de deux cellules de LAIOS, un entier court pour le numéro du processeur et une variable liée pour l'adresse locale.

Les informations suivantes sont relatives à l'**emplacement de données**, ce sont des adresses locales (variables liées). Apparaissent successivement l'adresse des données en descente, l'adresse des données internes, l'adresse des données à la remontée, l'adresse du début de la file d'attente des données en sortie à la remontée et l'adresse de la fin de cette file d'attente. Toutes ces adresses pourront être remplacées par la constante LAIOS variable libre (nil) lorsqu'elles ne pointent sur rien.

Les informations sur les fils closent ce descripteur. Comme ce sont des informations de longueur variable, elles ont été reléguées en fin de descripteur pour permettre d'accéder aux précédentes par déplacement (offset) constant. Apparaissent successivement le numéro du fils courant, le compteur du nombre de parents utilisés pour garantir la destruction sans incohérence, puis la **table des adresses des fils**. Dans le modèle adopté actuellement, ce champ contient selon le cas l'adresse physique du fils lorsqu'il est implanté ou son numéro d'ordre lorsqu'il ne l'est pas. Un choix différent pourrait être fait qui conserverait toujours le numéro d'ordre et éviterait d'utiliser le modèle du nœud lors de la disparition des fils.

II. 3. 2. La gestion des nœuds processus.

Quelques processus systèmes résidant dans chaque processeur prennent en charge la gestion des nœuds processus.

Un processus **contremaître** (scheduler) dirige l'activité de tous les autres processus.
Dans sa version la plus rustique, il possède une file d'attente des processus activables. Il donne l'activité au processus de tête jusqu'à ce que ce dernier ait terminé son travail et lui ait rendu la main. Il active alors le suivant dans la file.
Pour obtenir un contremaître efficace, des améliorations doivent être apportées à ce principe. En particulier, lorsqu'un nœud processus est en tête de file, le contremaître doit d'abord

chercher à savoir si l'activité de ce nœud nécessite ou non la présence de son nœud modèle. Dans l'hypothèse où cette présence est nécessaire, il doit contrôler la présence physique de ce modèle avant d'activer le nœud processus. Si l'absence du modèle est constatée, le contremaître doit demander au gestionnaire du cache le chargement de la page contenant ce modèle et suspendre le nœud processus. Cette suspension peut se faire soit par relégation en fin de file d'attente, soit par mise dans une autre file d'attente des processus suspendus, file prioritaire sur la précédente.

Le processus **gestionnaire de la mémoire cache** travaille indépendament des autres. Il est capable de répondre affirmativement ou négativement à la demande de la présence physique d'un nœud modèle dans le cache. Dans le cas d'une réponse positive, il est capable de maintenir cette présence durant l'utilisation. Dans le cas d'une réponse négative, il est capable de mettre en route le processus de chargement de la page le contenant. Un algorithme classique est utilisé pour effacer au fur et à mesure des chargements les pages les moins utiles.

Le processus de **calcul de la charge** a pour mission d'essayer de retarder le plus possible une saturation du processeur et son blocage. Il dispose pour celà d'informations sur le nombre de processus implantés, le nombre de processus activables, le nombre de processus générateurs et le taux d'occupation de chaque page mémoire. Il intègre toutes ces données en un nombre appelé **charge du processeur**. Il est capable de mettre à jour cette charge constamment et de la transmettre à la demande.

La fonction de calcul de cette charge selon les données est une heuristique qui devra faire l'objet d'une mise au point particulière.

Enfin, le processus **mère** contient une file d'attente des processus générateurs. Il est capable à tout moment d'enregistrer une arrivée dans la file ou de donner le numéro du premier fils non encore créé du nœud processus tête de file.

Ce processus est activé périodiquement par le contremaître. La périodicité d'activation reste une heuristique à déterminer en fonction de la charge. Lorsque le processus mère est activé, il élit le plus chargé parmi les processeurs voisins du sien et le sien propre. Il effectue alors l'implantation du premier fils non encore créé du nœud processus tête de file des processus générateurs chez le processeur élu.

CHAPITRE III.
LE TRAITEMENT DES DONNEES.

Les chapitres ci-dessus décrivent le contrôle du déroulement. Il reste à préciser les opérations effectuées sur les données dans chaque noeud de l'arbre de recherche.

III. 1. TRANSMISSION DES DONNEES.

La portée des variables étant réduite à la règle, deux sortes de transmission entre nœuds seront effectuées selon que les nœuds représentent des prédicats appartenant à la même règle ou non.

III. 1. 1. Tableau de variables.

Lors de la compilation d'une règle, chaque variable apparaissant dans la règle a reçu un numéro correspondant à son nom. Les variables partagées par au moins deux prédicats ont été déclarées variables externes pour les prédicats de la règle. Le nombre de variables externes de la règle est connu de chaque prédicat de la règle.

L'utilisation d'une règle se traduit dans l'arbre de recherche par l'existence d'une chaîne circulaire de nœuds processus débutant par un nœud de type ET, le père, et reliant successivement tous les fils, nœuds de type OU, selon des liens frère-ainé-frère-cadet, pour se terminer par un retour sur le père.

Lorsqu'un des nœuds est activé, il lui est nécessaire de connaitre les assignations effectuées sur les variables par les nœuds le précédant. Pour celà, chaque transmission sur cette chaîne aura la forme d'un **tableau de variables externes**. Dans un tel tableau, chaque variable est soit une variable libre, soit une variable liée à une cellule (éventuellement une autre variable), soit une constante (assignation de la variable à une valeur).

Les assignations multiples se traduiront par des tableaux distincts; ils seront soit séparés dans le temps (pipe line), soit stockés en file d'attente au niveau d'un nœud (synchronisation).

III. 1. 2. Le bloc d'arguments.

Par contre, lorsque l'on recherche l'effacement d'un but par l'utilisation d'un ensemble de règles candidates, il n'est plus possible d'utiliser les variables par leur nom.
Ce problème est assez semblable à l'appel d'une procédure dans un langage impératif. Les paramètres effectifs de l'appel sont substitués aux paramètres formels de la procédure, chaque paramètre étant repéré par sa position et non plus par son nom.

De même, un nœud OU désirant activer ses fils, nœuds ET ou TERM, leur transmettra un bloc de cellules LAIOS. Ce bloc aura été fabriqué par le nœud OU à partir de son bloc paramètres. Il sera comparé par chaque nœud fils à son propre bloc paramètres. Ainsi les assignations des variables seront transmises par position dans le bloc. Ce bloc transmis est appelé **bloc argument**.

Le même problème se pose à la remontée, aucune liaison n'ayant été créée entre les variables du nœud "appelant" et celles du nœud "appelé", il faudra avoir recours à la même solution et

c'est encore des blocs arguments qui seront transmis.

III. 1. 3. L'opérateur universel.

Le modèle exécutif d'un nœud va donc être un opérateur paramétré par le bloc paramètre du nœud. Cet opérateur peut être traversé dans les deux sens, à la descente et à la remontée. Une face de cet opérateur sera une entrée-sortie acceptant un bloc argument, l'autre face, une entrée-sortie acceptant un tableau de variables.

La comparaison du bloc argument avec le bloc paramètre qui permet de déduire le tableau de variables est appelé **unification**, elle correspond effectivement à la recherche du plus petit unificateur commun des deux blocs. Le calcul qui consiste à substituer dans le bloc paramètre les variables par leur valeur et à en déduire le bloc argument est appelé **évaluation**, c'est une opération très voisine de l'évaluation d'une liste en LISP [DEVI_85].

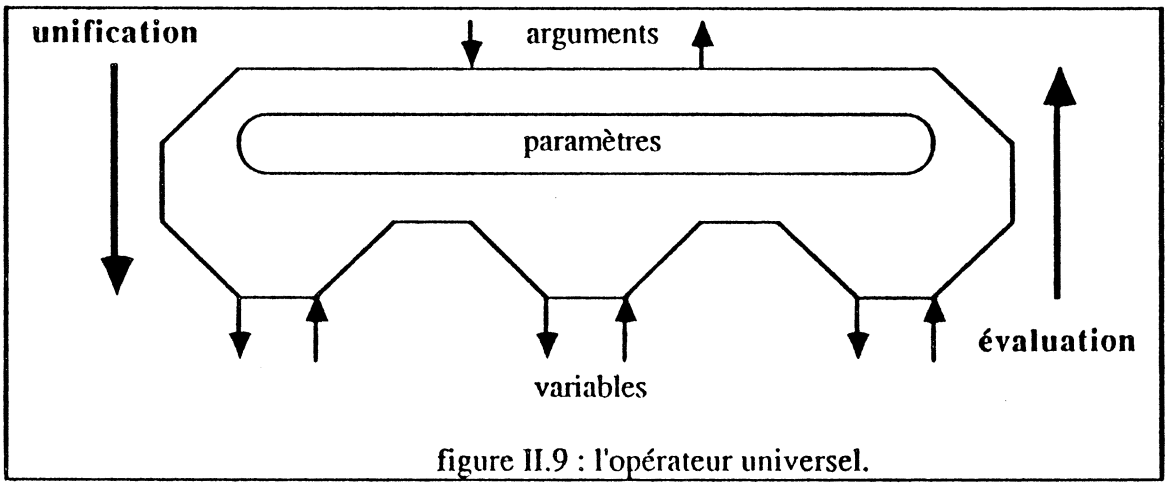


figure II.9 : l'opérateur universel.

III. 1. 4. Arbre d'exécution.

En remplaçant dans l'arbre de recherche, chaque nœud par son opérateur universel convenablement orienté, on obtient l'**arbre d'exécution** sur les données.

Le nombre d'informations qui apparaissent sur un tel arbre est vite considérable. Le premier exemple choisi est donc extrèmement simple; il s'agit du calcul de la longueur d'une suite.

```
"clause 1"
long ( [], 0).
"clause 2"
long ( (X, Y), + (N, 1)) :- long ( Y, N ).
"clause 3"
:- long ( ( a, ( b, () ) ) ).
```

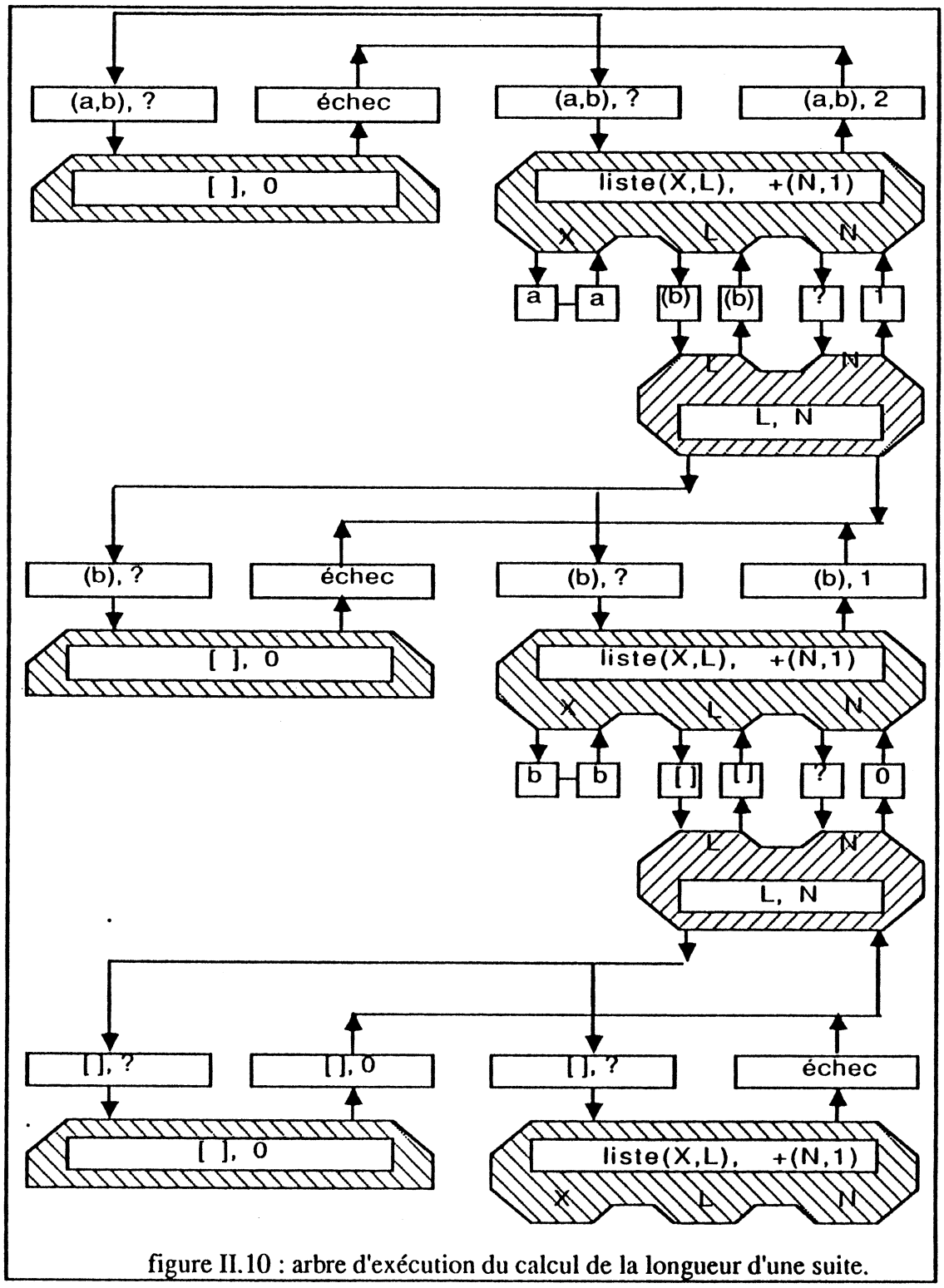


figure II.10 : arbre d'exécution du calcul de la longueur d'une suite.

Le second exemple choisi n'est qu'une partie de l'arbre d'exécution du calcul de la suite de Fibonnacci pour mettre en évidence le cycle parcouru par un tableau de variables.

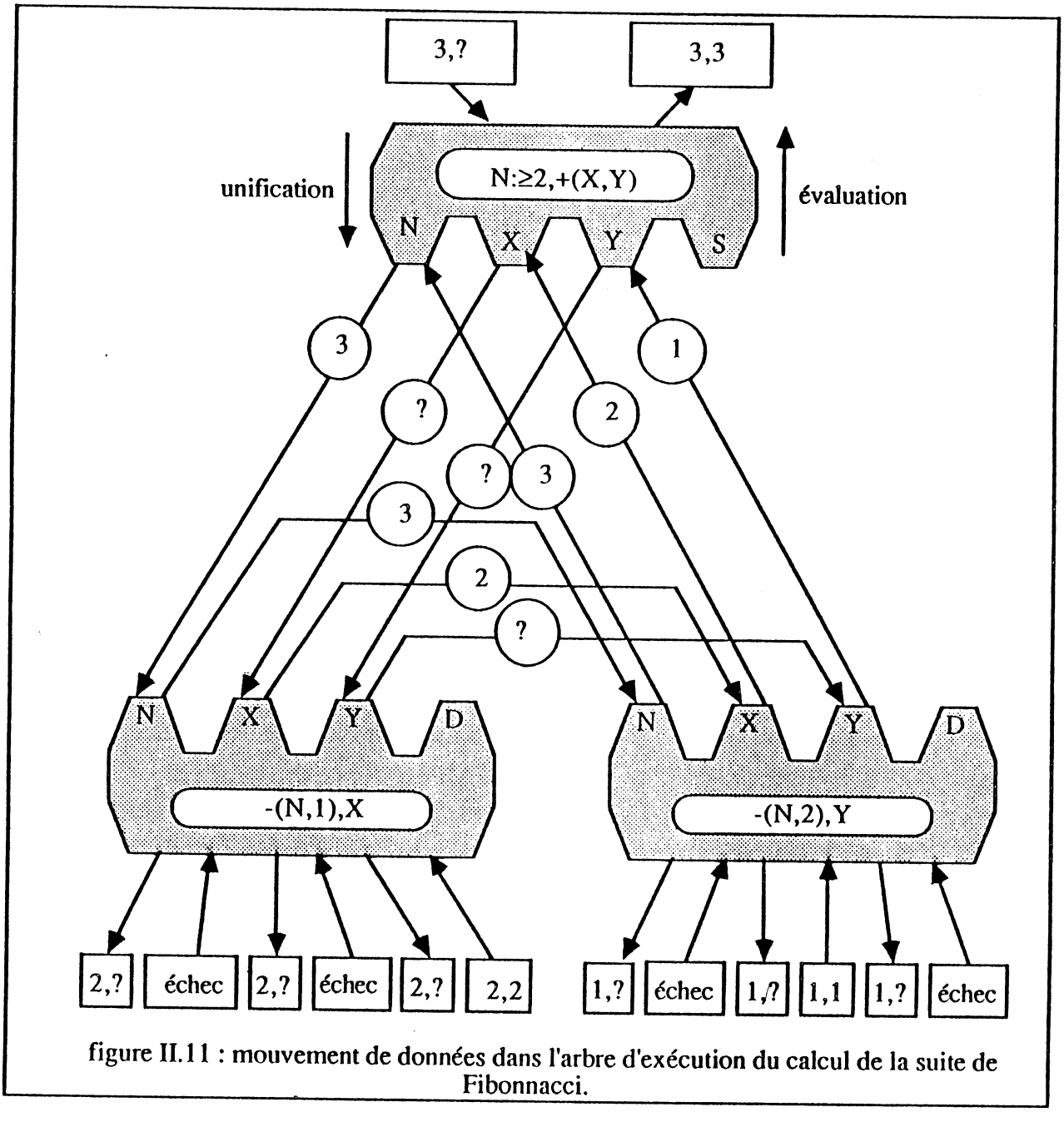


figure II.11 : mouvement de données dans l'arbre d'exécution du calcul de la suite de Fibonnacci.

III. 2. L'UNIFICATION.

Dans les deux paragraphes suivants sont présentés les algorithmes d'unification et d'évaluation; une convention d'écriture est adoptée: en caractères standard les instructions et en caractères italiques les commentaires.

L'unification consiste à comparer le bloc arguments reçu au bloc paramètres constitutif du nœud. Les nœuds de type ET ou TERM la font à la descente, les nœuds de type OU à la remontée. Le résultat de cette opération est soit un échec, soit une assignation des variables.

III. 2. 1. Situation initiale.

L'algorithme d'unification utilise les objets suivants:

- *un pointeur adresse paramètre* **PP,**
- *un pointeur adresse argument :* **PA,**
- *une valeur du paramètre courant :* **VP,**
- *une valeur de l'argument courant :* **VA,**
- *un tableau des variables :* **TV,**
- *un tableau des inconnues :* **TI,**
- *un pointeur du haut de la pile paramètre :* **SP,**
- *un pointeur du haut de la pile argument :* **SA,**
- *un compteur paramètre :* **CP,**
- *un compteur argument :* **CA,**
- *un compteur auxiliaire :* **AUX.**

La situation initiale est la suivante:

- PP pointe sur le début du bloc paramètre du modèle du nœud;
- VP est égal à la cellule descripteur du bloc;
- CP = 0; SP = nil; *la pile est vide;*
- PA pointe sur le début du bloc argument reçu en entrée;
- VA est égal à la cellule descripteur du bloc;
- CA = 0; SA = nil;
- si le nœud est de classe ET ou TERM, toutes les cases de TV et TI sont initialisées à variable libre;
- si le nœud est de classe OU, TV reprend la valeur qu'il avait en fin d'évaluation *(il s'agit d'enrichir les assignations de variables reçues en entrée)*, toutes les cases de TI sont initialisées à variables libres.

III. 2. 2. Parcours des blocs.

L'unification s'effectue en un parcours parallèle des deux blocs.

REPETER

mise à jour des compteurs;
CP := CP + arité [VP];
CA := CA + arité [VA];

traitement particulier des fonctions évaluables qui se comportent commme leur résultat;
SI etiquette [VP] = fonction évaluable
ALORS sauter au résultat (PP, VP, CP, AUX);
ce parcours modifie PP, VP et CP et nécessite un compteur auxiliaire, les paramètres de la fonction sont parcourus sans être analysés, à la fin du parcours PP pointe sur le résultat de la fonction et VP contient cette cellule résultat;
SI etiquette [VA] = fonction évaluable
ALORS sauter au résultat (PA, VA, CA, AUX);

déréférenciation des variables liées;
TANT QUE étiquette [VP] = variable liée ou résultat
FAIRE
SI arité [VP] > 0
ALORS empiler (PP, arité [VP]) dans SP;

si la variable est sujette à des contraintes, on note dans la pile l'adresse de cette variable et le nombre de contraintes;
déréférencer (PP, VP, CP, SP);
FIN FAIRE;

la déréférenciation consiste à exécuter les instructions suivantes:
- SI CP > 0 ALORS empiler (PP+1, CP-1) dans SP;
c'est l'adresse de retour;
- PP := valeur [VP];
c'est la nouvelle adresse;
- VP := mémoire [PP];
c'est la nouvelle valeur; par une fonction logique sur les étiquettes, les contraintes de types sont conservées dans VP;
- CP := arité [VP];
nouvelle valeur du compteur pour pouvoir parcourir le nouveau bloc;

même déréférenciation sur les arguments;
TANT QUE étiquette [VA] = variable liée ou résultat
FAIRE
SI arité [VA] > 0
ALORS empiler (PA, arité [VA]);
déréférencer (PA, VA, CA, SA);
FINFAIRE;

désormais VP et VA ne peuvent être que des variables libres, des constantes ou des identificateurs formels;
comparaison (VP, PP, VA, PA);
cette fonction clé est détaillée dans le paragraphe suivant;

passage au paramètre suivant
avancer (PP, VP, CP, SP);
cette fonction procède ainsi:
- SI CP > 0 ALORS PP := PP+1; CP := CP-1;
c'est la poursuite en séquence dans le bloc;
- SI CP = 0 ALORS PP := valeur [SP]; CP := arité [SP]; SP := SP-1;
c'est le retour au bloc précédent par dépilement du point de retour;
- VP := mémoire [PP];
chargement de la nouvelle valeur;

traitement particulier des contraintes;
TANT QUE étiquette [VP] = contrainte
FAIRE
verifier (VP, SP);
la vérification de la contrainte inscrite dans VP sur la variable dont l'adresse est dans SP peut réussir ou échouer; cette vérification peut aussi entrainer la modification de VP, PP et CP si la contrainte utilise des paramètres stockés dans le bloc;
SI arité [SP] > 0
ALORS arité [SP] := arité [SP] -1
il reste des contraintes à vérifier
SINON SP := SP-1;
dépiler l'adresse de la variable sous contrainte;
avancer (VP, PP, CP, SP);
FIN FAIRE;

même traitement pour les arguments:
avancer (VA, PA, CA, SA);
TANT QUE étiquette [VA] = contrainte
FAIRE
vérifier (VA, SA);
SI arité [SA] > 0
ALORS arité [SA] := arité [SA] -1
SINON SA := SA -1;
avancer (VA, PA, CA, SA);
FIN FAIRE;

fin de la boucle de parcours des blocs
JUSQU'A échec OU fin paramètre OU fin argument;
fin paramètre est décelé par une tentative de dépiler SP lorsque cette pile est vide, de même pour fin argument avec la pile SA.

III. 2. 3. Comparaison des objets.

La fonction de comparaison est essentiellement une étude de cas sur les étiquettes:

CAS SUIVANT (étiquette [VP], étiquette [VA]):

(variable libre, variable libre):
SI PP > PA ALORS lier (VP, VA);
SI PA > PP ALORS lier (VA, VP);
si les deux variables libres sont différentes, il faut lier la moins importante à la plus importante, comme leurs adresses sont décroissantes suivant l'ordre d'importance, le choix du sens de liaison s'effectue sur les adresses. S'il s'agit de la même variable, la comparaison réussit sans qu'il n'y ait rien à faire. La fonction lier s'assure de la légalité du lien par comparaison des contraintes de types sur les étiquettes;

(variable libre, constante):
lier (VP, PA);
parcourir (VA, PA, CA , AUX);
même remarque que précédemment sur la fonction lier qui vérifie les contraintes de types; si la constante est longue, il est nécessaire de la parcourir;

(variable libre, identificateur formel utilisateur):
lier (VP, PA);
parcourir (VA, PA, CA, AUX);
même remarque sur la fonction lier; de plus, il est nécessaire de parcourir tous les paramètres de la fonction formelle sans les analyser;

(constante, variable libre):
lier (VA, PP);
parcourir (VP, PP, CP, AUX);

(identificateur formel utilisateur, variable libre):
lier (VA, PP);
parcourir (VP, PP, CP, AUX);

(constante, constante):
SI valeur [VP] ≠ valeur [VA] ALORS échec;
cette comparaison s'effectue sur toute la longueur de la constante; un comparateur

intelligent doit être capable de déceler l'égalité de nombres sous des formats différents; pour les constantes longues, VP, PP, CP, VA, PA et CA sont modifiés;

(identificateur formel utilisateur, identificateur formel utilisateur):
SI arité [VP] ≠ arité [VA] OU valeur [VP] ≠ valeur [VA] ALORS échec;
du fait de la structuration en blocs, la comparaison sur les paramètres s'effectuera en séquence dans la boucle, il suffit donc de s'assurer de l'identité des fonctions formelles;

(identificateur formel système, identificateur formel système):
SI étiquette [VP] ≠ étiquette [VA] OU arité [VP] ≠ arité [VA] ALORS échec;
le champ valeur de ces cellules sert à noter la taille mémoire des objets, ce champ peut être différent sur le paramètre et sur l'argument;

autre cas : échec;
FIN CAS.

III. 2. 4. Fabrication du tableau de variables.

Lorsque l'unification réussit, le parcours se termine et les seules opérations effectuées durant ce parcours ont été des liaisons entre des variables libres et des objets.

Le tableau de variables à transmettre sera fabriqué à partir de la table des variables externes de la manière suivante:

Si la variable est libre, elle est recopiée telle quelle.
Si la variable est liée, il faut déréférencer.
Si après déréférenciation, on aboutit à une variable libre, il faut recopier une variable liée à cette variable libre. Du fait du choix de fabrication des liens, il s'agit d'une autre variable externe de numéro inférieur. La transmission de liens créés dynamiquement entre variables est ainsi assurée.
Si la déréférenciation aboutit à une constante courte, la constante courte est recopiée. Cette substitution permet à la fois de gagner de la place et du temps en parcours de chaînage.
Si la déréférenciation aboutit à une constante longue alors il faut faire une copie de cette constante et mettre dans le tableau une variable liée qui pointe dessus.
Enfin, si la déréférenciation aboutit à une fonction formelle, alors il faut faire un bloc de la fonction formelle et mettre dans le tableau une variable liée qui pointe dessus. La fabrication du bloc demande le parcours des paramètres de la fonction auxquels le traitement suivant est appliqué:
- *les constantes et les fonctions formelles sont recopiées sans modification;*
- *les fonctions évaluables sont remplacées par leur résultat;*
- *les résultats et les variables liées sont déréférencés;*
- *pour les variables libres, il est fabriqué une variable liée qui pointe dessus;*
- *enfin, les contraintes sont ignorées.*

Les nœuds de classe ET doivent mémoriser le tableau de leurs variables internes, il procède de même que pour le tableau des variables externes.

La fabrication des tableaux utilise les mêmes structures, piles, compteurs..., et les mêmes primitives que l'unification et l'évaluation.

III. 3. EVALUATION.

L'évaluation consiste à utiliser un tableau de variables reçu pour calculer un nouveau bloc argument avec le bloc paramètres. Cette opération peut également échouer, soit sur des opérations impossibles (division par zéro par exemple) soit sur des contraintes posées sur les résultats et non satisfaites, soit enfin parce qu'un paramètre d'une fonction évaluable n'est pas connu. Les nœud de type OU réalisent l'évaluation à la descente et les nœuds de type ET ou TERM à la remontée.

III. 3. 1. Situation initiale.

L'algorithme d'évaluation utilise les objets suivants:

- *un pointeur adresse de bloc* **PB**;
- *une valeur de bloc* **VB**;
- *le tableau des variables* **TV**;
- *une pile opérative* **SO**;
- *une pile de parcours* **SP**;
- *un compteur de parcours* **CP**;
- *un compteur de renomination* **CN**;
- *un compteur auxiliaire* **AUX**;
- *une boite à opération avec son accumulateur* **ACC**.

La situation initiale est:

- PB pointe sur le début du bloc paramètre du nœud modèle;
- VB contient la cellule descripteur du bloc (première cellule);
- TV contient le tableau des variables externes reçu en entrée pour les nœud de classe ET et OU, pour les nœud de classe ET, le tableau des variables internes mémorisés, pour les nœud de classe TERM le tableau fabriqué lors de l'unification *car les deux opérations s'effectuent alors en séquence*. Les variables non initialisées sont des variables libres.
- SO et SP sont vides, CP = 0, CN = début du tableau des inconnues.

III. 3. 2. Fabrication du bloc.

Le bloc des paramètres est parcouru. Pour chaque cellule rencontrée, le traitement diffère selon la nature de la cellule.

```
REPETER

    CP := CP + arité [VB];

    CAS SELON (étiquette [VB]):

        (variable libre):
            SI val [VB] = 0
            ALORS val [VB] := CN; CN := CN+1;
            première rencontre de cette variable à qui est attribué un nom d'inconnue;
            empiler (variable liée de numéro val [VB]) dans SO;
            mise de l'inconnue dans le bloc;

        (variable liée):
            SI arité [VB] > 0
```

ALORS empiler (variable sous contrainte, arité [VB], PB) dans SP;
pour pouvoir vérifier les contraintes, il est procédé de la même façon que pour l'unification;
déréférencer (PB, VB, SP, CP);

(constante courte):
empiler (VB) dans SO;

(constante longue):
AUX := arité [VB];
empiler (VB) dans SO;
REPETER
avancer (PB, VB, SP, CP);
en fait SP n'est pas modifié car CP ≥ AUX;
empiler (VB) dans SO;
JUSQU'A AUX = 0;
réalise la copie de la constante longue dans la pile opérative;

(identificateur formel):
empiler (VB) dans SO;

(identificateur fonction évaluable):
empiler (SO) dans (SP);
empiler (VB) dans (SP);
la fonction est mémorisée dans la pile de parcours avec l'adresse actuelle de pile opérative; elle réapparaitra au moment où le parcours parviendra au résultat qui lui correspond; à ce moment , les paramètres de la fonction seront empilés dans la pile opérative à partir de l'adresse mémorisée;

(résultat de fonction):
f := dépiler (SP);
on obtient la fonction à calculer;
SO := dépiler (SP);
on obtient l'adresse du premier opérande dans la pile opérative qui devient le nouveau pointeur de la pile SO;
calculer (f, SO, ACC);
SO n'est pas modifié et le résultat de l'opération est dans ACC;
SI arité [VB] > 0
ALORS empiler (variable soumise à contrainte) dans SP;
afin de pouvoir vérifier les contraintes posées sur le résultat ;
déréférencer (VB, PB, SP, CP);
afin de retrouver la variable du tableau liée au résultat;
SI étiquette [VB] = variable libre
ALORS lier (VB, SO);
mémorise l'emplacement du résultat dans la pile opérative;
SI étiquette [VB] ≠ variable libre ET VB ≠ ACC
ALORS échec;
si le résultat avait été lié précédement, il faut vérifier que l'objet auquel ce résultat avait été lié est bien celui obtenu par le calcul; ce genre de vérification à posteriori est utilisé par les nœuds de classe ET qui unifient avant d'évaluer, l'unification identifiant une fonction évaluable à sa variable résultat;
SI non échec
ALORS empiler (ACC) dans SO;
le résultat est mis en place, il peut occuper plusieurs mots mémoire de la

pile;

(identificateur de contraintes):
verifier (VB, SP);
SI arité [SP] > 0
ALORS arité [SP] := arité [SP] -1
SINON SP := SP -1;
même traitement des contraintes que lors de l'unification;

FIN CAS;

avancer (VB, PB, SP, CP);
JUSQU'A échec OU fin de parcours.

III. 3. 3. Fin de l'algorithme.

Dans la mesure où le parcours se termine sans échec, le bloc à transmettre se trouve dans la pile opérative. Une cellule descripteur de bloc indiquant la longueur exacte du bloc dans son champ valeur est placée en tête du bloc.Selon la classe du nœud, il peut être nécessaire de faire des copies multiples de ce bloc.

De plus, les nœuds de classe OU doivent aussi mémoriser leurs résultats sous forme d'un tableau de résultats pour les unifications à venir. Ce tableau est fabriqué de la même façon que le tableau de variables dans l'unification.

III. 4. CONCLUSION DE CE CHAPITRE.

Le matériel demandé par ces algorithmes se résume à:

- quelques registres adresses, quelques registres données et quelques compteurs;
- une boite à opérations capable de réaliser les opérations courantes et les vérifications de contraintes sur des données de format variable;
- une zone mémoire privilégiée contenant deux tableaux et deux piles; ce sera le rôle de la page appelée brouillon.

Le logiciel se résume à quelques primitives de base: empiler, dépiler, déréférencer, avancer, lier, parcourir, comparer les étiquettes ou les arités, calculer, vérifier et copier.

Tout ceci sera examiné en détail dans les livres suivants sur l'architecture et sur la simulation.

LIVRE TROISIEME :

L'ARCHITECTURE .

CHAPITRE I.
LE RESEAU.

Dès que l'on s'affranchit de l'architecture de Von Neumann, les possibilités de combiner processeurs et mémoires en réseau deviennent immenses. Almasi [ALMA_85] a recensé cinquante cinq architectures différentes à haut degré de parallélisme! La difficulté consiste à choisir les paramètres permettant d'obtenir un réseau cohérent et, même s'il n'est pas tous services (general purpose), au moins efficace dans un domaine choisi d'applications.

Ce chapitre présente un rapide survol de quelques propositions ou réalisations en fonction de leurs caractéristiques [KHAR_86], puis, reprenant l'étude précédente, précise le cahier des charges pour terminer par les choix fondamentaux de LAIOS.

I. 1. CARACTERISTIQUES D'UN RESEAU.

Les divers paramètres qui permettent de distinguer les réseaux entre eux sont d'ordres divers: dimension, topologie, connectique, organisation, gestion de la mémoire ...

I. 1. 1. La granularité.

Un réseau peut être **homogène** ou **hétérogène**. Pour des raisons de simplicité de construction et de programmation, les algorithmes parallèles sont mis en œuvre sur des réseaux homogènes constitués de **modules** identiques.

Le processeur qui se trouve au cœur de chaque module peut avoir une taille extrêmement variable, depuis les processeurs un bit et quelques instructions des réseaux systoliques jusqu'aux processeurs 64 ou 128 bits des calculateurs scientifiques. La granularité du réseau sera spécifiée à la fois par la taille des données, l'ampleur du jeu d'instructions et la quantité de mémoire adressable.

Cette caractéristique a une influence déterminante sur l'implantation. Les réseaux systoliques permettent l'implantation de plusieurs processeurs sur une même puce de silicium, certains réseaux de granularité modeste peuvent être réalisés avec une puce par module alors que d'autres nécessitent une carte (FPS série T) [GUS&_86], [CUS&_87] voire une armoire (RP3 d'IBM) [PFI&_85].

I. 1. 2. L'organisation de la mémoire.

La distinction entre **mémoire partagée** et **mémoire distribuée** s'est avérée l'un des critères les plus fondamentaux en programmation parallèle puisque son influence s'étend jusqu'à l'algorithmique.

Dans les architectures à mémoire partagée, tous les processeurs ont accès directement à toute la mémoire. Si la mémoire ne possède qu'un accès, celui-ci devient rapidement un goulot d'étranglement avec l'accroissement du nombre de processeurs. Pour éviter ce problème, on construit des mémoires multiaccès où la mémoire est divisée en bancs, chaque banc ayant son accès propre [DES&_85], [CAS&_87]. Il faut alors établir un réseau de communication entre les processeurs et les bancs mémoire.

Dans les architectures à mémoire distribuée chaque processeur possède sa mémoire locale. Il reste alors à définir le moyen par lequel un processeur peut accéder aux informations stokées dans les mémoires locales des autres processeurs.

Des méthodes spécifiques de gestion de la mémoire doivent être mises au point. Le problème de **cohérence** des données se pose quelle que soit l'architecture choisie. Dans le cas d'une mémoire partagée, des synchronisations producteurs/consommateurs doivent être mises en place pour pouvoir contrôler la valeur de chaque donnée à chaque accès. Dans le cas d'une mémoire distribuée, les diverses copies distribuées dans les mémoires locales des processeurs doivent rester identiques au cours du temps.

Le problème de la **récupération de la place** [BEK&_86] qui est fondamental dans les algorithmes dynamiques n'a généralement pas de solution simple non plus. Si la mémoire est partagée, il est délicat de savoir lorsqu'une donnée n'est plus utile alors qu'elle peut être utilisée par plusieurs processeurs; la mémoire doit donc être munie d'une capacité d'autogestion. Dans le cas d'une mémoire distribuée, la gestion de la mémoire locale est généralement confiée au processeur du module, mais ce dernier doit être sûr qu'aucun autre processeur ne lui demandera la donnée avant de la détruire, un certain nombre d'informations sur l'état d'avancement du programme lui sont donc nécessaire.

A propos de l'utilisation de la mémoire, il faut mentionner le projet original de Chu [CHU&_85] qui se propose de réaliser une machine PROLOG essentiellement à l'aide d'une mémoire associative de grande taille.

I. 1. 3. L'organisation des communications.

Le problème des communications se pose non seulement entre processeurs et mémoires mais aussi entre processeurs eux-mêmes. La même distinction que précédemment peut être faite entre communications globales qui permettent de transférer de l'information d'un point quelconque du réseau à un autre point et communications locales qui ne permettent les transferts qu'entre éléments d'un même module ou de modules voisins.

Les **communications globales** peuvent être réalisées par un bus reliant tous les points du réseau, par exemple le ZMOB [WEI&_85], ou par un réseau d'aiguillages commandés par l'adresse de la destination, le réseau oméga par exemple [CAS&_87]. Elles peuvent aussi être réalisées par un réseau "postal" où chaque module possède son processeur de communication relié à ses voisins; dès que ce processeur reçoit un paquet de données, en fonction de l'adresse de destination, il le conserve dans le module s'il lui est destiné ou, dans le cas contraire, le renvoie à un processeur voisin plus proche de sa destination [DEV&_85], [COR&_87].

Les **communications locales** peuvent être assurées par une mémoire partagée [BORG_84] ou par des liens entre processeurs; par exemple, le TRANSPUTER possède quatre liens par construction [WHIT_85].

I. 1. 4. Le contrôle.

La multiplicité des centres d'exécutions rend le contrôle plus ardu. Le plus simple à mettre en place est un ensemble de **signaux globaux** qui sont "**radiodiffusés**" en tous points en même temps. Ce sera le cas par exemple du signal de reset. Mais ces signaux ne peuvent que réaliser les tâches les plus grossières.

Des problèmes plus complexes comme par exemple la **répartition de la charge** ne pourront être résolus que par programme. La première solution qui vient à l'esprit pour répartir le travail est de décider à la **compilation** des tâches exécutées par chaque processeur. Cette

solution permet d'optimiser un certain nombre de paramètres comme par exemple le nombre de transferts de données mais souffre d'une trop grande rigidité sur les algorithmes dont l'exécution est dirigée par les données. Une seconde solution consiste à désigner un **processeur maître** dans le réseau qui coordonne le travail de tous les autres. Comme tout contrôle centralisé, la difficulté est d'éviter que celui-ci n'ait un coût prohibitif. La troisième solution consiste à mettre au point une **heuristique distribuée et égalitaire**; chaque processeur contribue avec le même algorithme à la répartition sans avoir une vue globale du problème; il s'ensuit une répartition non optimale mais statistiquement satisfaisante pour un coût acceptable et même souvent modeste.

Un parallèle intéressant peut être fait entre les trois systèmes de répartition de la charge présentés ci-dessus et l'évolution de la gestion de la mémoire dans les diverses générations informatiques. Au début, la mémoire était rare et chère, aussi était-elle gérée par les compilateurs avec parcimonie. Puis, son coût diminuant, un plus grand espace mémoire fut mis à la disposition du processeur, ce qui permit les créations dynamiques d'objets dans une partie réservée à cet effet appelée tas dont le processeur était maître. Enfin, l'espace adressable devenant considérable, l'utilisation de mémoire cache s'est développée, dont la gestion échappe au processeur lui-même pour ne plus faire confiance qu'à la statistique pour améliorer les performances(1).

Un autre problème délicat est la possibilité offerte à l'utilisateur de mettre au point ses programmes (debug). Il faut alors pouvoir contrôler l'état de tous les points du réseau. De plus, un fonctionnement correct en une phase de mise au point ne prouvera pas la validité du programme, des problèmes de synchronisation pouvant passer inaperçus dans un cas particulier de fonctionnement. Il semble évident que ce type de problème encouragera le développement de langages à parallélisme implicite dans lesquels la correction d'un programme en fonctionnement monoprocesseur entraine la correction du même programme pour tout fonctionnement multiprocesseurs.

Un avantage des réseaux multiprocesseurs est d'offrir la possibilité, inconnue des monoprocesseurs, de tolérer les pannes. La possibilité d'isoler certains modules du réseau de manière logicielle, la reconfiguration sur pannes, et de conserver un fonctionnement correct du réseau est évidemment très intéressante. Les techniques permettant la tolérance aux pannes sont sophistiquées, elles seront juste évoquées ici. Il est nécessaire de détecter le module en panne, d'isoler ce module, de définir un état cohérent antérieur à la panne et de reprendre à partir de cet état.

Dans LAIOS, la détection des modules en pannes reste à faire. Par contre, si la base de connaissances est suffisament protégée (toute modification y est datée), il est possible de redémarrer le réseau sous sa nouvelle configuration et d'y soumettre à nouveau les requêtes n'ayant pas abouties précédemment. Bien sûr, cet algorithme n'est pas exempt de critique. En l'absence d'information sur la date exacte du début de la panne, il est impossible de garantir la validité des résultats obtenus avant détection. De même, dans l'hypothèse où des modifications de la base de connaissances auraient été effectuées par le réseau, la question se pose de savoir à partir de quelle date ces modifications doivent être ignorées. L'intervention d'un opérateur humain pour prendre les décisions à ce niveau n'est pas à exclure à priori.

Il est aussi utile de chiffrer la dégradation des performances en fonction des pannes. Il est également intéressant d'éviter d'avoir à recompiler les programmes source à chaque reconfiguration, ce qui plaide pour un maximum de dynamicité dans le fonctionnement du réseau.

(1) *Des phénomènes un peu semblables ont même été mis en évidence dans le fonctionnement du cerveau humain qui a un accès rapide aux informations utilisées fréquement et un mécanisme lent de rappel d'informations anciennes, le fameux esprit de l'escalier.*

I. 1. 5. La topologie du réseau.

Les possibilités de connexions de processeurs et de mémoires sont extrêmement variées et la présentation ci-dessous ne prétend nullement à l'exhaustivité.

Des réseaux à une seule dimension utilisent un bus unique (connectique totale) qui relie les processeurs et la mémoire comme le ZMOB [WEI&_85] ou PRISM [KOH&_87]. Une autre solution consiste à réunir les modules en chaîne (connectique partielle) les uns à la suite des autres. Pour diviser par deux le plus long chemin, on boucle généralement la chaîne en anneau ce qui permet de tolérer un module en panne. Des études ont été menées [ARD&_81] pour corder cet anneau afin de raccourcir encore davantage ce plus long chemin et d'augmenter sa tolérance aux pannes. Quelle que soit la valeur de ces améliorations, le nombre de processeurs dans une configuration de dimension un reste limité.

La conquête de la deuxième dimension permet de réaliser des réseaux d'aiguillages connectant totalement des processeurs et des bancs de mémoires [CRO&_85]. Elle permet aussi de réaliser des réseaux rectangulaires [ONA&_85], [TAKI_86], [COR&_87] ou hexagonaux comme FAIM-1 [DAV&_85], [AND&_86] en connectant chaque module à ses quatre ou six voisins (connectique partielle). Des connections encore plus riches (huit voisins par exemple) sont également possibles. Ces réseaux peuvent éventuellemnt être bouclés en tores pour éviter que les modules aux frontières soient distincts des autres. Elle permet également de réaliser des réseaux en arbres, binaires par exemple comme DADO2 [STOL_87], voire en rebouclant les feuilles de l'arbre sur les autres nœuds, en Sneptree [LI_86]. Elle permet enfin de réaliser des architectures en grappes comme dans P.I.M. [FUC&_87].

La conquête de dimensions supérieures est surtout utilisée dans les hypercubes [COS&_87]. L'intérêt de ce type d'architecture est d'offrir une longueur maximale de chemin égale au logarithme en base deux du nombre de processeurs, une numérotation binaire des processeurs et des chemins d'accès et la possibilité de simuler d'autres topologies (anneau, rectangulaire plan...).

Il faut noter aussi la proposition d'obtenir une configuration programmable à l'aide d'un composant spécifique, le configurateur, proposition envisagée pour le SuperNode à base de Transputer [MUNT_87].

Même si certaines de ces solutions sont plus utilisées que d'autres, aucune n'est encore parvenue à faire la preuve de sa supériorité.

I. 2. LE CAHIER DES CHARGES DE LAIOS.

Il faut reprendre les grandes lignes des deux livres précédents pour chercher leurs implications sur le plan de l'architecture.

I. 2. 1. Le déroulement dynamique.

Une des caractéristiques importantes du modèle d'exécution est le déroulement dynamique de l'arbre de recherche. Chaque nœud processus de cet arbre sera pris en charge par un seul processeur du réseau. Pour des raisons évidentes d'efficacité, la plupart des nœuds n'étant pas actifs en même temps, un même processeur prendra en charge plusieurs nœuds. Un module se devra donc d'être **multitâches**.

Du fait que l'arbre de recherche, l'arbre réellement déroulé, n'est qu'une partie de l'arbre théorique, il avait déjà été noté que l'attribution des nœuds processus aux modules par

compilation préalable ne pouvait conduire à une répartition équilibrée de la charge. Chaque module devra donc être capable de créer n'importe quel nœud processus de l'arbre et de prendre en charge son exécution. Les modules seront dits **banalisés.**

Le choix de recopie des données entraine d'importants mouvements de données entre des nœuds parents proches; il convient donc d'en **minimiser le coût.** Cette contrainte va s'opposer à celle d'une bonne répartition de la charge sur le réseau. Que la situation la plus fréquente soit l'implantation de nœuds processus parents proches sur des modules voisins mais différents est un compromis entre ces deux contraintes.

Il faut alors choisir un nombre suffisant de voisins, c'est-à-dire physiquement reliés par des liens de communications, pour chaque module. Une connectique totale est la solution la plus grossière au problème mais son coût important, son manque de souplesse et le fait que ses liens ne seraient que faiblement utilisés la font rejeter au profit d'une connectique partielle.

Les quelques statistiques réalisées sur des ensembles de programmes PROLOG [ONA&_84] estiment à environ 2,5 le nombre moyen de fils pour un nœud. Ce nombre n'est grand (supérieur à la dizaine) que dans des programmes de type bases de données; il est facile de voir qu'une compilation plus fine des nœuds modèles permet dans ce cas de limiter ce nombre en effectuant un premier tri sur un ou plusieurs paramètres choisis comme critères. Un nombre de voisins inférieur à la dizaine devrait s'avérer suffisant d'autant plus que les modules étant multitâches, il n'y a pas lieu de craindre des blocages avant saturation.

I. 2. 2. Les différentes couches.

La hiérachie de classes effectuée sur les nœuds, (classe de nœuds, nœuds modèles, nœuds processus), se traduit dans l'architecture par la réalisation de trois couches différentes.

La **couche contrôle** se situe au niveau des classes de nœuds. Ceux-ci sont des automates définis par le langage lui-même et son modèle d'interprétation. Ces automates vont donc se traduire par des programmes système identiques présents dans chaque module.

La **couche connaissance** va prendre en charge les nœuds modèles. Ceux-ci représentent une masse considérable d'informations à durée de vie longue. Cette couche comprend un stockage de masse de type disque, magnétique ou optique, lorsque les problèmes de réécriture seront résolus. Chaque module du réseau est banalisé, il doit donc avoir accès en lecture à la totalité des informations mémorisées. Cette couche a donc un réseau de transmission en lecture uniquement, reliant chaque module à la mémoire de masse.

La **couche données** concerne les nœuds processus et leur activité. Chaque nœud est entièrement pris en charge par un module du réseau qui gère sa vie. Cette couche comprend les processeurs et une partie de la mémoire locale de chaque module; la durée de vie des objets appartenant à cette couche est inférieure à la durée d'exécution du programme contrairement aux informations appartenant aux deux couches précédentes. Cette couche comprend aussi les liens physiques reliant les modules voisins, ceux-ci supportent deux types d'échanges: les transferts de données entre nœuds processus qui peuvent avoir une taille assez importante du fait du choix de la recopie de données, les échanges de messages entre nœuds processus qui sont toujours limités à quelques mots mais doivent pouvoir être pris en compte immédiatement.

I. 2. 3. Les contraintes externes au modèle.

De plus, un certains nombre d'exigences viennent du concepteur pour assurer la faisabilité du réseau et tenir compte des critères de coût de réalisation et d'exploitation.

La manière la plus économique de fabriquer un réseau consiste à répéter un grand nombre de fois le même motif. C'est pourquoi le réseau doit être **modulaire** avec un petit nombre de modules différents. Pour pouvoir adapter facilement la puissance à la demande, il est bon également que ce réseau soit **extensible** par ajout de modules.

Afin de limiter le coût de son exploitation, le réseau devra être **multi-usagers**. Avec l'essor des microordinateurs et leur montée en puissance, le réseau peut être conçu comme le dorsal de plusieurs microordinateurs qui permettraient de s'affranchir de toute l'interface homme-machine.

Enfin, il est souhaitable que le réseau ait une **tolérance aux pannes** la plus grande possible afin que la panne de quelques modules n'interdise pas son exploitation. Le problème difficile de la détection des pannes ne sera pas envisagé ici et sera supposé résolu. L'hypothèse de travail est que chaque module, lors de la mise en route, est informé de l'état de ses voisins et est capable de lui-même de renoncer à communiquer avec tout voisin déclaré en panne.

I. 3. LES CHOIX ARCHITECTURAUX.

Certains de ces choix ont commencé à s'esquisser lors de l'exposé du cahier des charges. Ils vont être précisés ici.

I. 3. 1. Le principe de localité.

Du fait de la quantité d'échange entre deux parents proches d'une part, de la possibilité de supprimer tout échange entre parents non proches dans le modèle de LAIOS, il est possible de formuler l'exigence suivante:
deux nœuds processus parents proches (père-fils ou ainé-cadet) **sont implantés sur le même module ou sur des modules physiquement voisins** (principe de localité).

Ce principe a essentiellement pour but de diminuer le coût des échanges de données dus au choix de recopie de données. Il facilite grandement la mise en œuvre des communications dans la couche données puisqu'il autorise une connectivité partielle. D'après le modèle défini précédemment, un processus actif dispose toujours dans la mémoire locale du module qui l'a pris en charge des données qui lui sont nécessaires (modèle data flow).

Il n'en va évidemment pas de même dans la couche connaissances où chaque module doit être relié en consultation à la mémoire de masse.

I. 3. 2. La structure hexagonale.

Il est donc possible de limiter le nombre de voisins physiques d'un module à quelques modules. Ce nombre doit être choisi de manière à permettre un déroulement sans blocage et le plus réparti possible de l'arbre de recherche.

Bien que, grâce au multitâche, il soit possible de faire autrement, le principe de localité rend fortement souhaitable la présence de cycles triangulaires dans le réseau autorisant l'implantation sur des processeurs différents du père, d'un fils ainé et d'un fils cadet travaillant en ET pipe line.

Les réseaux complets sont donc abandonnés, comme trop coûteux, difficilement extensibles et trop riches pour les besoins; les réseaux en dimension un sont refusés pour des raisons symétriques: pas assez riches et trop sensibles aux pannes. La candidature des réseaux en

grappes ou des hypercubes est écartée car ils n'offrent pas de cycles triangulaires, de même que parmi les réseaux plans, celle des arbres binaires et des réseaux rectangulaires. En fin de compte, il reste peu de candidats parmi les réseaux classiques. Le plus simple satisfaisant aux conditions précédentes est le réseau plan hexagonal.

Le **réseau plan hexagonal** offre six voisins pour chaque module. Ces voisins sont eux-mêmes voisins deux à deux, ce qui facilite le ET pipe line. Le réseau est facilement extensible par ses bords. Il est plan et donc facilement réalisable sur carte; il faut cependant remarquer que cette contrainte n'est pas très forte dès que les connexions souples entre cartes sont possibles (elle serait au contraire très forte dans une réalisation WSI, c'est-à-dire sur une seule rondelle de silicium).

Ce nombre de six voisins est également à comparer avec la parenté moyenne d'un nœud processus (un père, deux frères éventuels, deux ou trois fils). Elle permet d'ores et déjà de préssentir la possibilité de programmer une répartition de la charge satisfaisante. Une étude plus précise dans le livre suivant sur les simulations confirmera que cette topologie offre une possibilité intéressante de répartition de la charge et une bonne tolérance aux pannes.

I. 3. 3. Les modules.

La plupart des modules du réseau hexagonal sont des modules opératoires prenant en charge les nœuds processus. Toutefois, un certain nombre de modules ont une autre finalité. Un module est réservé à la mémoire de masse, vu son rôle il est placé au centre du réseau.

Un module placé en périphérie peut être un module opératoire connecté au module périphérique symétrique par rapport au centre, le réseau se rebouclant alors sur lui-même à la manière d'un tore, cette architecture est déjà celle de FAIM [DAV&_85], [AND&_86]. Il peut être aussi, c'est une autre solution, un module d'interface avec l'extérieur, soit connecté à un microordinateur frontal, soit connecté directement à une console et à un écran avec un système intégré. Ces modules d'interfaces résolvent aisément le fonctionnement multi-usager en utilisant un module périphérique par utilisateur. De toute façon les six modules périphériques placés aux coins du réseau ne peuvent avoir un nombre suffisant de voisins pour être de véritables modules opératoires et seront toujours des modules d'interfaces avec l'extérieur.

Enfin, d'autres modules spécialisés peuvent être utilisés comme il apparaitra plus loin avec les modules mémoires cache.

I. 3. 4. La numérotation des modules.

Puisque le réseau est plan, il suffit de deux coordonnées pour numéroter chaque module. Le réseau contient trois axes orientés à 60° les uns des autres. Il suffit de privilégier deux d'entre eux, le repère n'est pas orthogonal.

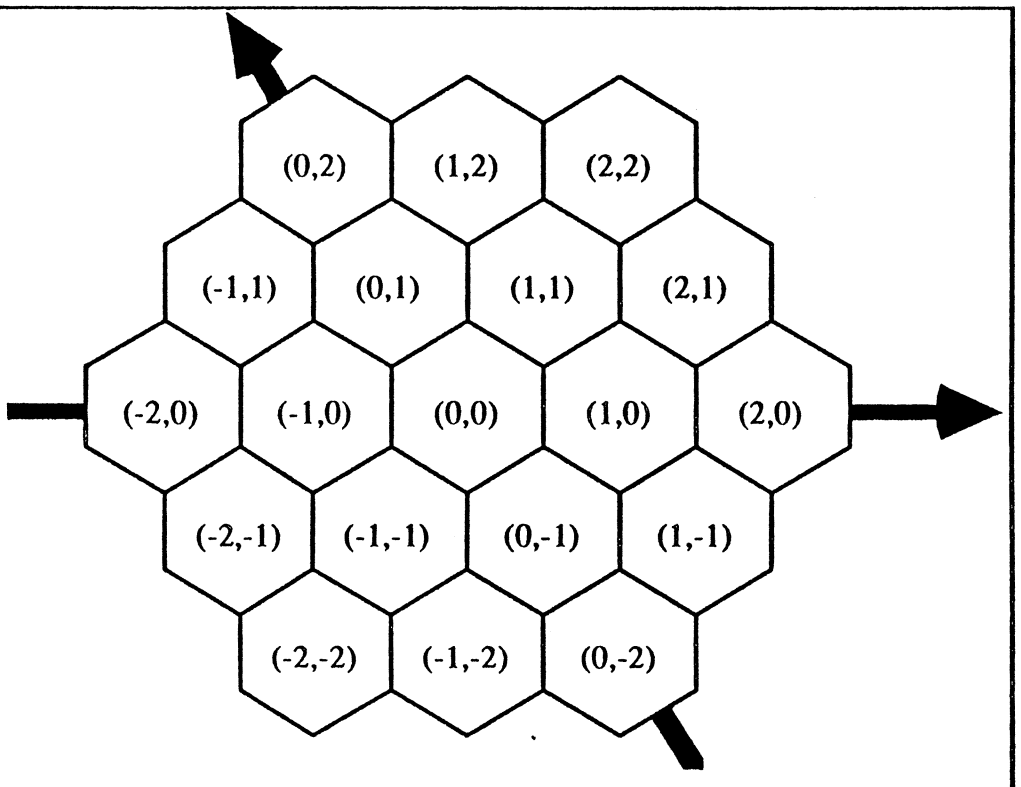


figure 3.1 : la numérotation dans le réseau.

Ainsi, le module numéroté (x,y) a pour voisins les modules numérotés (x+1,y), (x-1,y), (x,y+1), (x,y-1), (x+1,y+1) et (x-1,y-1). Le module mémoire de masse a pour numéro (0,0).

CHAPITRE II.
LES COMMUNICATIONS.

La facilité de communication entre les modules d'un réseau et l'adéquation du réseau à l'exécution des algorithmes sont des facteurs fondamentaux de performances. Dans LAIOS, chacune des trois couches définies précédemment dispose d'un réseau de communications adapté à ses besoins propres.

II. 1. LE TRANSFERT DE CONNAISSANCES.

Rappelons que les connaissances sont écrites sous forme de clauses PROLOG, qu'elles sont compilées sous forme de nœuds modèles. Ces nœuds modèles sont stockés dans une mémoire de masse au centre du réseau.

II. 1. 1. Organisation du module mémoire de masse.

Le compilateur voit la mémoire de masse comme une **base de connaissances**, il utilise les primitives d'**ajout**, de **retrait** ou de **modification**. Cette solution est naturelle dans les langages de programmation déclaratifs. Elle autorise une grande souplesse dans la mise au point des programmes; les modifications sont apportées alors par l'utilisateur.

Il est possible que les programmes aient eux-mêmes accès au gestionnaire de la base de données pour rendre possibles les programmes **auto-apprenants**. Toutefois, pour éviter les effets de bords, il est impossible de répercuter ces modifications sur les programmes en cours d'exécution. La base devra alors posséder une **horloge propre** qui permettra de dater les versions successives de la base et toujours donner à un programme l'accès à la base dans l'état où elle se trouvait à la date de début de l'exécution du programme.

Par contre, les modules opératoires voient la mémoire de masse comme le moyen de pouvoir consulter les nœuds modèles qui lui sont utiles. Un module opératoire a besoin de consulter le nœud modèle lors de la création et de l'activation en descente ou en remontée du nœud processus. La mémoire est donc partagée en lecture uniquement par tous les modules opératoires; cette situation est la plus favorable à l'utilisation de caches.

La mémoire est donc paginée et chaque nœud modèle est repéré par son adresse globale avec un numéro de page et un numéro de ligne. La taille d'un nœud modèle est à quelques exceptions près comprise entre une dizaine et une centaine de mots. C'est pourquoi une taille de 1Kmots est raisonnable pour les pages; un nœud modèle est en principe contenu dans une page et chaque page contient généralement quelques dizaines de nœuds modèles.

Afin de diminuer les accès mémoire, le gestionnaire de la base de connaissances place dans la même page des nœuds modèles parents proches.

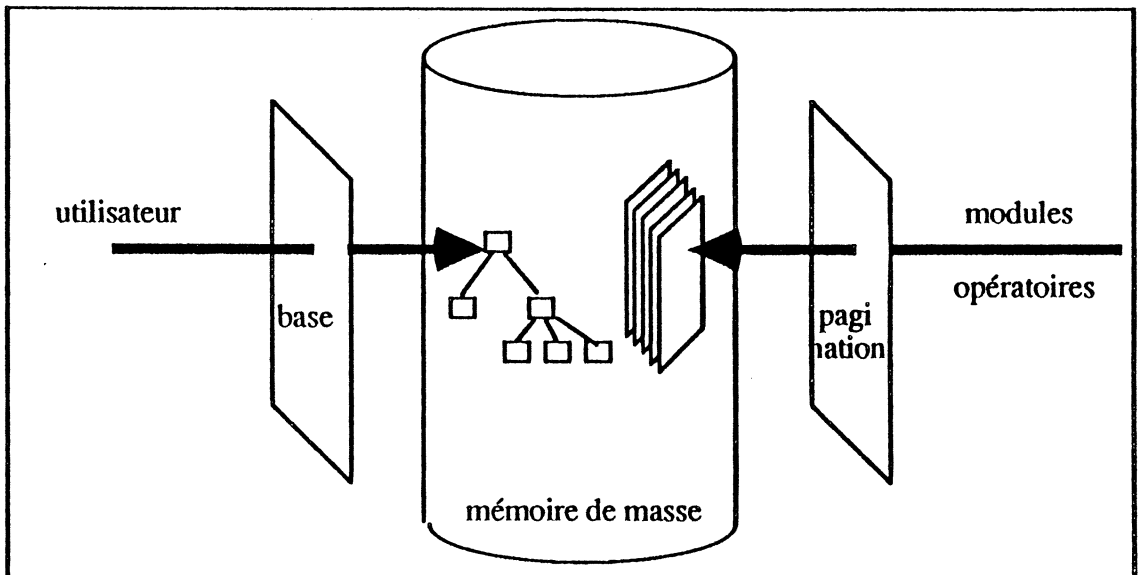


figure III.2 : les vues de la mémoire de masse de LAIOS.

II. 1. 2. Première solution architecturale.

Cette solution utilise un bus pour relier le module mémoire de masse et les autres modules.

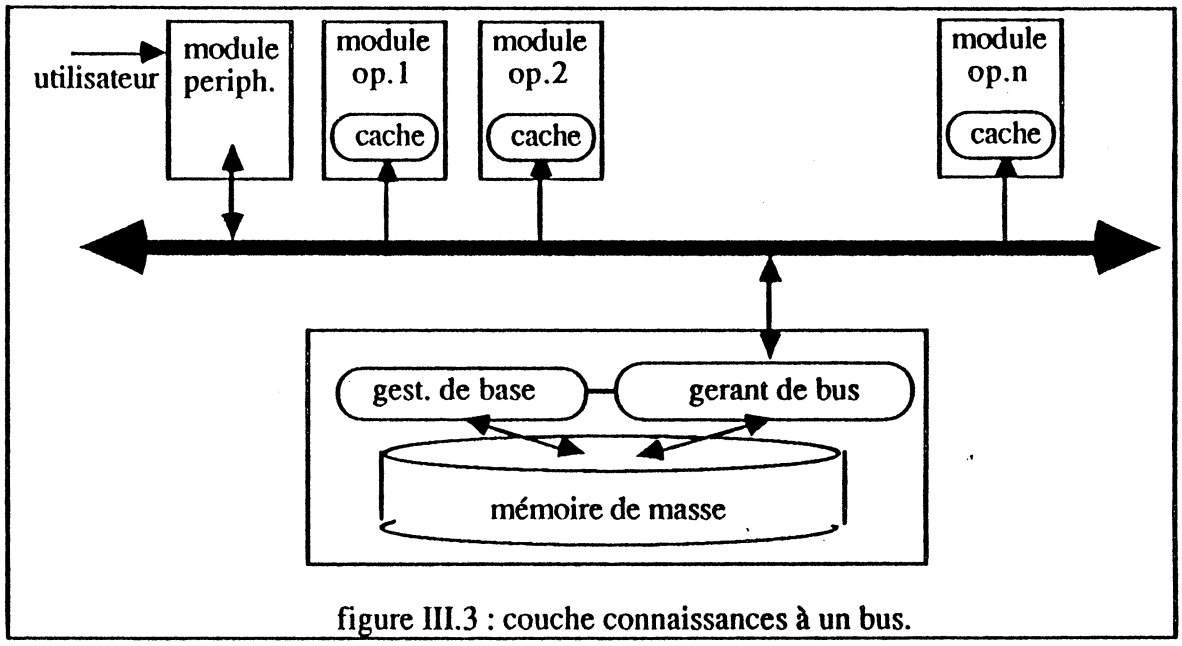


figure III.3 : couche connaissances à un bus.

Dans une telle architecture, le bus sert à transférer aussi bien les demandes de fonctionnalités base de connaissances que des demandes de pages ou leur transfert.

Le module mémoire de masse est le **maître du bus**, il possède un **gérant de bus** qui synchronise les communications, recense les requêtes et assure les transferts de pages. Il possède aussi le **gestionnaire de la base** qui exécute les primitives de gestion, contrôle la mise en page et gère la datation.

Chaque module opératoire possède sa **mémoire cache** avec son **gérant**. Celui-ci effectue une requête à chaque défaut de page et gère au mieux les pages physiques du cache.

Les modules périphériques assurant la communication avec l'extérieur; ils peuvent soit transmettre une requête au gestionnaire de la base de connaissances, soit demander une page et une date pour créer un nœud processus de type REQ racine d'un arbre de recherche (exécution d'un programme).

Le problème avec une telle architecture vient du fait que des nœuds processus parents sont pris en charge par des modules distincts. Si aucun mécanisme spécialisé n'est mis en place, les demandes de la même page se multiplient et il n'a servi à rien que le gestionnaire de la base ait regroupé sur la même page les nœuds modèles des parents proches.

Une solution à ce problème est la mise en place d'un **espion** dans chaque module opératoire. Cet espion observe en permanence le trafic sur le bus. De son côté, le gérant du cache laisse toujours au moins une page physique du cache effaçable. Dès qu'une page est transférée sur le bus, l'espion la recopie sur une page physique effaçable du cache. Le rôle du gérant de cache est le calcul de l'importance de chaque page; la page en cours d'utilisation est ineffaçable, les pages demandées sont plus importantes que les pages simplement espionnées. Ainsi, la probabilité pour que la page contenant le modèle du parent d'un nœud processus actif soit déjà présente dans le cache du module qui prend ce parent en charge est considérablement augmentée.

II. 1. 3. Deuxième solution architecturale.

Si le réseau contient un grand nombre de modules, il y a un risque non négligeable que le bus devienne un goulot d'étranglement. C'est pourquoi, pour les réseaux de grande taille, une seconde solution est envisagée.

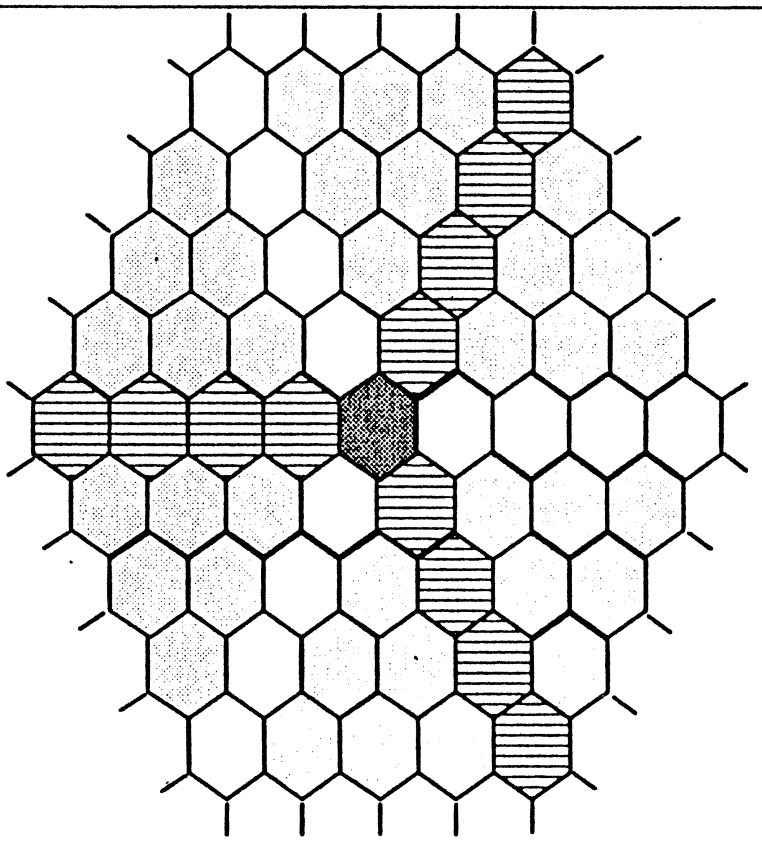


figure III.4 : le réseau hexagonal avec les modules mémoires caches.

Elle consiste à distribuer dans le réseau un certain nombre de **modules** spécialisés **mémoires cache**. Ces modules ont pour but de créer des tampons entre la mémoire de masse et les modules opératoires dont la taille de la mémoire cache est modeste.

Ces modules sont placés sur les trois axes du réseau hexagonal en Y (figure III.4).

Afin que ces modules soient **transparents** pour la couche données, d'autres modules n'ayant pas de rôle sont placés symétriquement (figure III.5). Ainsi, les modules opératoires séparés par des modules caches ou symétriques restent voisins. Les réseaux avec les modules caches et les modules symétriques ou sans ces modules sont équivalents pour les voisinages des modules opératoires.

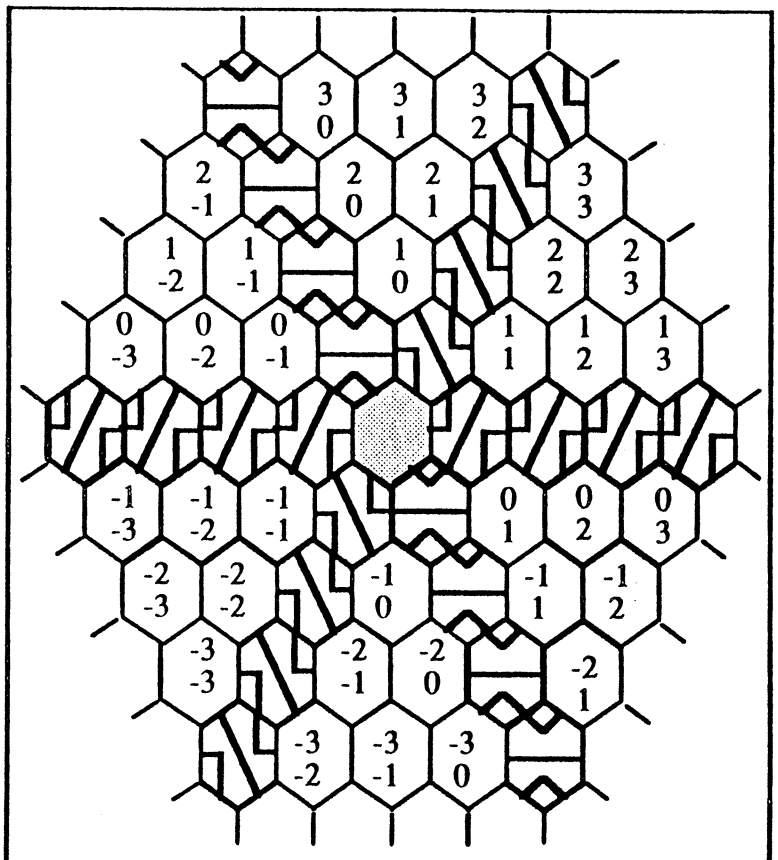


figure III.5 : conservation des voisinages au travers des modules mémoires caches.

Un **bus primaire** relie ces modules mémoires caches avec le module mémoire de masse (figure III.6). Le fonctionnement de ce premier niveau est identique à celui de la première solution.

Des **bus secondaires** relient les modules opératoires aux mémoires caches en suivant les trois directions du réseau (figure III.6).

Ainsi la fonctionnalité mémoire cache reste conservée. Les modèles des nœuds parents lorsqu'ils sont sur la même page sont accessibles rapidement par des modules opératoires voisins.

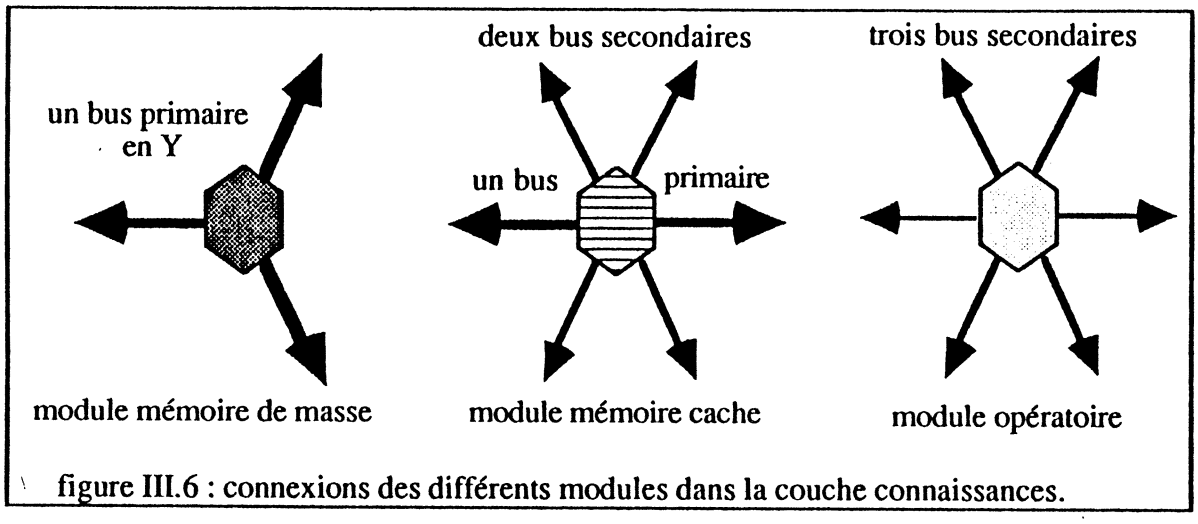


figure III.6 : connexions des différents modules dans la couche connaissances.

La figure III.7 se commente comme suit. Supposons que le module opératoire mo0 prenne en charge un nœud processus N0. La page contenant le modèle de N0 sera demandée aux trois mémoires caches mc1, mc2 et mc3. Celles de ces trois mémoires qui n'ont pas cette page la demandent à la mémoire de masse mm. Lorsqu'un voisin de mo0 -mo1, mo2, mo3, mo4, mo5 ou mo6- va vouloir prendre en charge un fils N1 du nœud N0, en supposant que le gestionnaire de la base de connaissances ait pu placer le modèle de N1 dans la même page que N0, il se trouve au moins un des caches possédant déjà cette page et susceptible de répondre immédiatement à la requête du module opératoire.

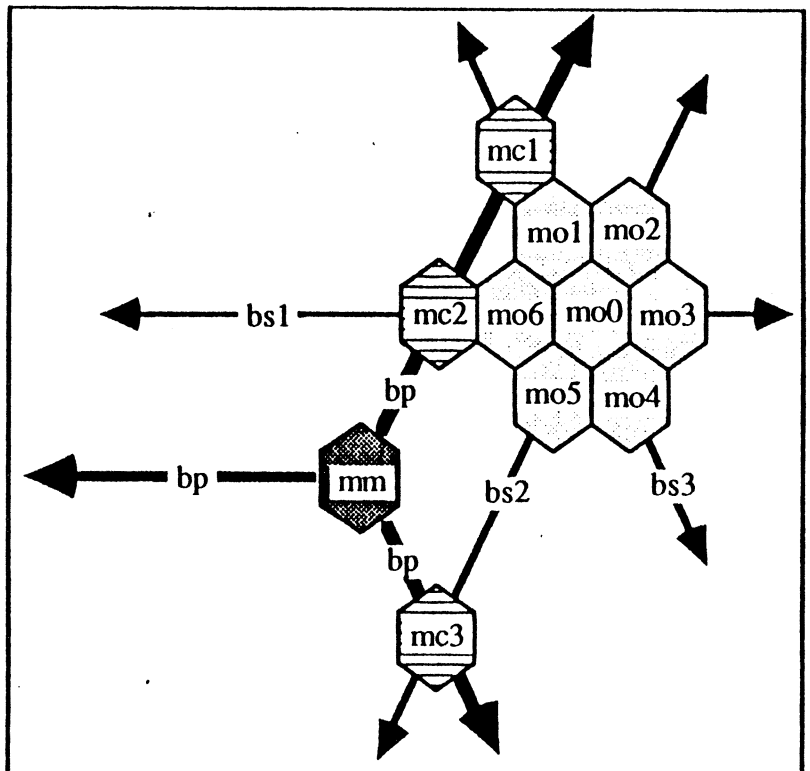


figure III.7 : généralisation de la fonction cache au réseau.

Pour éviter que les trois requêtes entrainent trois transferts de page, il suffit de modifier le protocole; une requête entraine une réponse de présence de la page en mémoire cache (plus ou moins rapide), cette réponse de présence est traduite comme une offre de transfert qui est

acceptée ou refusée par le module opératoire.

II. 2. LES COMMUNICATIONS LOCALES.

Entre deux modules opératoires, deux sortes d'échanges ont lieu. Deux processus parents implantés chacun sur l'un des modules peuvent échanger des messages ou des données. Les **messages** sont toujours de petite taille, un ou deux mots, et doivent être pris en compte immédiatement pour autoriser le meilleur parallélisme possible. Les **données** sont soit sous forme de blocs, soit sous forme de tableaux de variables, elles occupent couramment plusieurs dizaines voire centaines de mots. Elles sont produites par un nœud processus puis consommées par un autre nœud, parent proche du précédent.

II. 2. 1. Les boîtes aux lettres.

Entre deux modules voisins sont installées deux **boîtes aux lettres**, une dans chaque sens. Le module émetteur a la possibilité de savoir si la boîte est vide et, dans l'affirmative, d'écrire un message dans cette boîte. Le module récepteur est prévenu par interruption de la présence d'un message dans la boîte, il a alors la possibilité de le lire une fois ce qui a pour effet de vider la boîte.

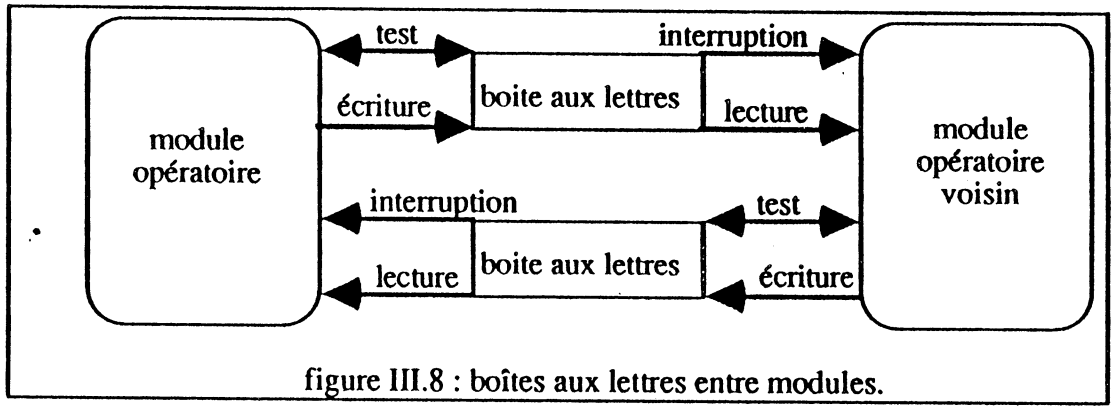


figure III.8 : boîtes aux lettres entre modules.

Le format des messages est constitué de trois champs: le **code** du message, l'**adresse** locale du nœud **processus destinataire** et éventuellement une **adresse de données**.

Pour permettre la reconfiguration sur pannes, chaque module opératoire doit pouvoir inhiber les interruptions venant de ses boîtes aux lettres.

II. 2. 2. Les mémoires partagées.

Les mémoires à double accès, par exemple HM65231[BE&T_86], dédoublent leur port adresses et données. Elles possèdent un arbitre d'accès fonctionnant généralement selon des protocoles simples comme par exemple **"premier entré, premier servi"**. Les signaux de contrôle sont soit synchrones et provoquent l'attente (busy), soit asynchrones selon la technique classique du rendez-vous (req et ack).

L'emploi de mémoires partagées pour réaliser les transferts de données entre modules opératoires voisins présente plusieurs avantages (figure III.9).

Les processus émetteur et récepteur n'ont pas besoin d'être actifs simultanément pour que l'échange ait lieu. Ce procédé respecte donc le fonctionnement multitâche de chaque module et ne nécessite **ni synchronisation ni retard** du à des changements de contextes imposés dans

l'un des modules.

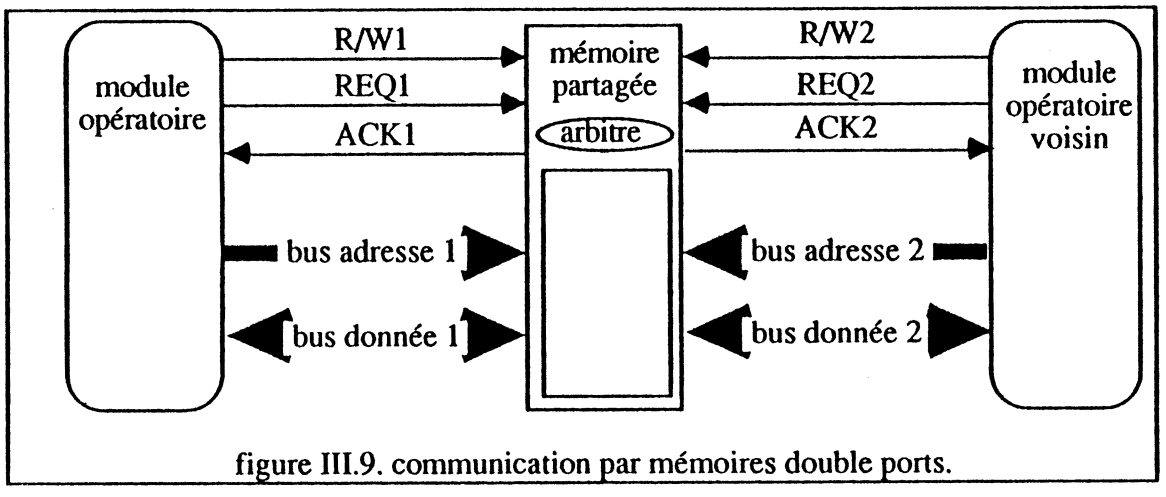


figure III.9. communication par mémoires double ports.

La taille des données peut être **quelconque** pourvu qu'elle ne dépasse pas les possibilités mémoire. Ceci était indispensable pour obtenir un fonctionnement conforme au modèle. Il faut cependant remarquer que rien n'est prévu, et que d'ailleurs le problème n'est pas simple, pour transmettre une donnée d'une taille supérieure à la taille disponible dans la mémoire partagée.

Enfin, le mécanisme contrôlant un processus n'a à se soucier de l'adresse du destinataire que pour le choix de la page mémoire, sinon le fonctionnement est identique que le destinataire soit un processus installé dans le même module ou dans un module voisin. Ce fonctionnement est toujours basé sur le même principe (voir livre précédent, le détail des algorithmes d'unification et d'évaluation) qui consiste à lire les données en notant dans le brouillon les modifications puis à mettre en forme toujours dans le brouillon les données qui seront envoyées par recopie en bonne place au(x) destinataire(s).

II. 2. 3. La gestion des mémoires partagées.

Le problème qui n'a pas été évoqué dans le paragraphe précédent est la gestion de telles mémoires pour éviter les conflits d'adressage. En fait, ce problème se replace dans le contexte plus général de la gestion de l'espace mémoire et de la récupération de la place après utilisation.

A chaque page mémoire données est donc associé un **gestionnaire d'occupation**. Ce gestionnaire est accessible par les mêmes processeurs que la page mémoire qu'il gère. Ce gestionnaire a une primitive d'allocation et une de désallocation.

Un processeur effectue une requête d'**allocation** auprès du gestionnaire en lui indiquant la longueur d'espace contigu désiré dans la mémoire. Si la réservation est possible, le gestionnaire l'effectue, cet espace étant dès lors occupé et transmet l'adresse du premier mot de cet espace au processeur demandeur. Si aucun espace de la taille demandée n'est libre, le gestionnaire répond négativement à la demande.

Un processeur peut aussi effectuer une requête de **désallocation** d'un espace auprès du gestionnaire. Il lui indique alors la taille de l'espace à désallouer ainsi que l'adresse du premier mot. Le gestionnaire répond par un acquittement car l'opération est toujours possible.

Le choix de recopie de données facilite grandement la gestion de la mémoire car soit une donnée est interne à un processus soit elle est produite par un processus et consommée par un autre. Les deux primitives d'allocation et de désallocation décrites ci-dessus suffisent donc à

assurer la gestion de toute la mémoire donnée avec la récupération de la place.

Il est certain que les algorithmes des deux primitives exposés ci-dessus pourraient être perfectionnés par des attentes en cas d'allocation impossible ou de réorganisation de la mémoire, par des vérifications lors de la désallocation. Le but poursuivi dans LAIOS est différent, ces gestionnaires étant souvent sollicités, leurs algorithmes doivent être suffisament simples pour pouvoir faire une réalisation cablée de ces gestionnaires.

II. 2. 4. Les communications avec l'extérieur.

Ces communications sont assurées par des modules spécialisés situés à la périphérie du réseau. Le fonctionnement multi-usagers est réalisé par la multiplication de ces modules. Aucun risque d'interférence n'existe du fait de la structuration en arbre de recherche du déroulement des programmes. Le seul point nécessitant une définition claire de ce qui est partagé et de ce qui est propre à chaque utilisateur se situe au niveau de la base de connaissances et les techniques à employer sont identiques à celles utilisées à l'heure actuelle dans les bases de données.

Les modules d'interfaces peuvent être soit de véritables ordinateurs avec un programme système soit plus simplement des dorsaux d'ordinateurs. Cette vision consistant à utiliser des ordinateurs personnels comme frontaux de machines puissantes présente beaucoup d'avantages. Les machines sont alors déchargées des tâches de dialogues avec les utilisateurs au profit des tâches de calcul proprement dite.

Les communications peuvent être de trois types différents.

Soit l'utilisateur désire lancer **l'exécution d'un programme**, le module assure alors la prise en charge du nœud processus racine de l'arbre de recherche et utilise les modules opératoires voisins pour démarrer les premières branches de l'arbre.

Soit l'utilisateur désire **dialoguer avec la base de connaissances** et le module réalise alors l'interface permettant d'accéder au bus puis au gestionnaire de la base.

Soit l'utilisateur désire réaliser une **fonctionnalité système**. Parmi ces fonctionnalités se trouvent: le reset impératif, le reset avec reconfiguration du réseau pour pouvoir déclarer un ou plusieurs modules en panne (opérations privilèges de l'opérateur), le reset après accord des autres utilisateurs, l'abandon qui permet de détruire l'arbre de recherche à partir de la racine sans modifier les autres arbres se déroulant sur le réseau.

Le parallélisme étant implicite dans LAIOS, la **mise au point** des programmes peut être faite en monoprocesseur, l'exécution pas à pas d'un programme est donc assurée par un module interface sans utiliser les autres modules du réseau.

CHAPITRE III.
LE MODULE OPERATOIRE.

Le module opératoire de LAIOS possède à peu près la puissance d'un microordinateur. Il peut être réalisé avec presque exclusivement des circuits du commerce, organisation qui sera exposée dans les premiers paragraphes de ce chapitre. Le chapitre se termine par l'exposé d'architectures spécialisées.

III. 1. ARCHITECTURE DU MODULE.

Un module opératoire ne contient qu'un processeur et de la mémoire. L'architecture de ce module confie un certain nombre de tâches (ramasse miettes, dialogue avec les voisins, dialogue avec les bus, détection et gestion des défauts de page...) directement à la mémoire qui acquiert une part d'intelligence et décharge le processeur.

III. 1. 1. Organisation générale.

Le module opératoire est formé d'un processeur 32 bits relié à des pages mémoires par un bus interne au module. Le bus adresse interne n'est que de 16 bits car une adresse locale tient sur le troisième champ d'un descripteur (cf livre premier, les objets de LAIOS).

Une adresse est découpée en deux champs, le numéro de la page et le numéro de la ligne. Actuellement, un module opératoire contient 16 pages de 1K mots de mémoire, ce qui n'utilise pas tout l'adressage. Deux bits sont encore disponibles pour étendre, soit le nombre de lignes par page jusqu'à 4K, soit le nombre de pages jusqu'à 64, soit les deux jusqu'à 32 pages de 2Kmots. Ces extensions permettront de tenir compte d'exigences nouvelles apparaissant lors de l'exécution de gros programmes.

Le processeur doit être capable d'effectuer les opérations arithmétiques et logiques classiques mais une faible part de son temps de calcul y sera consacré. Il est donc inutile qu'il possède toutes les opérations cablées mais seulement celles de base, les autres pouvant être obtenues par programme. Le processeur choisi doit également être capable de découper un mot par champs de huit bits afin d'accéder à l'étiquette, l'arité ou la valeur du descripteur.

Un petit nombre (quatre de chaque) de registres de données, de registres d'adresses et de compteurs sont indispensables pour obtenir une programmation rapide de l'unification ou de l'évaluation (cf livre deux). Les modes d'adressages les plus utilisés sont l'adressage indirect pour la déréférenciation et l'adressage indexé pour les tableaux de variables.

Il n'est pas indispensable de construire un nouveau processeur 32 bits pour animer les modules opératoires de LAIOS, des microprocesseurs comme le 68020 de Motorola (pour son jeu d'interruptions) ou le TRANSPUTER d'INMOS (pour sa mémoire interne) satisfont au cahier des charges.

III. 1. 2. Organisation de la mémoire.

La mémoire est divisée en plusieurs bancs d'une ou plusieurs pages physiques ayant des rôles distincts.

Une page est destinée au système (couche contrôle), elle contient les programmes des automates, les programmes des processus mère, contremaître et contrôleur de charge, la gestion des interruptions venant des boîtes aux lettres, du gestionnaire du cache ou du bus (reset) et enfin l'état des modules voisins pour la reconfiguration. Elle contient aussi les programmes de toutes les opérations arithmétiques sophistiquées (opérations en flottant par exemple). Une partie de ce banc est en ROM et l'autre en RAM.

Neuf pages sont destinées à accueillir les données et les descripteurs de nœuds processus. Six d'entre elles sont des pages doubles accès partagées avec les modules voisins pour la transmission des données. Chaque page est munie de son gestionnaire d'occupation cablé. La numérotation de ces pages est telle que chaque page partagée a le même numéro de page dans les deux modules qui se la partagent. Il a suffi pour faire cette numérotation de reprendre la coloration du réseau hexagonal par trois couleurs.

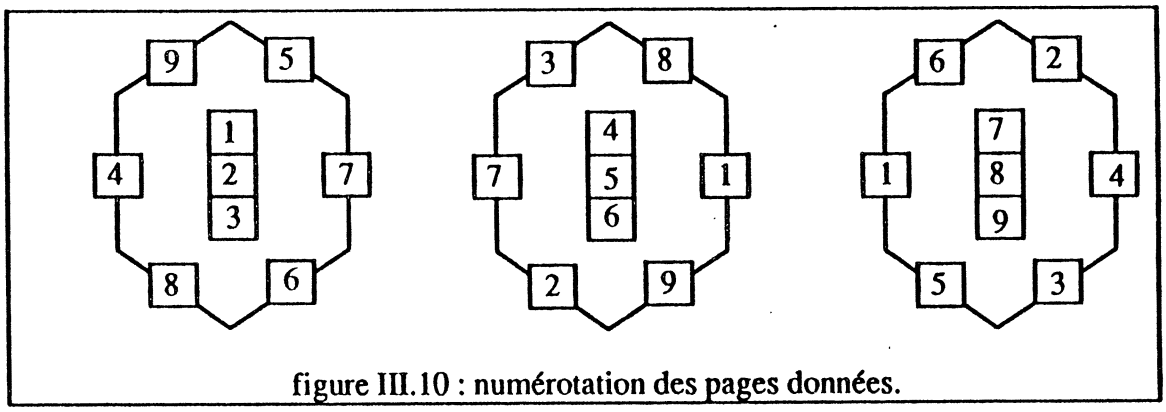


figure III.10 : numérotation des pages données.

Une page est réservé au brouillon. Cette page doit avoir un accès rapide par le processeur car elle est fréquemment utilisée. L'idéal serait de disposer de cette page à l'intérieur du processeur comme dans le TRANSPUTER [WHIT_85]. Cette page est divisée en quatre quarts, deux tableaux et deux piles, cette division pouvant être uniquement logicielle. De plus, cette page doit pouvoir être remise à zéro (au moins toutes les lignes des tableaux) en une seule opération (changement de tâches).

Les cinq dernières pages sont utilisées pour stocker les nœuds modèles. Comme celà a déjà été dit précédemment, ces pages sont organisées en mémoire cache des pages de la base de connaissances. Un gestionnaire de ce cache contient la mémoire associative permettant de détecter les défauts de pages; il prend en charge les requêtes de pages, le chargement de ces pages à partir d'un bus connaissances en accès direct à la mémoire et le calcul de l'importance de chaque page.

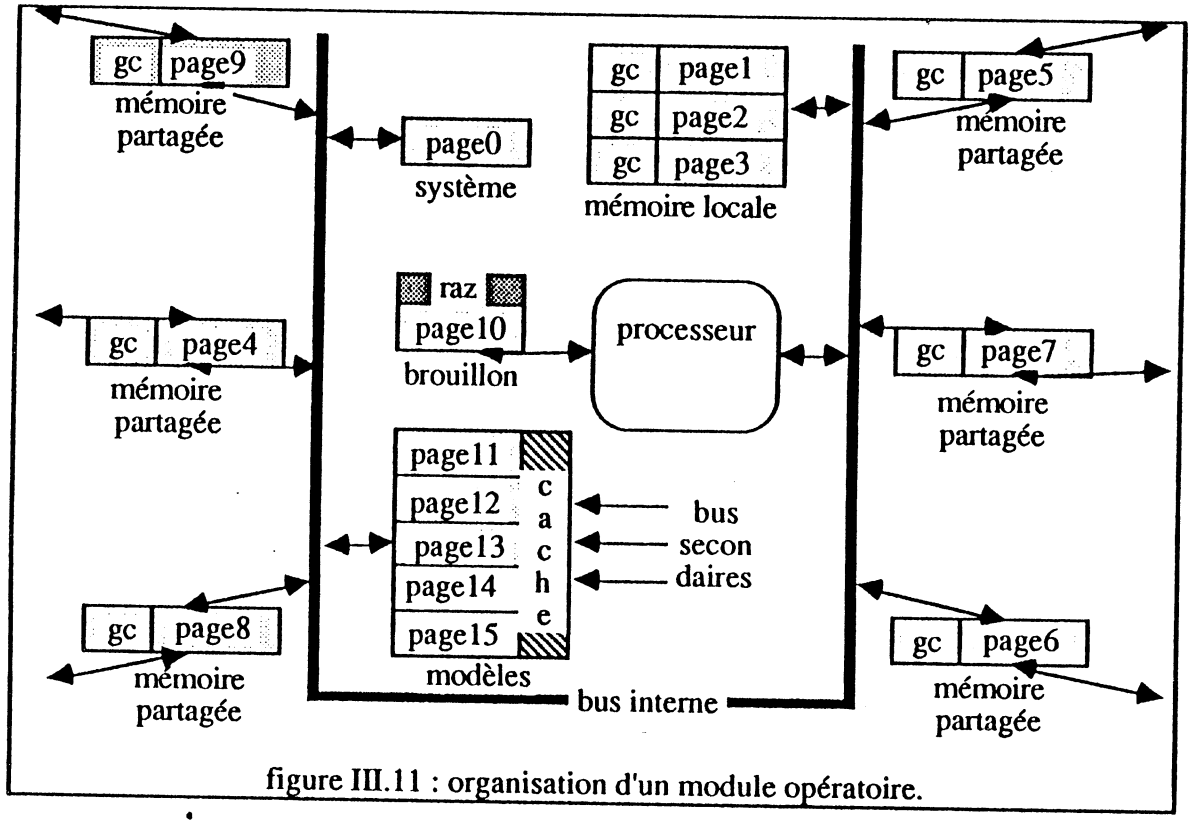


figure III.11 : organisation d'un module opératoire.

III. 2. CIRCUITS SPECIFIQUES.

Le premier circuit décrit -le gestionnaire d'occupation- est indispensable au module quel que soit le choix du processeur. Le reste du module peut être réalisé avec des circuits commerciaux ou avec les circuits spécifiques proposés ci-dessous.

III. 2. 1. Réalisation du gestionnaire d'occupation.

Le gestionnaire d'occupation peut être à simple ou double accès selon la nature du banc mémoire qu'il contrôle. Sur chaque accès, il possède une broche de sélection et une connexion aux bus adresse et donnée locaux (figure III.12).

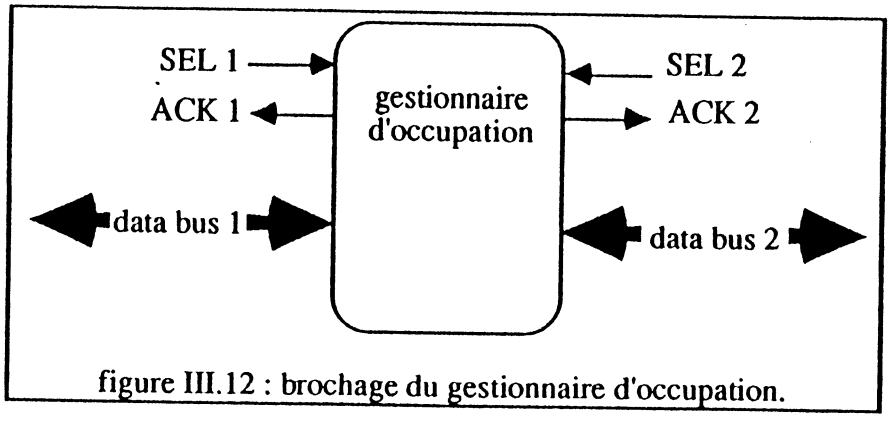


figure III.12 : brochage du gestionnaire d'occupation.

Lorsque le gestionnaire est selectionné, il effectue une lecture sur le bus données. Le mot de 32 bits qu'il y trouve se divise en trois champs: un **code opération** (allocation ou désallocation), un champ **longueur** (1024 au maximum soit 10 bits) et un champ **adresse**

locale utilisé uniquement en désallocation pour indiquer le numéro de la première ligne. Il transmet alors une réponse valide avec le signal d'acquittement qui, en allocation uniquement, utilise le bus données pour transmettre un mot de trois champs également: l'état (succès ou échec), la longueur et l'adresse locale qui indique la première ligne.

Le probléme algorithmique de l'allocation se ramène à déterminer la première occurrence d'un mot de n lettres consécutives identiques codant "libre" dans un mot de p (ici 1024) lettres d'un alphabet à deux lettres: "libre" et "occupé". Plusieurs algorithmes sont utilisables et le choix porte sur le meilleur compromis surface/rapidité.

L'algorithme proposé consiste à réaliser le calcul en parallèle sur les 1024 emplacements en n itérations. L'architecture du gestionnaire d'occupation se conçoit alors de la manière suivante (figure III.13).

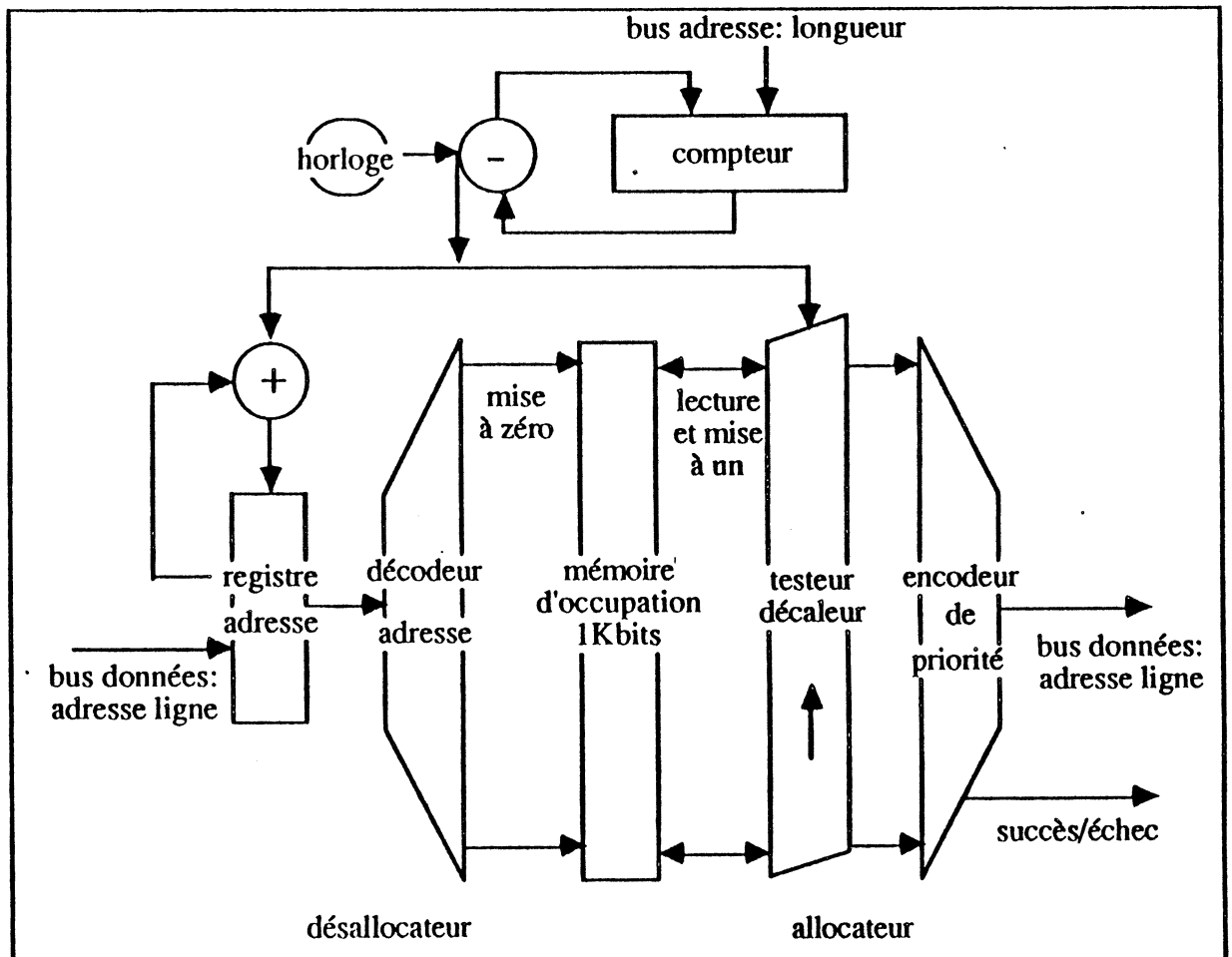


figure III.13 : architecture interne du gestionnaire d'occupation.

Une **mémoire vive** de 1Kbits mémorise l'occupation (1) ou l'inoccupation (0) de la ligne de même adresse dans la mémoire.

Un **compteur** dix bits et une horloge interne permettent d'obtenir un signal d'itération contenant un nombre de cycles égal à la longueur demandée.

La partie allocation utilise un **décaleur-testeur** de 1024 cellules. Ces cellules sont initialisées avec la valeur du bit d'occupation correspondant. A chaque cycle, la nouvelle valeur

de chaque cellule est égale au OU logique de la valeur précédente de la cellule suivante et du bit d'occupation. A la fin du signal d'itération, le premier zéro apparaissant en sortie du décaleur-testeur indique la première ligne d'un bloc allouable. Un **encodeur de priorité** calcule cette adresse pendant qu'une écriture de 1 dans la mémoire d'occupation s'effectue sur la totalité du bloc allouable.

Le mécanisme de désallocation comprend un **registre adresse** initialisé à la première ligne à désallouer et incrémenté à chaque cycle du signal d'itération. Un décodeur d'adresse permet l'écriture de 0 dans la mémoire d'occupation.

III. 2. 2. Réalisation d'un processeur spécialisé.

L'intérêt d'un dessin spécialisé (figure III.14) est de permettre le traitement des trois champs d'un descripteur en parallèle. La partie opérative de ce processeur se divisera en quatre parties.

La **partie adresse** possède quelques registres 16 bits, une partie opérative 16 bits réalisant l'incrémentation et la décrémentation. Cette partie communique en lecture avec le bus adresse locale du module et en lecture-écriture avec le champ valeur du bus données.

La **partie données** possède quelques registres 32 bits et une partie opérative travaillant soit sur les 16 bits du champ valeur soit sur les 32 bits. Elle réalise les comparaisons et les opérations entières. Cette partie communique en lecture-écriture avec la totalité ou le champ valeur du bus données.

La **partie arité** dispose de quelques registres et d'une partie opérative 8 bits. Elle réalise surtout des opérations additives entières et des comparaisons. Elle communique avec le champ arité du bus données.

La **partie étiquette** dispose d'une partie opérative réalisant les comparaisons et les décodages. Le résultat de ces opérations est transmis à la partie contrôle du processeur. Cette partie prend les étiquettes sur le champ étiquettes du bus données. Quelques étiquettes peuvent être mémorisées comme constantes (par exemple variable liée) afin d'être injectées directement sur le bus données lors de la fabrication de descripteurs.

La **partie contrôle** d'un tel processeur exécute les algorithmes écrits pour la simulation de la machine (cf livre quatrième la simulation du module).

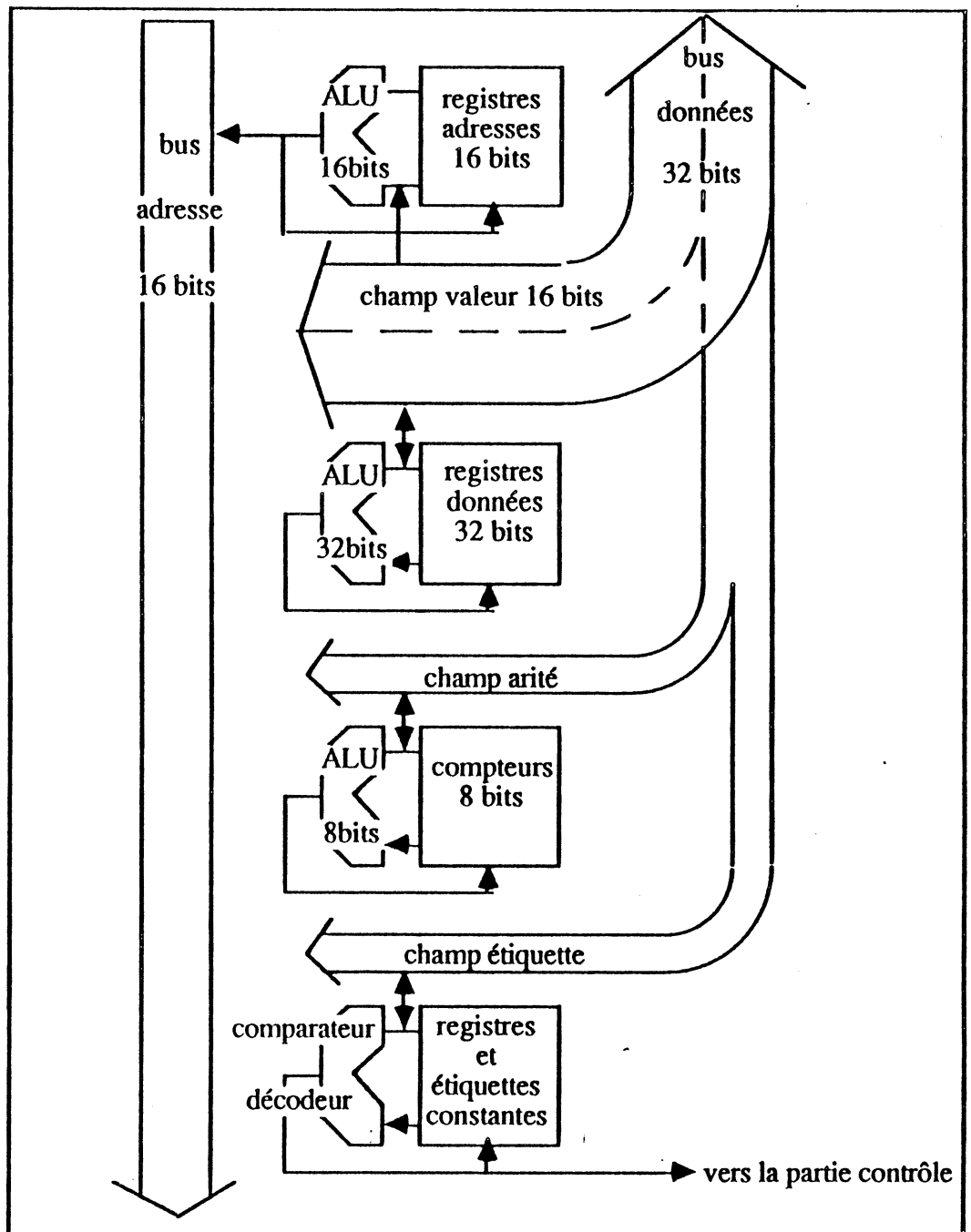


figure III.14 : architecture de la partie opérative du processeur.

III. 2. 3. Réalisation d'un brouillon spécialisé.

Le brouillon se divise en deux parties, les tableaux et les piles (figure III.15).

Le tableau des variables (256 x 32 bits) et le tableau des inconnues (256 x 32 bits) sont adressés directement par le bus adresse. Ils communiquent en lecture et en écriture avec le bus données. Un dispositif de remise à zéro permet le changement de contexte à chaque changement de nœud processus actif.

Une mémoire vive de 512 x 32 bits est organisée en double piles allant à la rencontre l'une de

l'autre. Elle est adressée par deux registres de haut de pile. Ces registres peuvent recevoir leur valeur directement du bus adresse ou d'une boîte à opération entière 9 bits qui la calcule à partir de la valeur précédente, ils sont également réinitialisés à chaque changement de contexte. De plus, l'une des piles sert de piles opératives lors de l'évaluation, elle est reliée à une unité arithmétique et logique capable de travailler dans tous les formats de LAIOS (entier 16 ou 48 bits, flottant 48 ou 80 bits).

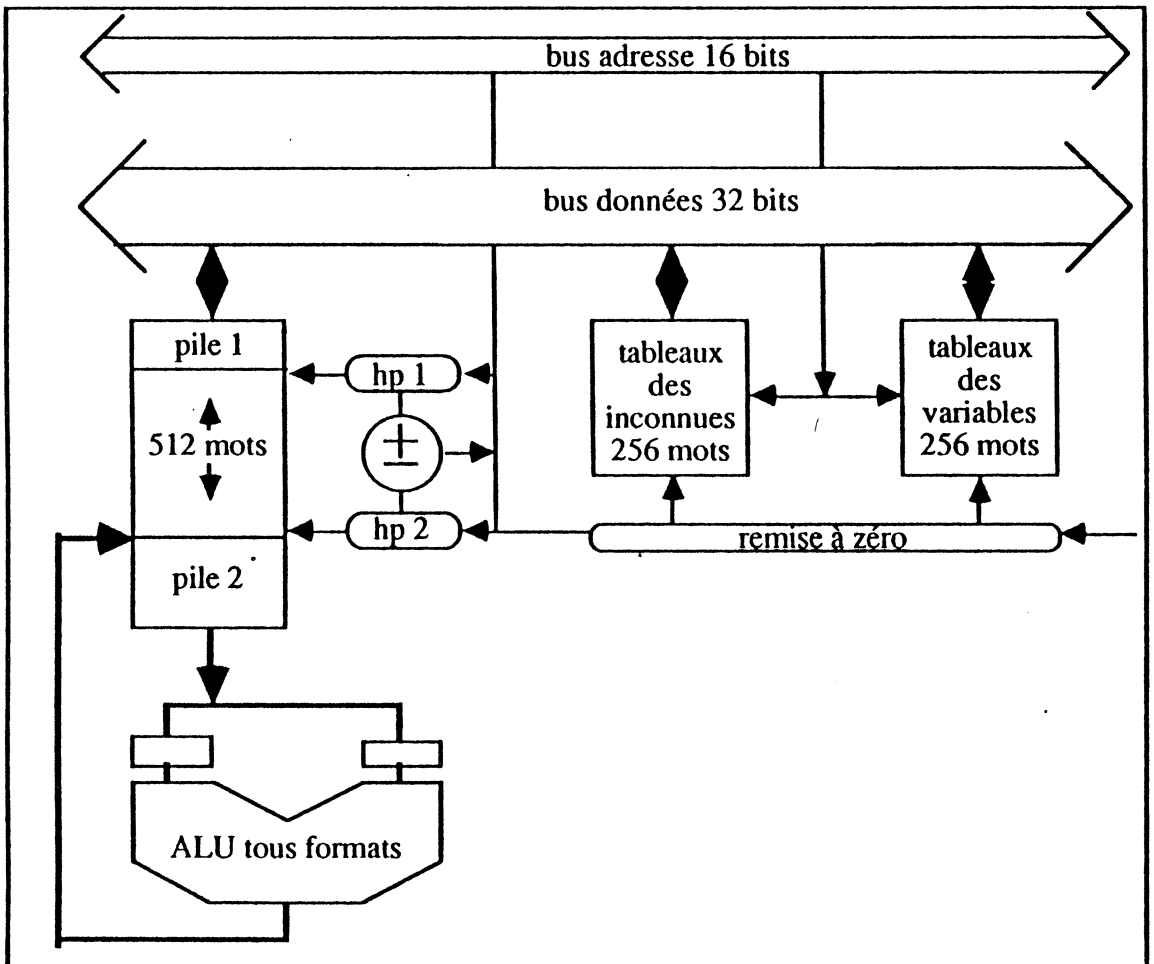


figure III.15 : architecture du brouillon.

LIVRE QUATRIEME :

LES SIMULATIONS.

CHAPITRE I.
LA REPARTITION DE LA CHARGE.

Ce problème a déjà été évoqué dans les livres précédents. L'architecture de LAIOS repose en partie sur ce qui n'était jusqu'à présent qu'un postulat, le principe de localité. Les simulations présentées dans ce chapitre ont pour but d'en étudier la validité.

I. 1. LES DIVERSES SOLUTIONS.

La répartition de la charge a pour objectif d'obtenir le rendement maximum d'un multiprocesseur. Il faut donc éviter qu'un processeur soit inactif lorsque d'autres ont plusieurs tâches en attente. Le problème est compliqué par la prise en compte du temps de transmission entre processeurs qui fait que la répartition la plus égalitaire n'est pas toujours la plus optimale.

I. 1. 1. La répartition statique.

L'essor des périphériques intelligents (gestionnaires de bases de données, gestionnaires de communication, gestionnaires d'écran...) permet une répartition **cablée** du travail. Il revient aux architectes d'éviter les goulots d'étranglement et d'assurer un fonctionnement harmonieux de la machine sur l'ensemble des applications auxquelles elle est destinée. Toutefois, cette répartition est figée de façon définitive et ne peut être que le résultat d'un compromis.

Avec les réseaux de processeurs dans lesquels chaque unité d'exécution est semblable aux autres et peut prendre en charge n'importe quelle tâche de l'application en cours, l'allocation des tâches sur les unités d'exécution n'est pas déterminée à priori, par l'architecture.

Cette allocation peut être réalisée par le **programmeur lui-même** ou lors de la **compilation** du programme [PAZ&_87]. C'est le fonctionnement retenu pour les réseaux de TRANSPUTER [WHIT_85]. P.Kacsuk propose une interprétation de PROLOG sur un réseau rectangulaire de processeurs [SYRE_85].

L'intérêt d'une répartition statique est la possibilité de dégager un maximum de parallélisme, de synchroniser les tâches de manière optimale et surtout de minimiser le routage des messages. Cependant cette répartition se concilie mal avec une création dynamique de processus qui est par nature difficilement prévisible. Elle est inefficace sur les programmes récursifs car elle surcharge localement les unités qui réalisent les appels successifs.

I. 1. 2. La répartition dynamique centralisée.

Lorsque le modèle d'exécution fait principalement appel à la création dynamique de processus, il est généralement préférable d'implanter un algorithme de répartition sur le réseau lui-même. Le modèle centralisé consiste à définir une unité **maître** de la répartition.

Cette unité peut être un véritable **allocateur** [HONG_86] qui recueille toutes les créations et les redistribue sur les autres unités qui sont considérées par le programme comme des unités esclaves. Grossièrement, il est clair qu'un tel système fonctionne de manière optimale lorsque le produit du temps moyen d'allocation d'une tâche par le nombre d'unités d'exécution est égal au temps moyen d'exécution d'une tâche. Ce système sera donc efficace sur des réseaux de quelques dizaines de processeurs exécutant des tâches de durée importante et assez largement

indépendantes les unes des autres.

Lorsque les tâches sont plus brèves et plus dépendantes, il convient de diminuer le travail de l'allocateur. Le modèle de **pool de jetons** proposé par Ciepielewski [CIEP_84] permet de réduire pratiquement l'unité maître de la répartition à une file d'attente distribuant et engrangeant les jetons à la demande. Chaque jeton contient toutes les informations utiles pour la tâche à exécuter.

I. 1. 3. La répartition dynamique décentralisée.

Dans les réseaux ayant un grand nombre d'unités d'exécution, la gestion centralisée devient un goulot d'étranglement qui en ralentit le fonctionnement. Il faut dès lors distribuer le travail de répartition sur toutes les unités. La difficulté provient du fait qu'il est impossible (ce serait beaucoup trop coûteux) que chaque unité ait une connaissance de la distribution de la charge sur la totalité du réseau. Dès lors, les programmes ne pourront être que des heuristiques s'approchant de l'optimum.

Peu de propositions ont encore vu le jour dans ce domaine. Une idée séduisante consiste à chercher à ce que le processus "s'écoule" sur le réseau depuis son point de naissance selon le **gradient de plus faible charge** jusqu'à un minimum local en une unité qui l'exécutera. Cet algorithme ne demande à chaque unité que de connaitre l'état de ses voisins et de disposer d'un processeur de transmission capable de garder ou de faire circuler l'information. Il est probable que la répartition de la charge obtenue avec un tel algorithme soit régulière mais la circulation de données risque d'être importante.

I. 1. 4. L'algorithme de LAIOS.

Dans LAIOS, le déroulement dynamique de l'arbre conduit à une répartition dynamique; il n'existe pas de module maître de la répartition de la charge, cette répartition est décentralisée. Par contre, le principe de localité qui veut que des processus parents proches soient implantés sur des modules physiquement voisins limite les possibilités de répartition.

Chaque module possède donc sa propre file d'attente de processus désirés (à créer). Il possède également un processus mère, processus système résidant. Lorsqu'il est activé, le processus mère commence par élire le module le plus chargé entre le sien propre et ses six voisins. Il prend alors le premier processus désiré par le module élu et tente de l'exécuter.

Cette implantation peut échouer dans le cas du ET pipe line où les contraintes de voisinages concernent non seulement le père mais aussi le frère ainé. En faisant l'hypothèse qu'un processus a une probabilité uniforme d'être implanté sur chacun des sept modules (celui de son père et ses six voisins), la probabilité d'échec pour l'implantation d'un processus ayant un frère ainé n'est que de 18/49 (produit de 6/7, probabilité que le module du frère ainé soit voisin de celui du père, et de 3/7, probabilité que le module implanteur ne soit pas voisin de celui du frère ainé). Avec l'hypothèse qu'un processus sur quatre possède un frère ainé, la probabilité d'échec d'implantation d'un processus n'est que de (1/4)*(18/49) soit environ 1 pour 10.

Le point délicat est le calcul de la charge d'un module car plusieurs paramètres peuvent entrer en jeu. Le nombre de processus implantés et le nombre de processus désirés sont les paramètres les plus évidents. Il est bon de tenir également compte de l'encombrement mémoire pour éviter des blocages prématurés.

I. 2. LES PROGRAMMES DE SIMULATIONS.

Ces programmes sont écrits en MACPASCAL qui présente l'intérêt de disposer d'un graphisme intégré et d'un environnement de programmation ergonomique.

I. 2. 1. Le réseau.

La **topologie** du réseau est modélisée par un tableau de pointeurs à deux dimensions, les coordonnées étant prises sur deux axes du réseau comme il a été exposé dans le livre précédent. Les deux paramètres utilisés sont le diamètre **P** et le rayon N avec la relation P=2N+1.

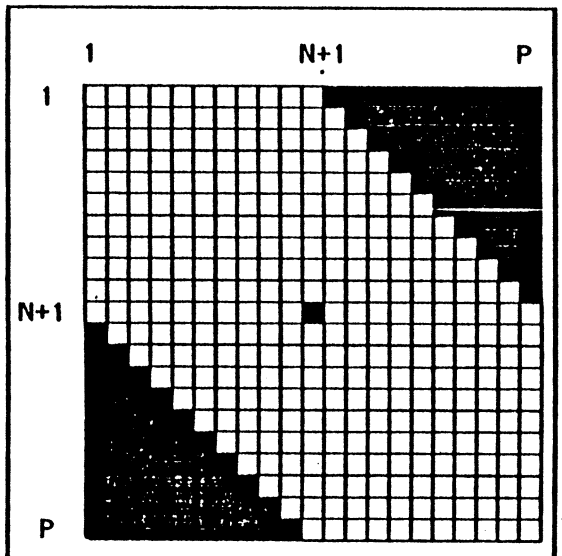


figure IV.1 : le tableau des modules.

Chaque module opératoire est un enregistrement contenant sa charge et ses paramètres de dessin à l'écran. Les cases inutilisées du tableau (en dehors de l'hexagone, module central mémoire de masse et des modules opératoires en panne) restent vides (nil).

La charge de chaque module est égale au nombre de processus désirés.

I. 2. 2. L'activité.

Le programme consiste en une boucle principale dans laquelle est effectué un **tirage aléatoire** des coordonnées du module dont le processus mère est activé. Cette activité se limite à trois actions. La première consiste à élire le module le plus chargé entre lui-même et ses voisins. La seconde consiste a ôter un processus au module élu s'il en possède au moins un. La troisième consiste à tirer aléatoirement le nombre de processus créés par celui qui vient d'être pris en charge.

Le tirage du nombre de fils créés suit une loi uniforme entre zéro et un nombre maximum de fils (**nbfi**) qui est un paramètre important du programme. De la sorte, le nombre moyen de fils est égal à la moitié de nbfi.

L'initialisation consiste à fournir une charge à un module périphérique (ou plusieurs lors de la simulation du fonctionnement multi-utilisateurs). Cette charge ne doit pas être trop faible (typiquement 5) pour que les premiers tirages aléatoires du nombre de fils n'arrêtent pas la simulation prématurément.

Ce modèle est criticable car assez rudimentaire. Il ne tient compte que du nombre de processus désirés pour la charge. Il limite l'activité des processus à la création des fils. Il suppose que le nombre de fils suit une loi uniforme dans l'intervalle [0,nbfi]. Il modélise le temps passé entre deux activations du processus mère par le temps qui s'écoule entre deux tirages aléatoires. Or, il est évident qu'un module peu chargé activera plus fréquement son processus mère qu'un module lourdement chargé; ce phénomène d'autorégulation ne pouvant toutefois qu'améliorer les résultats. Enfin, il ne simule que la phase d'expansion de l'arbre de recherche qu'il suppose infini, mais c'est la seule phase dans laquelle la répartition de la charge est critique. En dépit de ces remarques, ce modèle permet déjà d'estimer les qualités et les défauts de l'algorithme testé.

I. 2. 3. Les résultats recherchés.

Le but de cette simulation est de mettre en évidence le comportement de l'algorithme de répartition de la charge et l'influence de la taille du réseau (paramètre: le rayon) et du programme (paramètre: le nombre maximum de fils créés). La possibilité de rendre inactif un ou plusieurs modules est mise à profit pour observer la tolérance aux pannes.

La présentation des résultats peut prendre l'une des deux formes suivantes: une forme **observable** et une forme **mesurable**. La présentation observable consiste en une représentation symbolique des modules et de leur charge qui est dessinée sur l'écran et évolue au cours de la simulation. Quelques instantanés de cette évolution seront présentés dans les pages suivantes. La présentation mesurable consiste à calculer à chaque instant la charge moyenne des modules, leur écart avec la moyenne et l'écart-type ; la courbe de l'écart-type en fonction de la moyenne est alors dessinée.

I. 3. SIMULATION OBSERVABLE.

Cette simulation consiste en l'établissement d'une simulation de référence, puis, à partir de cette référence, à faire varier un paramètre à la fois, pour étudier son influence.

I. 3. 1. La référence.

Le rayon du réseau est de 10 modules soit un réseau de 330 modules opératoires. Le nombre maximum de fils est fixé à 5 ce qui établit le nombre moyen à 2,5. Le nombre maximum d'itérations est de 10000.

La palette de grisé utilisable dans MACPASCAL ne contient que cinq nuances qui sont utilisées pour symboliser la charge comme suit:

- blanc : 0,
- gris clair : 1 à nbfi,
- gris : nbfi+1 à 2nbfi,
- gris foncé : 2nbfi+1 à 3nbfi,
- noir : au delà de 3nbfi.

Le premier instantané représente la situation initiale (figure IV.2).

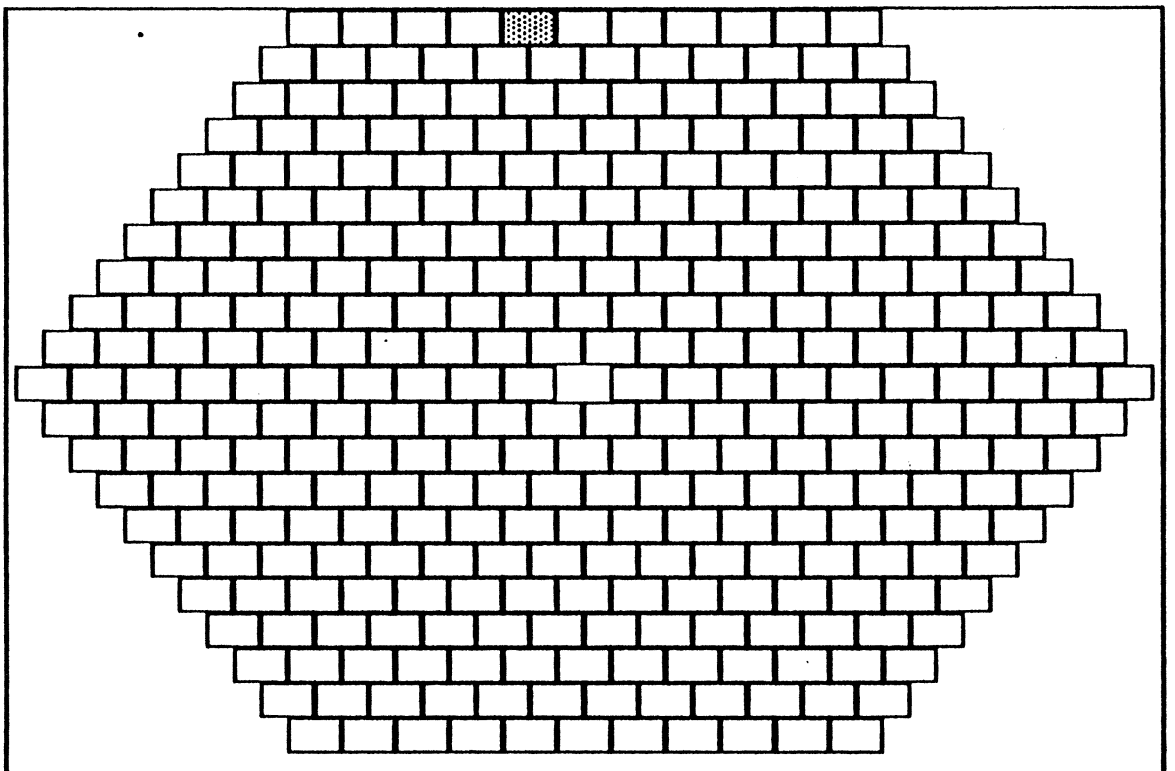


figure IV.2 : situation initiale de référence.

Le second instantané (figure IV.3) est pris au bout de 2500 itérations, il montre que la moitié des modules est déjà au travail sans qu'aucun ne soit encore lourdement chargé.

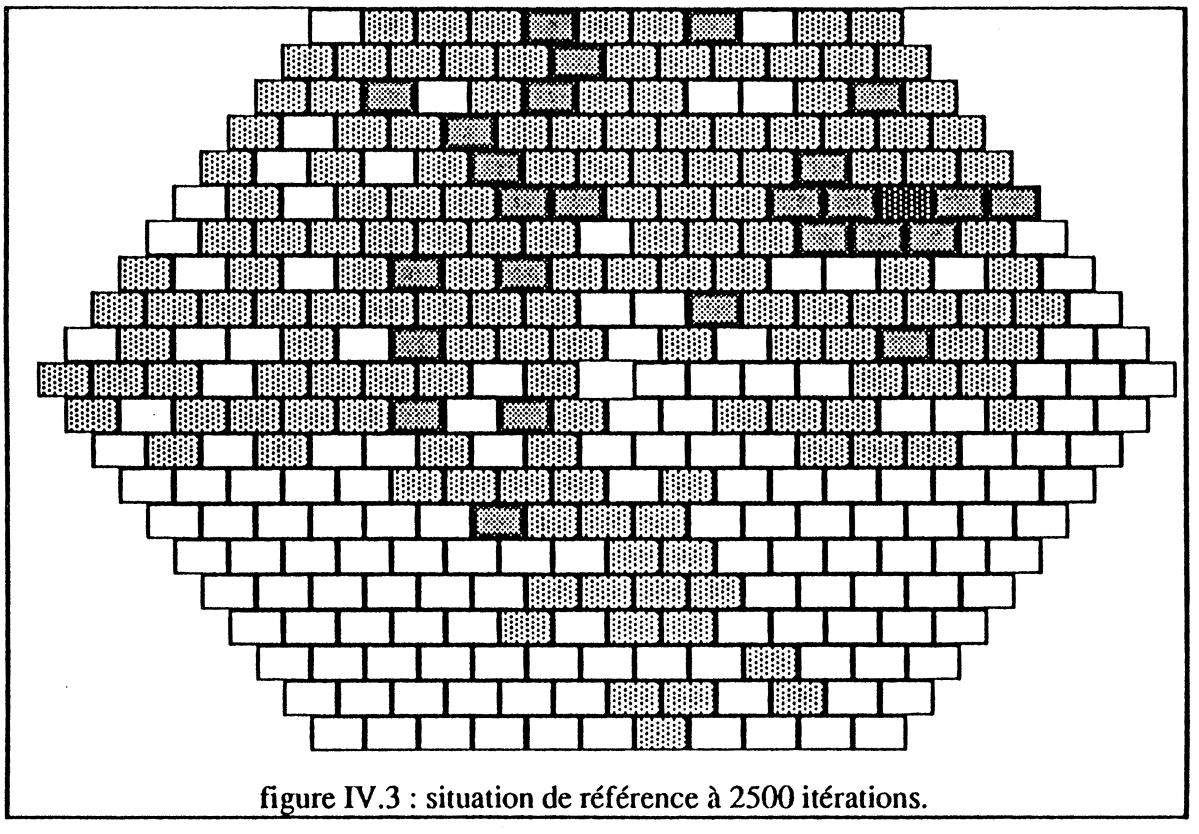


figure IV.3 : situation de référence à 2500 itérations.

Le troisième instantané (figure IV.4) représente la situation à mi-parcours (5000 itérations); la répartition est assez uniforme, les quelques modules saturés (noirs) sont isolés, leur saturation n'est que provisoire.

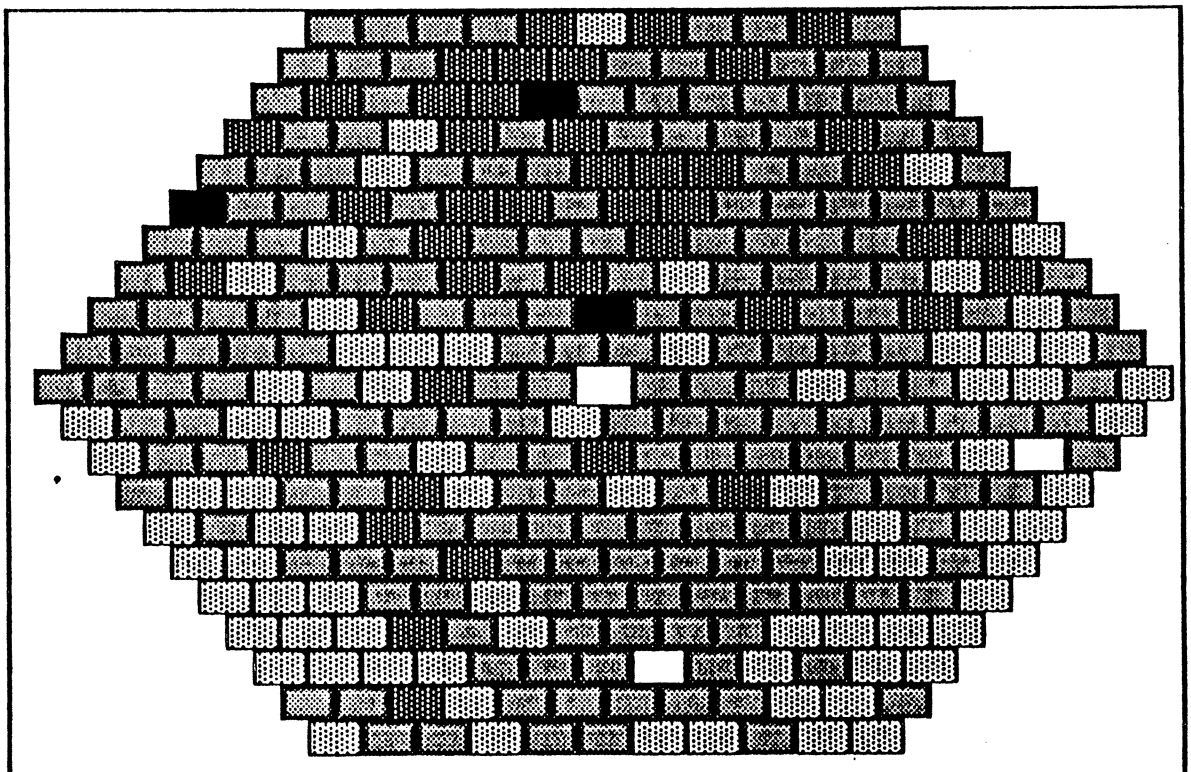


figure IV.4 : situation de référence à 5000 itérations.

Le quatrième instantané (figure IV.5) est pris au bout de 7500 itérations, l'ensemble des modules est lourdement chargé et des zones de saturations commencent à apparaitre de façon permahente.

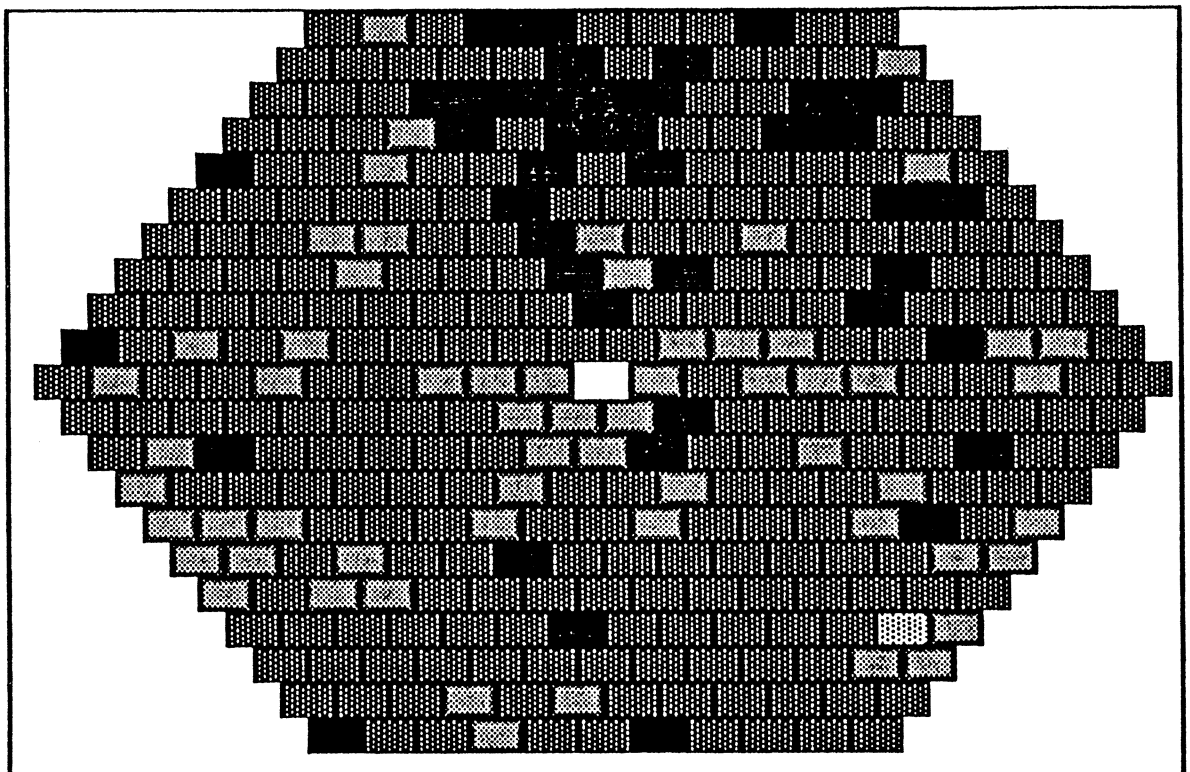


figure IV.5 : situation de référence à 7500 itérations.

Le dernier instantané (figure IV.6) représente la situation à la fin des 10000 itérations, l'ensemble du réseau étant uniformément saturé.

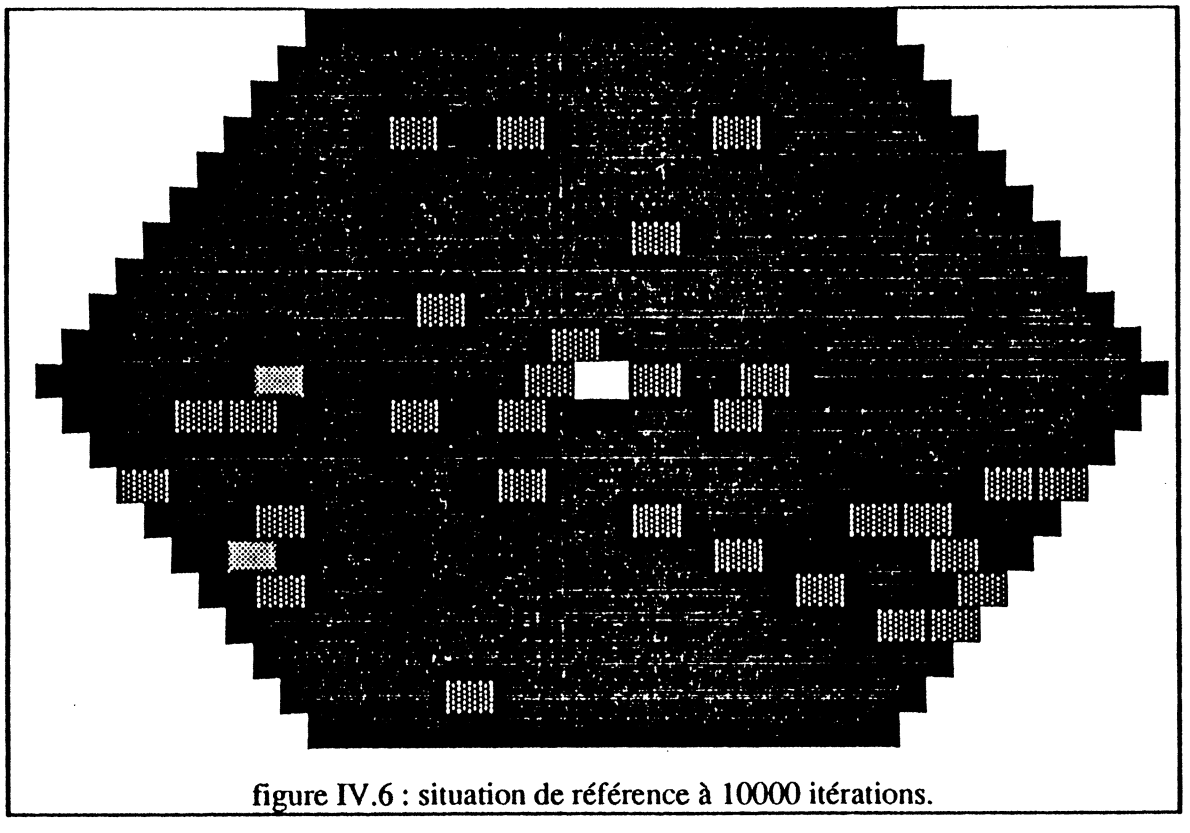


figure IV.6 : situation de référence à 10000 itérations.

I. 3. 2. Variations du nombre moyen de fils.

Le paramètre nbfi va être modifié mais les autres paramètres ne le seront pas. La modification de nbfi entraine celle des nuances de grisé qui lui est liée de façon à garder une sensibilité constante.

Avec nbfi égal à 4, la croissance de la charge est lente et la répartition est très régulière comme le montre l'instantané pris à 7500 itérations (figure IV.7).

La disparité de charge entre la droite et la gauche s'explique par l'algorithme d'élection utilisé pour determiner le module le plus chargé. En cas d'égalité de charge, l'élu est dans l'ordre de priorité: le module lui-même, puis celui de droite et les suivants en tournant dans le sens des aiguilles d'une montre. En inversant cet ordre, la répartition obtenue présente la tendance symétrique. Dans la réalité, la mesure de charge étant plus fine, les cas d'égalité devraient se faire suffisament rares pour qu'il n'y ait pas besoin de se préoccuper de cet effet parasite. S'il persistait, il serait facile de l'éliminer en diversifiant l'ordre d'examen sur les modules.

Si nbfi est inférieur à 4, la croissance devient tellement lente qu'elle ne parvient plus à couvrir le réseau en un temps raisonnable et de nombreuses simulations s'arrêtent après quelques itérations par disparition totale de la charge.

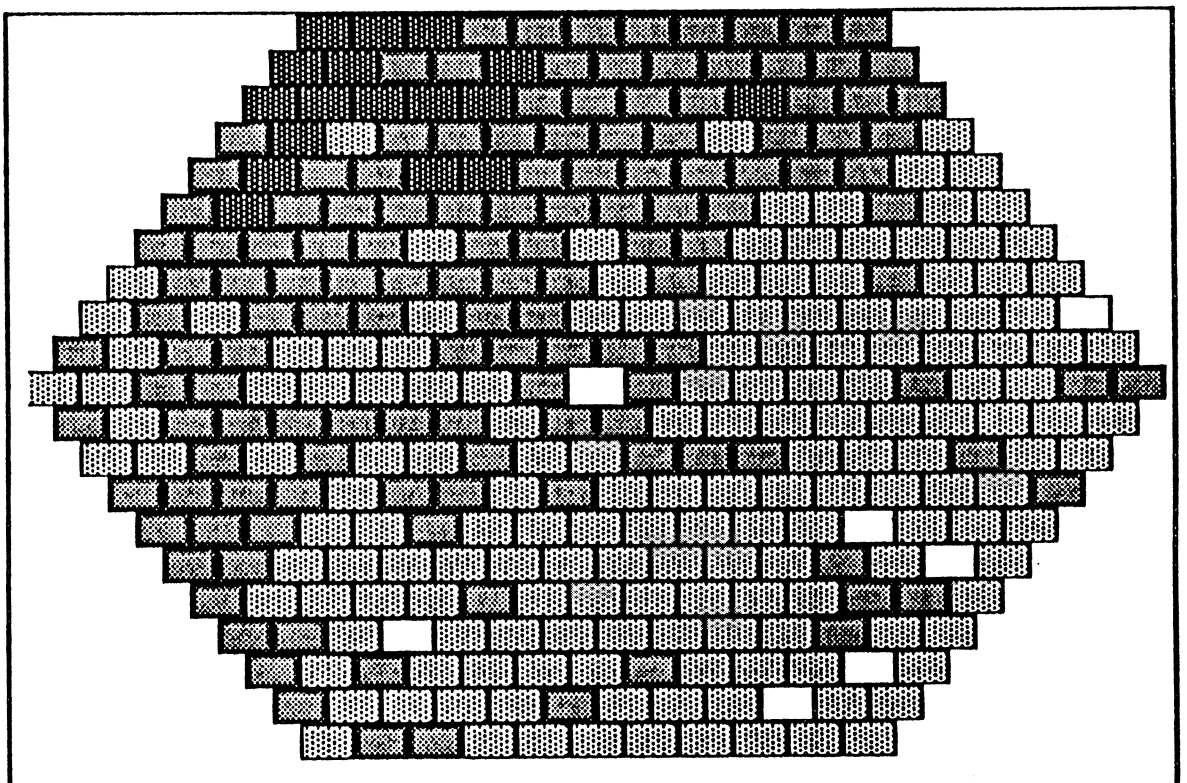


figure IV.7 : situation à 7500 itérations lorsque nbfi = 4.

Au contraire, lorsque la moyenne du nombre de fils créés augmente, la répartition a tendance à devenir plus inégale comme le montre l'instantané (figure IV.8) pris à 5000 itérations avec nbfi valant 8.

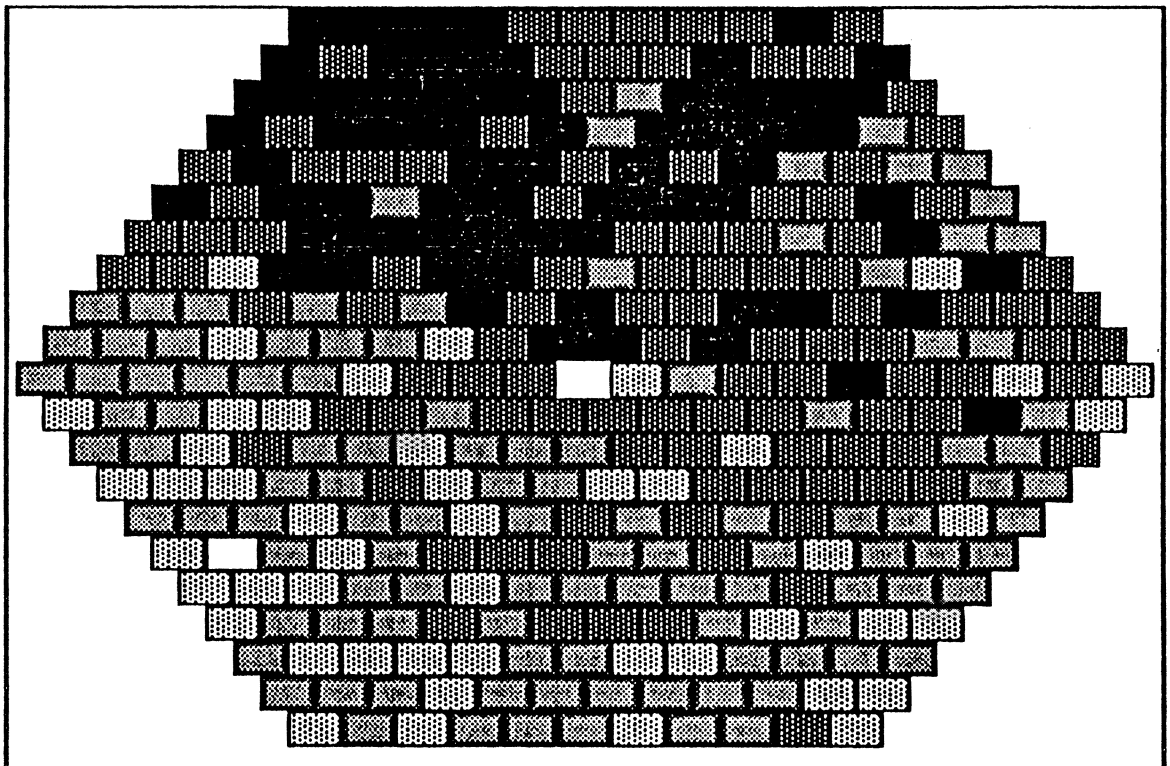


figure IV.8 : situation à 5000 itérations lorsque nbfi = 8.

I. 3. 3. Variations sur la taille du réseau.

En choisissant un réseau de rayon 5 (90 modules), la montée en charge du réseau est beaucoup plus rapide comme le montrent les deux instantanés ci-dessous. Afin d'éviter une saturation rapide du réseau, il a fallu diviser la sensibilité du coloriage par deux:

- blanc : 0;
- gris clair : 1 à 2nbfi;
- gris : 2nbfi+1 à 4nbfi;
- gris foncé : 4nbfi+1 à 6nbfi;
- noir : au-delà de 6nbfi.

Les deux instantanés (figures IV.9) pris respectivement à 2500 puis à 5000 itérations mettent en évidence une répartition quasi uniforme de la charge.

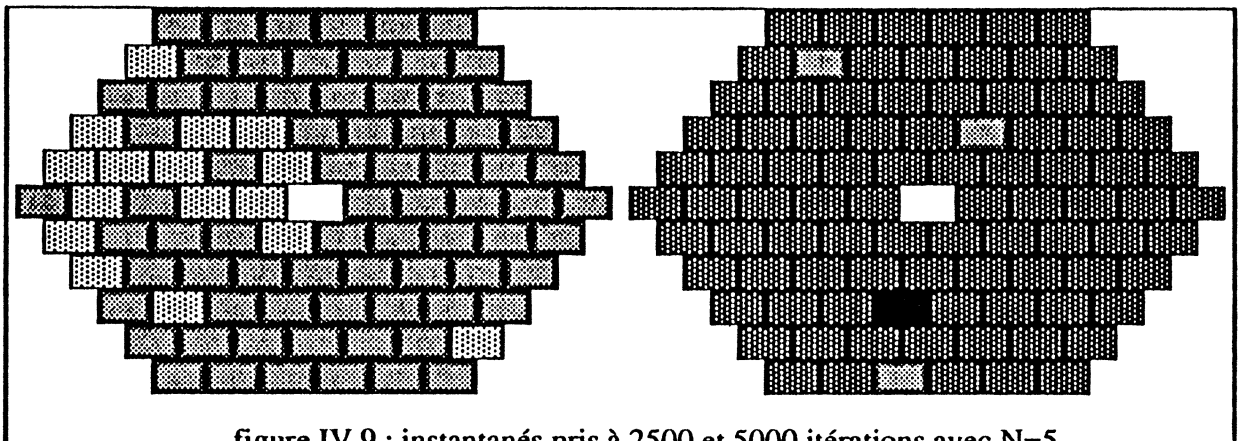


figure IV.9 : instantanés pris à 2500 et 5000 itérations avec N=5.

Au contraire, en choisissant un réseau de grande taille, avec un rayon de 20 soit 1260 modules, sous les mêmes conditions pour les autres paramètres que dans la situation de référence, la répartition de la charge sur le réseau se présente au bout de 30000 itérations (figure IV.10) sous la forme d'ondes concentriques autour de la case de départ (en haut au milieu) en dégradés de gris au fur et à mesure que l'on s'éloigne de cette case. Le grand nombre d'itérations met en évidence la capacité du réseau à traiter des problèmes décomposés en plusieurs milliers de processus.

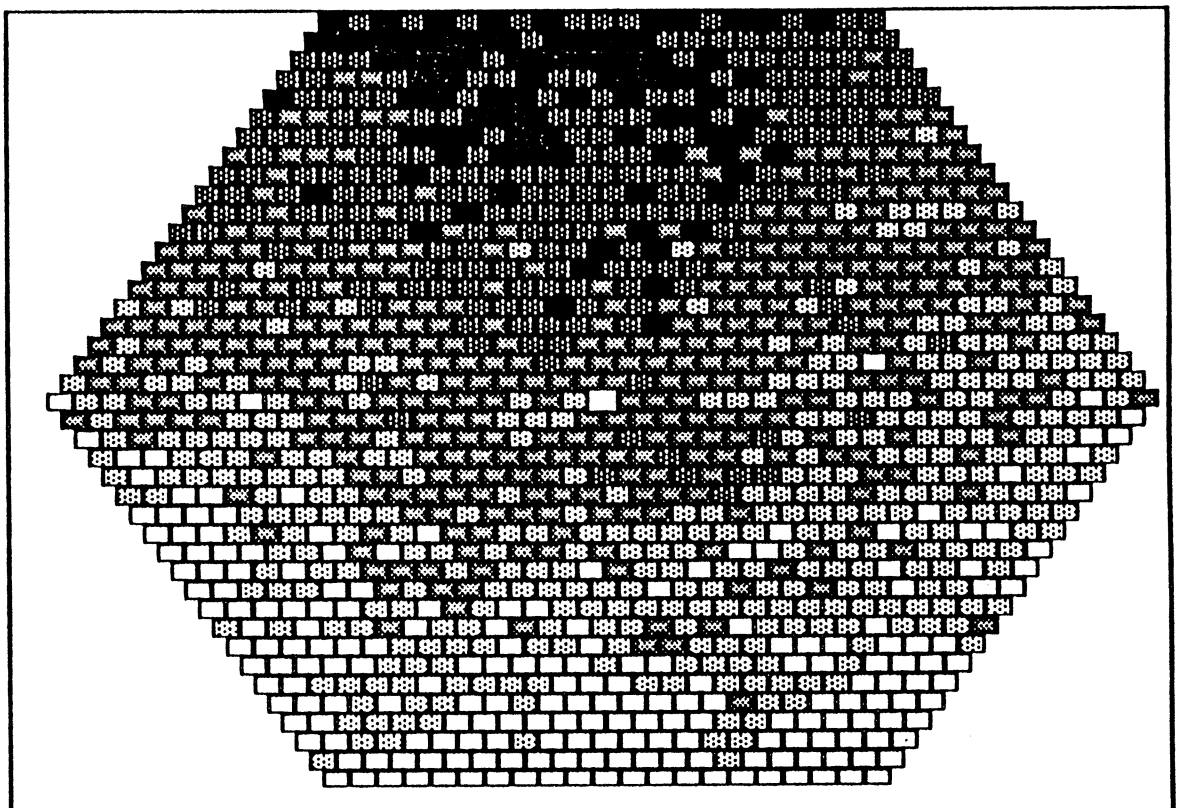


figure IV.10 : instantané pris à 30000 itérations avec N=20.

I. 3. 4. Le fonctionnement multi-utilisateurs.

A partir de la situation de référence, le fonctionnement multi-utilisateurs est simulé en chargeant plusieurs modules périphériques au lieu d'un seul.

En répartissant les charges sur la périphérie, une au milieu de chaque coté de l'hexagone la charge se répartit de façon très homogène comme le montre l'instantané pris au bout de 5000 itérations (figure IV.11).

En plaçant les six utilisateurs sur le même coté, la répartition est moins uniforme mais surtout, il se produit un phénomène de gène mutuelle qui fait que le déroulement des arbres est moins rapide que dans le cas précédent, l'instantané ayant été pris au bout de 7500 itérations (figure IV.12).

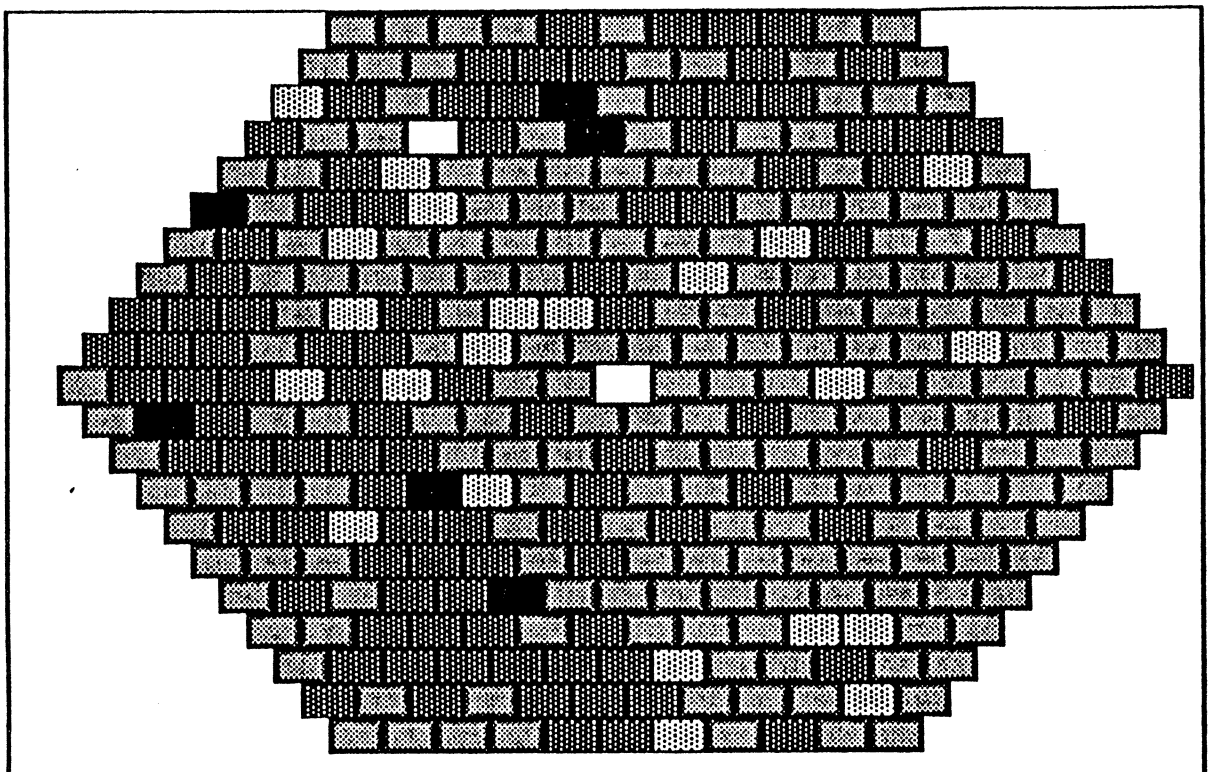


figure IV.11 : instantané à 5000 itérations avec six utilisateurs répartis.

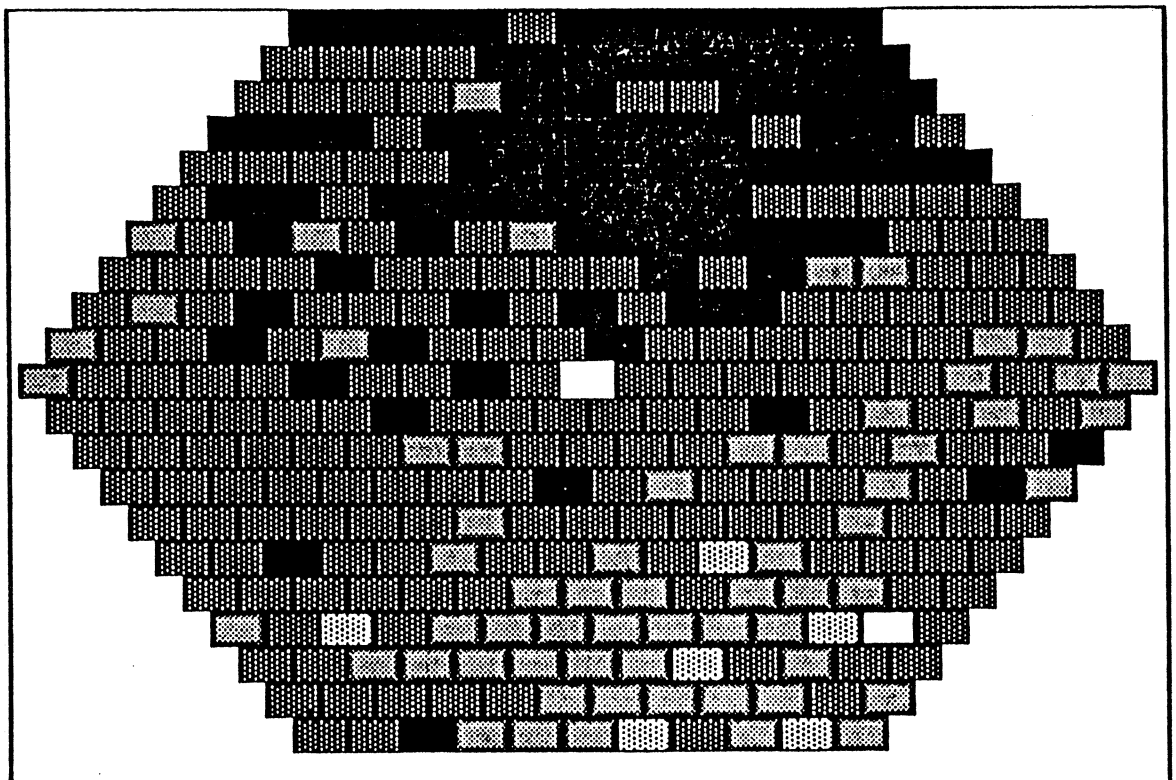


figure IV.12 : instantané à 7500 itérations avec six utilisateurs voisins.

I. 3. 4. La tolérance aux pannes.

La situation de départ est toujours la situation de référence dans laquelle quelques modules sont déclarés en panne.

La présence d'un ou de quelques modules en panne répartis sur le réseau n'influence pas de façon perceptible le déroulement et la répartition de la charge.

Une barrière d'une demi ligne n'est pas une gène insurmontable et même si elle ralentit le déroulement de l'arbre, elle ne provoque pas une grande perturbation dans la répartition de la charge comme le montre l'instantané à 5000 itérations (figure IV.13).

Une diagonale entière en panne provoque un ralentissement perceptible du déroulement et une répartition assez médiocre comme en fait preuve l'instantané pris à 7500 itérations (figure IV.14).

Enfin, le cas pathologique du goulot d'étranglement perturbe profondément le fonctionnement du réseau puisque au bout de 10000 itérations seuls les modules dans la nasse saturent alors que d'autres n'ont pas ou à peine commencé à travailler (figure IV.15).

En conclusion, il apparait que la tolérance aux pannes d'un tel algorithme est assez remarquable; la répartition n'est notablement altérée que sur des cas pathologiques très improbables. Par contre, et cela est assez naturel, il y a ralentissement du déroulement dès que le nombre de modules en panne dépasse les quelques unités.

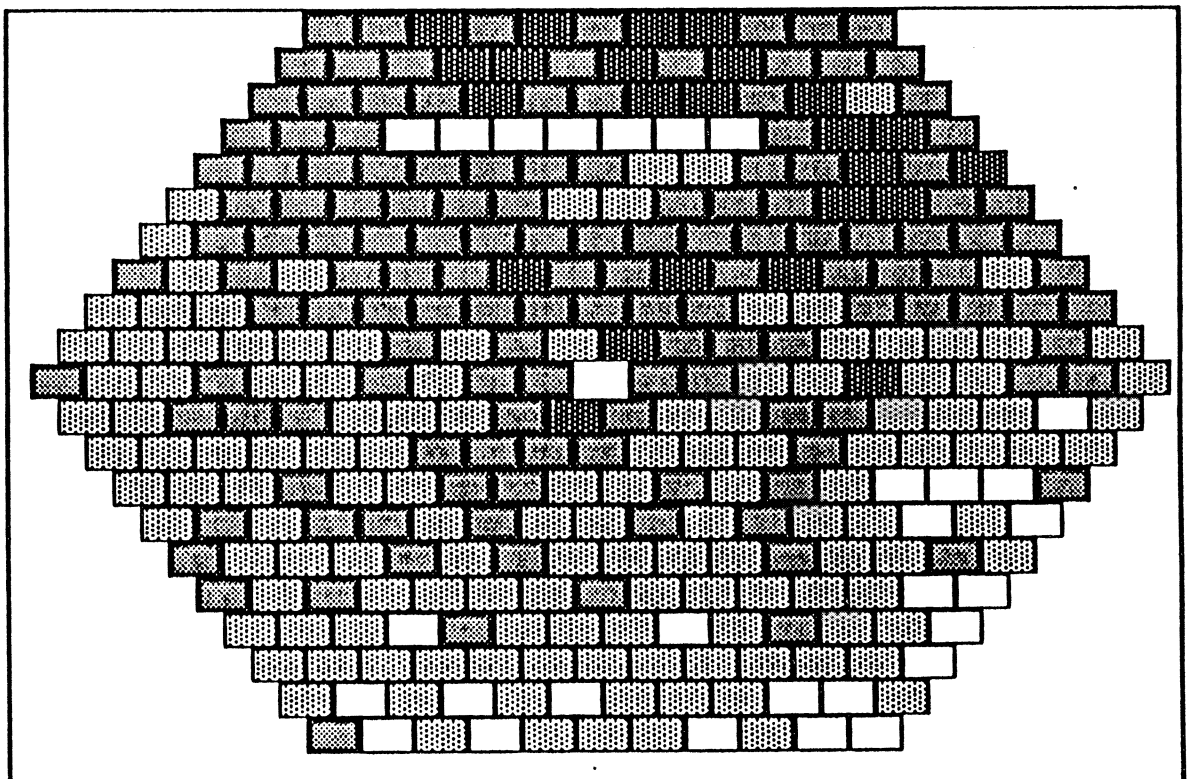


figure IV.13 : instantané à 5000 itérations avec une demi ligne en panne.

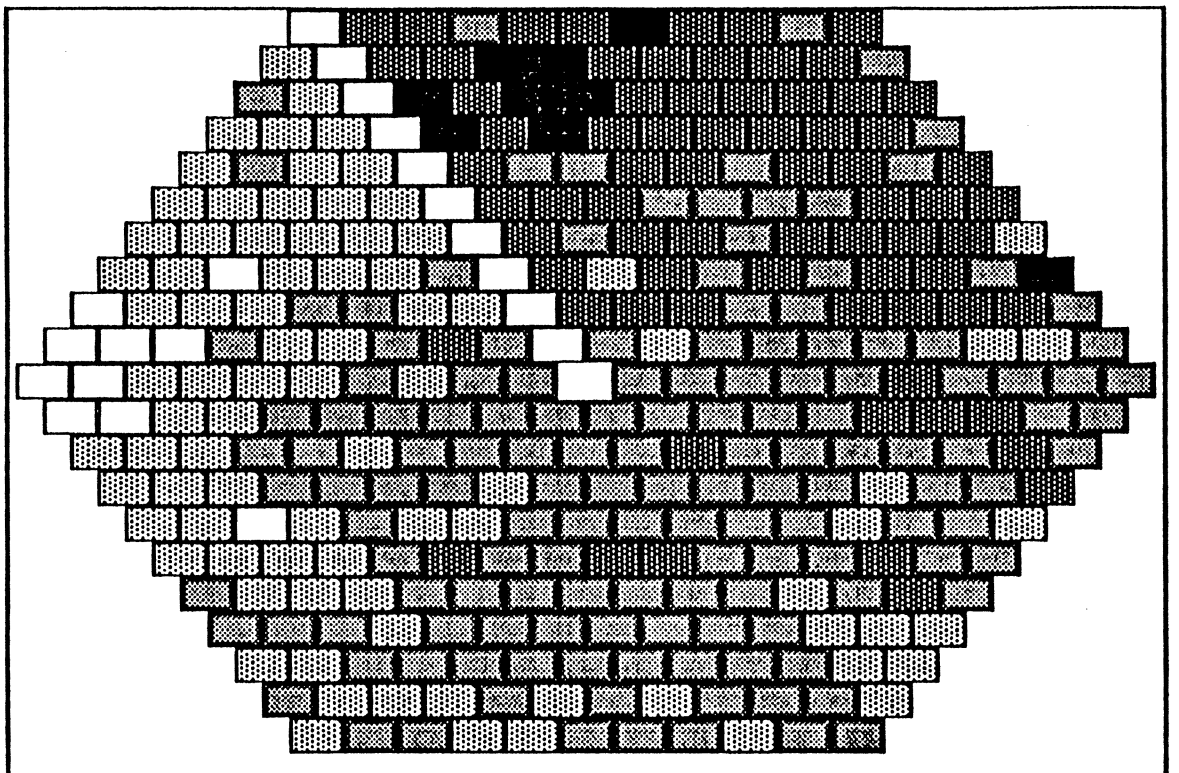


figure IV.14 : instantané à 7500 itérations avec une diagonale en panne.

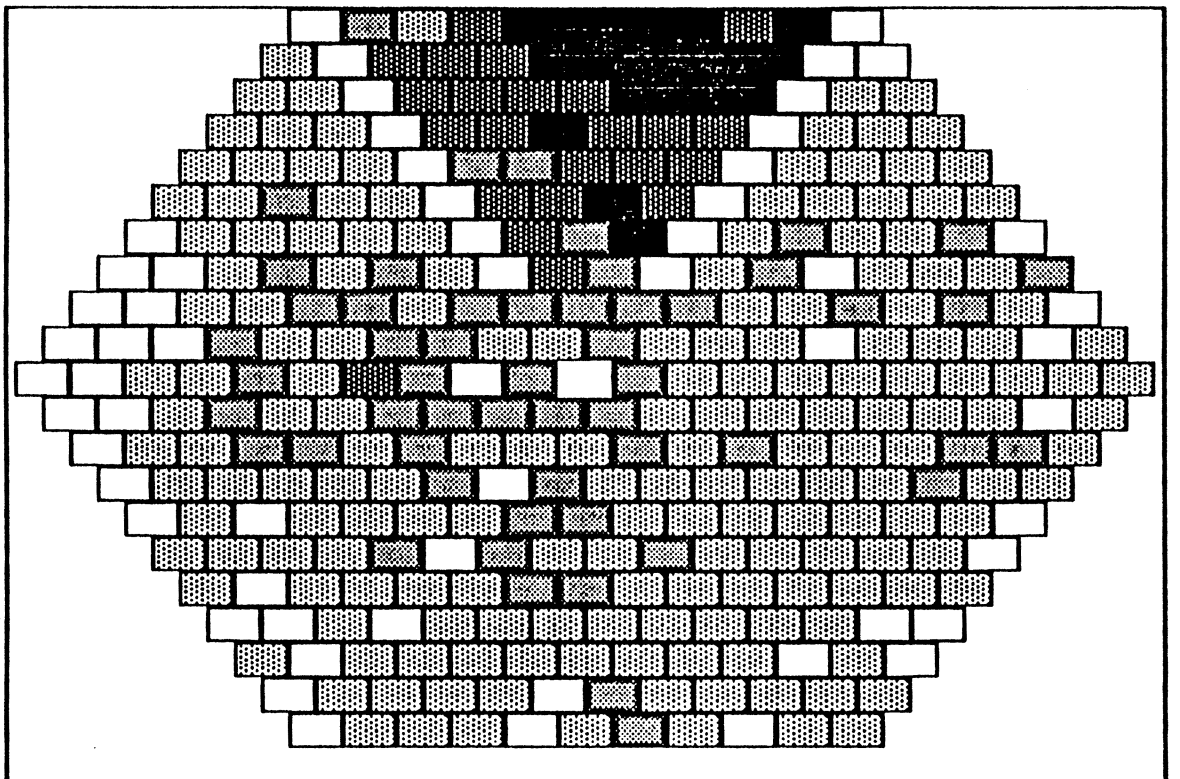


figure IV.15 : instantané à 10000 itérations avec un goulot d'étranglement.

I. 4. SIMULATION MESURABLE.

Les valeurs obtenues dans cette simulation vont permettre une première estimation de la taille maximale des problèmes solubles avec LAIOS.

I. 4. 1. Les courbes.

Le réseau utilisé est encore le réseau de référence choisi au paragraphe précédent. Cinq simulations sont effectuées de 20000 itérations chacunes, en faisant varier le nombre maximum de fils (nbfi) de 4 à 8.

Les courbes dessinées sont obtenues point par point en calculant à chaque itération la charge moyenne du réseau -qui a tendance à croître- et l'écart-type de cette charge sur l'ensemble des modules.

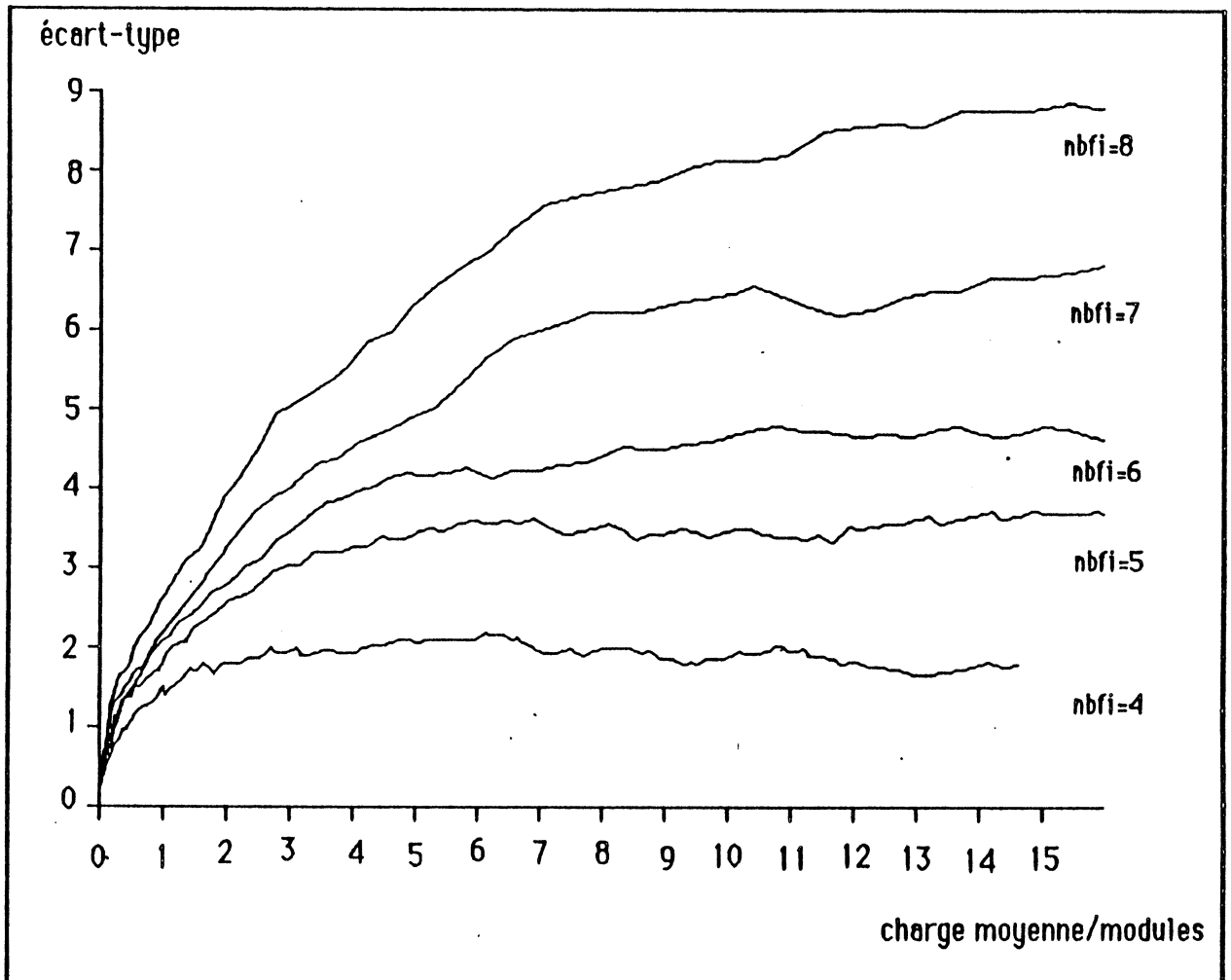


figure IV.16 : courbes de l'écart-type de la charge par module en fonction de la charge moyenne.

I. 4. 2. Interprétation du résultat.

L'écart-type est une grandeur qui traduit la qualité de la répartition, plus cet écart est faible meilleure est la répartition, une répartition parfaite se traduisant par un écart nul. Ces courbes sont naturellement légèrement bruitées par le choix des tirages aléatoires mais la tendance générale de ces courbes est claire et confirme les impressions obtenues lors des simulations observables.

Lorsque nbfi est égal à 4, c'est-à-dire que le nombre moyen de fils est de 2, l'écart-type reste inférieur à la valeur remarquable de 2 et semble même vouloir diminuer légèrement après être passé par un maximum. Ce résultat confirme l'observation faite au paragraphe précédent.

Lorsque nbfi est égal à 5 ou 6, soit un nombre moyen de fils entre 2,5 et 3, l'écart-type tend vers une valeur maximum de l'ordre de 4. Ce résultat est commenté plus loin.

Enfin, lorsque la valeur de nbfi continue d'augmenter, l'écart-type maximum augmente lui aussi mais l'allure de la courbe reste la même et semble tendre asymptotiquement vers une valeur maximale.

Les **statistiques** réalisées par Onai [ONA&_84] sur des programmes PROLOG donnent un nombre de 2,7 comme nombre moyen de fils par processus. Il est d'ailleurs facile de demander au compilateur lorsqu'il transforme les programmes en nœuds modèles de respecter des contraintes afin de rester au voisinage de cette valeur. Une règle ayant un grand nombre de prédicats dans son corps peut être décomposée en plusieurs règles. Un trop grand nombre de clauses ayant le même prédicat de tête pourront être regroupées en plusieurs paquets en choisissant un paramètre pour effectuer une premiere sélection. Il est donc possible de retenir la valeur de 4 comme maximum pour l'écart-type.

Les simulations suivantes montreront qu'une centaine de processus pris en charge par module est un nombre raisonnable pour une taille mémoire de 16Kmots. En admettant qu'un blocage par dépassement de capacité mémoire apparait dès que le nombre de processus pris en charge est supérieur à 100, le réseau peut donc fonctionner jusqu'à **90% de sa capacité** avec un probabilité faible de blocage (probabilité qu'un module ait une charge supérieure de 2,5 fois l'écart-type par rapport à la charge moyenne du réseau).

En conclusion, cet algorithme de répartition de la charge semble bien adapté à l'architecture et au modèle d'exécution choisis. Il allie simplicité de mise en œuvre et efficacité. Bien qu'il faille être prudent et ne pas oublier les simplifications et les hypothèses faites lors de ces simulations, cet algorithme devrait permettre une bonne exploitation des possibilités du réseau et faire preuve de tolérance aux pannes.

CHAPITRE II. SIMULATION DE L'OPERATEUR UNIVERSEL.

Le but poursuivi lors de ces simulations est double. Il s'agit d'abord de **valider le modèle** d'exécution et de s'assurer de l'**adéquation de l'architecture** du module avec ce modèle. Il s'agit aussi de procéder à une première estimation des temps d'exécution de programmes classiques et de les comparer avec ceux consommés par des machines monoprocesseurs de taille comparable au module de LAIOS; un même ordre de grandeur établirait la plausibilité des choix architecturaux.

II. 1. LE PROGRAMME.

Le programme est structuré en huit couches concentriques interprétant chacune une partie d'un des livres précédents.

II. 1. 1. La couche simul.

C'est la couche la plus interne. Elle contient donc les constantes du programme et les définitions des types adresse et descripteur ainsi que les variables globales.

Les constantes du programme sont le **nombre de pages** (16) et le **nombre de mots** par page (127). Pour ce type de simulation, il n'est pas utile d'avoir la capacité maximum par page. Ce sont aussi les temps estimés pour l'exécution avec **100ns** pour les **opérations internes** au processeur et pour les **accès au brouillon, 300ns** pour les **accès** au reste de la **mémoire locale** et **600ns** pour le **mécanisme d'allocation**. Ces temps correspondent aux possibilités actuelles des circuits CMOS VLSI.

Les types sont les **adresses**, les **descripteurs** et enfin les **en-tête de nœuds modèles**. Une adresse a deux champs: le numéro de la page et le numéro de la ligne. Un descripteur a trois champs: l'étiquette, l'arité et la valeur. Un en-tête de nœud modèle est caractérisé par un ensemble de six déplacements donnant accès respectivement à la classe, au nombre de variables externes, au numéro de la première variable interne, au nombre de variables internes, à la table des fils et au bloc des paramètres. Un certain nombre de simplifications ont été effectuées sans conséquences sur la validité des vérifications, les réels n'ont pas été implantés pour diminuer la taille mémoire d'un descripteur qui est un enregistrement Pascal, les en-têtes de nœuds modèles sont tous identiques, certaines variables étant dès lors inutiles ce qui simplifie l'écriture du programme. Les contraintes n'étaient pas encore intégrées dans le langage à l'époque où ces simulations ont été effectuées.

Les variables globales sont le nombre de processus créés, le nombre d'inférences réalisées et le temps d'exécution.

II. 1. 2. La couche machine.

Elle contient les procédures d'**adressages**, de **lecture**, d'**écriture**, d'**incrémentation** et de **comparaison** des adresses. Elle contient aussi le **filtre** de comparaison des **étiquettes** pour l'unification.

II. 1. 3. La couche mémoire.

Elle simule toute l'organisation interne de la mémoire dans le module, elle précise donc le numéro de chaque page -système, données, brouillon et programmes-. Elle contient les procédures d'**allocation** et de **désallocation** des pages données, la procédure de **remise à zéro** du brouillon et la **gestion en cache** des pages programmes. Cette dernière est simplifiée, la gestion des pages est une simple file, le but n'est pas de valider un algorithme de cache.

Cette couche réalise l'initialisation des différents paramètres de la mémoire, de son ramasse miettes et du cache.

II. 1. 4. La couche environnement.

Celle-ci simule ce qui ne fait pas directement partie du module.

Ainsi, elle charge en mémoire le **fichier répertoire** contenant l'ensemble des noms physiques des fichiers contenant les nœuds modèles et leurs numéros correspondants. Ceci est une simulation fort rudimentaire de la base de connaissances qui n'est vue que comme un ensemble de fichiers.

Cette couche simule également tout le **réseau de distribution** des nœuds modèles en réalisant le chargement du fichier le contenant dans une des pages programmes lorsque le nœud modèle n'est pas présent.

Enfin, cette couche contient deux programmes d'interface utilisateur, l'un permettant l'affichage sur l'écran d'un descripteur et l'autre l'affichage de l'état du module à un instant donné (**trace**).

II. 1. 5. La couche picoprogrammes.

Elle contient les instructions élémentaires réalisées par le processeur. Ce sont la gestion des deux piles (**empile** et **dépile**), le calcul sur la pile opérative (**calcul**), les **opérations sur un bloc** (recherche d'un élément, recopie d'un bloc, parcours d'un bloc et mesure de la longueur d'un bloc) et enfin la procédure fabriquant le descripteur de variable liée qui pointera sur une adresse donnée (**lier**).

II. 1. 6. La couche nanoprogrammes.

Cette couche utilise essentiellement la couche précédente pour réaliser des fonctions plus élaborées. Ce sont les procédures **déréférencement** qui permettent de parcourir une chaîne de pointeurs (variables liées) et d'accéder à un objet, **fetch** qui permettent une fois le traitement effectué sur une donnée de passer à la donnée suivante, **unifterm** qui réalise l'unification de deux termes et **evalterm** qui réalise l'évaluation d'un terme et enfin une procédure **fairebloc** qui fabrique dans la pile opérative un objet à partir de son adresse.

II. 1. 7. La couche microprogrammes.

Cette couche utilise essentiellement les deux couches précédentes. Elle réalise les algorithmes tels qu'ils ont été décrits dans le modèle d'exécution.

Elle contient donc les procédures de recopie d'un bloc du brouillon dans la mémoire (**copybloc**), d'unification de deux blocs définis par leur adresse de début (**unification**), de

génération d'une table de variables (**gentab**) ou de recopie d'une table de la mémoire dans le brouillon (**copytab**), d'évaluation d'un bloc défini par son adresse de début (**évaluation**), de génération d'une table de résultats après évaluation (**gentabres**) et enfin la libération de la place mémoire occupée par un bloc (**libbloc**) ou par une table (**libtab**).

II. 1. 8. La couche exécution.

Cette couche doit simuler le parcours de l'arbre de recherche. Elle est ici considérablement simplifiée et utilise les facilités de Pascal. En effet, elle ne contient que deux procédures s'appelant récursivement l'une l'autre, une procédure (**et**) simulant le fonctionnement d'un nœud processus ET et une procédure (**ou**) simulant le fonctionnement d'un nœud processus OU.

Les nœuds processus TERM sont gérés également par la procédure (et) avec un nombre de fils nul et le nœud REQ par le programme général de cette couche.

Cette façon de procéder permet de séparer la validation du traitement des données -unification et évaluation- de la validation des automates gérant la vie des nœuds processus et permettant le parcours de l'arbre de recherche. Par contre, il est impossible, en procédant ainsi, d'obtenir avec les deux processus récursifs un fonctionnement non déterministe. Cela limitera le jeu des programmes exécutés lors de cette première simulation.

II. 2. LES RESULTATS.

Le travail du compilateur est réalisé manuellement et chaque programme conduit à l'écriture d'un ensemble de fichiers. Un petit programme d'**édition** permet toutefois de vérifier pendant l'écriture, l'absence d'erreur sur les arités. Chaque fichier contient un seul nœud modèle et apparait comme une page pour le programme de simulation. Cette entorse à l'organisation de la base de connaissances de LAIOS dans laquelle chaque page contient plusieurs nœuds modèles (sauf exception éventuelle d'un nœud modèle de grande taille) présente un intérêt: bien que les programmes utilisés soient de petites tailles, des défauts de pages apparaitront dans la mémoire cache de programme.

II. 2. 1. Calcul de la suite de Fibonnacci.

Le programme a déjà été utilisé pour illustrer des propos précédents:

```
fib (0, 1).
fib (1, 1).
fib (N, +(X, Y)) :- fib ( -(N, 1), X), fib ( -(N, 2), Y).
```

En l'absence de contrainte, il a suffi de définir une soustraction sur les entiers naturels qui échoue sur les résultats négatifs.

L'ensemble des requêtes de ce programme va de **:- fib (2, X).** à **:- fib (10,X).**

Chaque programme contient donc sept fichiers: deux de type TERM, un de type ET, trois de type OU et un de type REQ.

Les résultats sont présentés dans les deux tableaux suivants avec **n** qui désigne l'indice de la suite, **np** le nombre de processus créés, **ni** le nombre d'inférences, **te** le temps d'exécution simulé en millisecondes, **occ** le nombre maximum de mots mémoire occupés et **perf** les performances en KLIPS. Le premier tableau (figure IV.17) correspond au programme sans

ramasse miettes et le second (figure IV.18) avec ramasse miettes.

n	np	ni	te	occ	perf
2	10	7	0,21	31	33
3	17	12	0,35	54	35
4	30	21	0,61	100	34
5	50	35	1,02	169	34
6	83	58	1,71	284	34
7	136	95	2,88	468	33
8	222	155	4,90	767	32
9				saturée	
10				saturée	

figure IV.17 : suite de Fibonnacci sur un module sans ramasse miettes ni gestion des nœuds processus.

n	np	ni	te	occ	perf
2	10	7	0,22	11	32
3	17	12	0,36	15	33
4	30	21	0,63	19	33
5	50	35	1,04	23	34
6	83	58	1,73	27	34
7	136	95	2,82	31	34
8	222	155	4,60	35	34
9	361	252	7,48	39	34
10	586	409	12,14	43	34

figure IV.18 : suite de Fibonnacci sur un module avec ramasse miettes et sans gestion des nœuds processus.

II. 2. 2. Concaténation de deux listes.

Le programme se réduit aux deux clauses:

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

Les requêtes sont toutes de la forme **:- conc (L1, L2, L3).** où L1 et L2 sont des listes de longueurs différentes à chaque essai.

Les deux tableaux de résultats présentent les même variables que précédement, le premier (figure IV.19) regroupant les résultats sans le ramasse miettes et le second (figure IV.20) les résultats avec.

L1	L2	np	ni	te	occ	perf
[]	[]	3	2	0,06	11	31
[]	(a,b,c,d,e)	3	2	0,17	63	11
(a,b)	(c,d,e)	9	6	0,48	181	12
(a,b,c)	(d,e)	12	8	0,60	223	13
(a,b,c,d,e)	[]	18	12	0,76	259	16
(a,b,c,d)	(e,f)	15	10	0,81	311	12
(a,b,c,d,e)	(f,g)	18	12	1,04	411	11
(a,b,c,d,e,f)	(g,h,i)	21	14	1,46	605	10
(a,b,c,d,e,f,g)	(h,i,j,k)	24	16	1,95	835	8

figure IV.19 : concaténation de deux listes sur un module sans ramasse miettes ni gestion de nœuds processus.

L1	L2	np	ni	te	occ	perf
[]	[]	3	2	0,07	6	30
[]	(a,b,c,d,e)	3	2	0,19	38	11
(a,b)	(c,d,e)	9	6	0,51	61	12
(a,b,c)	(d,e)	12	8	0,64	66	12
(a,b,c,d,e)	[]	18	12	0,80	88	15
(a,b,c,d)	(e,f)	15	10	0,87	110	12
(a,b,c,d,e)	(f,g)	18	12	1,11	148	11
(a,b,c,d,e,f)	(g,h,i)	21	14	1,55	190	9
(a,b,c,d,e,f,g)	(h,i,j,k)	24	16	2,07	263	8

figure IV.20 : concaténation de deux listes sur un module avec ramasse miettes et sans gestion de nœuds processus.

II. 2. 3. Inversion naïve d'une liste.

L'intérêt de ce programme, outre le fait qu'il constitue un programme de test classique, est d'utiliser le programme de concaténation précédent auxquel on ajoute les deux clauses suivantes:

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

Les requêtes de ce programme sont du type **:- nrev (L, LL).** où la variable L prend successivement des valeurs de listes de longueurs différentes.

Les mêmes paramètres que précédemment apparaissent dans les deux tableaux (figure IV.21 sans ramasse miettes et figure IV.22 avec).

L	np	ni	te	occ	perf
[]	3	2	0,05	8	37
(a)	8	5	0,24	56	21
(a,b)	16	10	0,56	162	18
(a,b,c)	27	17	1,09	336	16
(a,b,c,d)	41	26	1,85	590	14
(a,b,c,d,e)	58	37	2,90	936	13
(a,b,c,d,e,f)				saturée	
(a,b,c,d,e,f,g)				saturée	
(a,b,c,d,e,f,g,h)				saturée	

figure IV.21 : inversion naïve d'une liste sur un module sans ramasse miettes ni gestion des nœuds processus.

L	np	ni	te	occ	perf
[]	3	2	0,06	5	36
(a)	8	5	0,25	21	20
(a,b)	16	10	0,59	45	17
(a,b,c)	27	17	1,13	74	15
(a,b,c,d)	41	26	1,87	138	14
(a,b,c,d,e)	58	37	2,86	187	13
(a,b,c,d,e,f)	78	50	4,12	286	12
(a,b,c,d,e,f,g)	101	65	5,69	392	11
(a,b,c,d,e,f,g,h)	127	82	7,61	476	11

figure IV.22 : inversion naïve d'une liste sur un module avec ramasse miettes et sans gestion des nœuds processus.

II. 3. COMMENTAIRES.

Deux commentaires s'imposent au vu de ces résultats.

II. 3. 1. L'efficacité du ramasse miettes.

La première constatation est l'efficacité du ramasse miettes. Il s'avère peu coûteux en temps -il arrive même que le temps perdu à chercher de la place sans ramasse miettes soit supérieur à celui passé à la récupérer!- et autorise des gains sur la place occupée assez spectaculaires.

Cela est bien entendu dans une large mesure dû au choix de recopie de données. Ce choix s'avère nécessairement fort dispendieux en place mémoire sans ramasse miettes. Par contre, il facilite énormément le travail du ramasse miettes puisqu'une donnée est produite par un processus et consommée par un seul processus -soit un parent proche soit le producteur lui-même-.

Par la suite, le ramasse miettes sera implanté systématiquement.

II. 3. 2. L'influence de la longueur des données.

La seconde remarque concerne les performances. En moyenne, des performances d'une quinzaine de KLIPS semblent satisfaisantes car elles sont voisines de celles obtenues sur des stations de travail monoprocesseurs [ROU&_87], par exemple, la machine PSI de l'I.C.O.T. est annoncé pour trente KLIPS [FUC&_87]. Un autre élément de comparaison est fourni par Knödler et Rosenstiel, qui, avec l'aide d'un coprocesseur spécifique associé au 68000 de Motorola pensent atteindre les 40 KLIPS [KNÖ&_86].

Il faut cependant tempérer cette satisfaction par la remarque de l'importance considérable prise par la longueur des données. Lorsque les données sont brèves, c'est typiquement le cas du calcul de la suite de Fibbonnacci qui ne met en jeu chaque fois que deux ou trois entiers, les performances peuvent être considérées comme maximales. Mais, dès que la taille des données augmente, les performances diminuent rapidement.

Cela est dû à la recopie des données qui consomme une partie importante du temps d'exécution. Ce phénomène est particulièrement sensible sur le programme de concaténation qui n'effectue pratiquement aucun autre traitement que de déplacer les données.

CHAPITRE III. SIMULATION DU MODULE.

Comme pour les simulations précédentes, celles qui sont présentées ici ont deux objectifs, l'un de validation et l'autre de mesure. Il s'agit de valider le modèle de vie des nœuds processus, c'est-à-dire de vérifier si le fonctionnement des automates tels qu'ils sont définis émule bien le déroulement d'un programme PROLOG. La comparaison des temps d'exécution avec les précédents permettra d'estimer le surcoût dû à la gestion des processus.

III. 1. LE PROGRAMME.

Le programme est une extension du programme précédent. Il reprend les sept premières couches pratiquement sans modification. La huitième couche -couche exécution- est remplacée par quatre nouvelles couches.

III. 1. 1. La couche viedesnœuds.

Elle contient deux files d'attente, celle des **processus générateurs** et celle des **processus activés** avec les primitives de gestion de ces files (enfile, défile). Elle contient également la procédure de **création** des nœuds processus.

Pour des raisons de simplification du programme de simulation au détriment de la place mémoire occupée, le descripteur d'un nœud processus est identique quelle que soit sa classe, certaines informations étant alors dépourvues d'utilité ou de signification. Un nœud processus est défini comme un bloc d'arité 13. Les constantes de déplacement permettant d'accéder à chacune des informations sont définies dans la première couche.

Le bloc contient successivement:
- la classe du nœud sous forme d'un caractère;
- le numéro de son nœud modèle permettant l'accès à travers le cache au modèle;
- l'état de l'automate sous forme d'un entier;
- le numéro du fils courant, utile pour les nœuds de type OU,
- l'adresse du père;
- l'adresse du frère ainé;
- l'adresse du frère cadet;
- l'adresse des données en descente;
- l'adresse des données internes;
- l'adresse des données en entrée à la remontée;
- l'adresse du début de la file d'attente des données en sortie à la remontée;
- l'adresse de la fin de cette file d'attente;
- la table des fils.

Lorsqu'une adresse est inutilisée, la constante variable libre est recopiée à son emplacement. Dans la table des fils se trouvent soit le numéro du nœud modèle si le nœud processus n'est pas implanté, soit l'adresse du début du descripteur du nœud processus lorsqu'il est implanté.

III. 1. 2. La couche messagerie.

Elle contient tous les messages susceptibles d'être échangés entre processus, chacun de ces messages étant interprété par une procédure qui change l'état du destinataire et éventuellement lui communique une adresse de données.

La réception d'un message provoque toujours le chargement du processus dans la file d'attente des processus activés. Un même processus peut être rangé plusieurs fois dans cette file, et lorsqu'il est activé peut avoir un état différent de celui qu'il avait à l'entrée dans la file. Cette façon de procéder permet de vérifier le fonctionnement correct en cas de réception de plusieurs messages. Par contre, il n'est pas possible de simuler la réception d'un message par un processus actif alors que cela est possible en fonctionnement multiprocesseurs.

III. 1. 3. La couche exécution.

Elle contient toutes les procédures correspondant aux diverses activités selon la classe du nœud et son état.

Voici la liste de ces procédures:
- **genfils** engendre les fils d'un processus;
- **actinit** charge le modèle d'un nœud et initialise le processeur et le brouillon;
- **actpipe** amorce un flot de variable après un nœud ET;
- **actunif** réalise l'unification à la descente;
- **actlib1** libère un bloc de données à la descente;
- **actlib2** libère une table de variables à la descente;
- **annule** envoie un ordre de terminaison à tous les fils;
- **acteval** réalise l'unification à la descente;
- **actdist** distribue une copie du bloc donnée à chaque fils;
- **acteval2** réalise l'évaluation à la remontée;
- **enfileres** met un résultat en file d'attente à la remontée;
- **actunif2** réalise l'unification à la remontée;
- **actcont** termine une phase d'activité lorsqu'il n'y a plus de résultats;
- **açtenvoi** envoie un résultat au père sur sa demande;
- **actboucle** retourne à la situation initiale;
- **actchangefils** change de fils courant;
- **tuerfils** donne l'ordre aux fils de se détruire;
- **reprendre** revient à la situation générateur.

III. 1. 4. La couche activité.

Elle ne comporte que trois procédures.

La procédure **active** réalise l'activation du premier processus de la file des processus activés. L'action choisie dépend de l'état du processus et dans certains cas de sa classe; elle fait partie de la liste des procédures de la couche précédente.

La procédure **mère** réalise la génération des fils du premier processus de la file des processus générateurs. Lorsque la tête de file est un processus sans fils (de classe TERM), la procédure mère avance dans la file jusqu'à un processus ayant des fils à créer ou jusqu'à ce que la file soit vide. Cette bizzarerie provient du fait que ce sont les pères qui placent leurs fils dans la file des processus générateurs et ignorent leur classe (anticipation de la création sur l'activité).

La procédure **requête** permet d'initialiser l'arbre de recherche par la création du processus racine (de classe REQ).

Le programme général active alternativement la procédure mère et la procédure active. Les quelques heuristiques essayées favorisant soit la création soit l'activation n'ont pas eu de grandes répercussions sur le déroulement des programmes d'essai. Les premières ne permettent au mieux qu'un gain de quelques pour cent sur le temps, compensé par une perte du même ordre de grandeur sur l'occupation mémoire alors que les secondes offrent des gains et des pertes inverses.

III. 2. LES RESULTATS.

Les résultats sont présentés avec les mêmes variables **np, ni, occ, te, occ** et **perf**. Le non-déterminisme étant devenu possible, te désignera le temps nécessaire pour obtenir le dernier résultat; la fin de l'exécution demande en général 0,1ms de plus pour détruire les quelques processus encore actifs.

III. 2. 1. La suite de Fibonnacci.

Le nombre de processus est plus important que dans la simulation précédente à cause du non déterminisme qui fait qu'un succès n'empêche pas la poursuite de la recherche de solutions.

n	np	ni	occ	te	perf
2	37	13	599	1,19	10,8
3	61	22	725	2,05	10,7
4	101	37	852	3,62	10,2
5	165	61	980	6,04	10,1
6	269	100	1110	10,14	9,8

figure IV.23 : calcul de la suite de Fibonnacci par un module.

Les performances sont divisées par quatre à cause de la gestion des processus. Ce rapport doit être considéré pratiquement comme un maximum, car la gestion des processus demande un temps quasi constant par processus alors que le travail des processus dans ce programme est proche du minimum.

Il faut noter l'encombrement mémoire dû aux descripteurs des nœuds processus. Bien entendu, le ramasse miettes récupère également l'espace mémoire occupé par un processus à la mort de celui-ci.

III. 2. 2. La concaténation de deux listes.

Le programme est le même que précédemment mais pas les requêtes. En effet, il est désormais possible de vérifier le fonctionnement correct du non-déterminisme. En choisissant pour requête **:- conc (L1, L2, (...))**. avec une liste à la place des points de suspension, le programme doit calculer tous les couples de deux listes dont la concaténation fournit la liste donnée. Dans le tableau des résultats (figure IV.24) est ajoutée la variable **nr** indiquant le nombre de couples de résultats obtenus.

liste	np	ni	nr	occ	te	perf
[]	5	3	1	87	0,18	16,2
(a)	8	5	2	150	0,44	11,2
(a,b)	11	7	3	212	0,85	8,2
(a,b,c)	14	9	4	293	1,40	6,4
(a,b,c,d)	17	11	5	386	2,14	5,1
(a,b,c,d,e)	20	13	6	450	3,09	4,1
(a,b,c,d,e,f)	23	15	7	547	4,31	3,5
(a,b,c,d,e,f,g)	26	17	8	658	5,84	2,9
(a,b,c,d,e,f,g,h)	29	19	9	747	7,65	2,5
(a,b,c,d,e,f,g,h,i)	32	21	10	844	9,79	2,1
(a,b,c,d,e,f,g,h,i,j)	35	23	11	950	12,36	1,9

figure IV.24 : concaténation de deux listes sur un module.

Si le fonctionnement sur des programmes non déterministes est correct, par contre, il faut encore noter la grande sensibilité des performances à la longueur des données. Il est certain que la recopie des données n'est pas le principe le plus performant en fonctionnement monoprocesseur!

III. 2. 3. L'inversion naïve d'une liste.

Le programme est identique à celui du chapitre précédent, il va donc permettre une estimation du surcoût de la gestion des processus sur un programme ayant des traitements plus conséquents que dans le calcul de la suite de Fibonnacci.

liste	np	ni	occ	te	perf
[]	6	3	101	0,19	15,7
(a)	13	7	230	0,63	11,1
(a,b)	23	13	339	1,36	9,5
(a,b,c)	36	21	447	2,36	8,9
(a,b,c,d)	52	31	555	3,71	8,3
(a,b,c,d,e)	71	43	653	5,38	8,0
(a,b,c,d,e,f)	93	57	770	7,52	7,6
(a,b,c,d,e,f,g)	118	73	873	10,05	7,3
(a,b,c,d,e,f,g,h)	149	91	987	13,06	7,0
(a,b,c,d,e,f,g,h,i).	177	111	1106	16,60	6,7

figure IV.25 : inversion naïve d'une liste sur un module.

Il est intéressant de constater que le rapport de performances entre ces mesures et celles du chapitre précédent est à peu près de deux. Ceci montre bien qu'il n'est pas nécessaire que chaque processus ait beaucoup de traitements à effectuer pour que le temps passé au traitement soit supérieur ou égal à celui passé à la gestion des processus. Le choix de recouvrement de l'un par l'autre en anticipant les créations et en retardant les décès des nœuds processus par rapport à leur activation semble déjà se justifier; cette justification sera renforcée par les résultats des simulations suivantes.

Il est également intéressant de noter que les performances sont plutôt moins sensibles à la longueur des données dans ce programme que dans le précédent; le rapport entre le temps passé au traitement et le temps passé à la recopie est plus grand dans ce programme.

III. 2. 4. Une base de données familiale.

Il s'agit de faire un petit programme qui mette en valeur le non déterminisme de PROLOG avec des arbres de recherche plus équilibrés que celui (en arête de poisson) de la concaténation.

```
père (a,b).
père (a,c).
père (b,d).
père (c,e).
père (c,f).
père (d,g).
parent (X,Y) :- père (X,Y).
parent (X,Y) :- père (X,Z), parent (Z,Y).
frère (X,Y) :- père (Z,X), père (Z,Y).
cousin (X,Y) :- gpère (Z,X), gpère (Z,Y).
gpère (X,Y) :- père (X,Z), père (Z,Y).
```

Il y aurait évidemment beaucoup à dire sur la correction des résultats de tels programmes, mais le but n'est que de disposer de programmes simples de tests.

Trois requêtes sont envoyées successivement:

```
:- parent (a,g).
:- frère (X,Y).
:- cousin (X,Y).
```

la première a une réponse affirmative, la seconde a dix réponses à cause des répétitions (b-b, b-c, c-b, c-c, d-d, e-e, e-f, f-e, f-f et g-g) et la troisième dix également (d-d, d-e, d-f, e-d, e-e, e-f, f-d, f-e, f-f et g-g).

requête	np	ni	occ	te	perf
parent(a,g)	72	57	929	3,04	18,7
frère(X,Y)	17	44	302	2,69	16,3
cousin(X,Y)	77	115	606	7,10	16,1

figure IV.26 : base de données familiale sur un module.

Il est intéressant de constater que les performances redeviennent meilleures. Là encore, la taille des données est un facteur primordial et le fait d'avoir choisi des constantes courtes favorise les performances. Le compilateur de LAIOS augmentera donc les performances en utilisant des constantes formelles codées sur un seul mot.

CHAPITRE IV. SIMULATION DU RESEAU.

Cette simulation a deux objectifs. Le premier consiste à vérifier l'efficacité de la création anticipée des nœuds processus et de leur décès retardé. Le second consiste à faire une estimation de l'accélération obtenue sur l'exécution des algorithmes en utilisant le réseau de modules. Pour chaque programme de test, cette accélération sera comparée avec le parallélisme qui est mis en œuvre.

IV. 1. LE PROGRAMME.

Il reprend la structure et l'écriture du programme précédent en dix couches. La modification essentielle consiste à définir le temps non plus comme une variable globale mais comme une variable locale propre à chaque processus.

Cette option entraîne la modification du descripteur de processus par l'ajoût d'une variable **temps interne**. Cette modification produit une légère augmentation de l'espace mémoire occupé -ce qui apparait sur les tableaux de résultats- qui ne correspond à aucune réalité dans LAIOS. Il faut également modifier les procédures associées à chaque message qui doivent modifier le temps interne du processus récepteur. La date de début de l'activité d'un nœud processus provoquée par la réception d'un message est le maximum entre le temps propre du nœud processus et la date de réception du message.

Cette façon de procéder ne tient pas compte de l'architecture du réseau. Elle se place dans la **situation idéale** dans laquelle chaque nœud processus est implanté dans un module capable de prendre en charge son activité dès son réveil par réception d'un message. Les performances obtenues sont donc des maxima. Cependant, les programmes de tests sont tels que le nombre moyen de fils d'un nœud est voisin de deux et la première simulation a montré que cette situation est la plus favorable au déroulement sur le réseau hexagonal.

Deux mesures supplémentaires sont effectuées à chaque simulation, le nombre maximum de processus en attente de création dans la file des processus désirés **maxgen** et le nombre maximum des processus activés **maxact**. La somme de ces deux nombres est un majorant du nombre de modules différents utilisés. Il permet de "mesurer" le parallélisme mis en œuvre. Cette mesure est à prendre avec prudence car la simulation est séquentielle et la corrélation entre le temps réel du programme de simulation et les temps simulés de chaque processus n'est pas simple et introduit des distorsions.

IV. 2. LES RESULTATS.

Les programmes de tests sont les mêmes que précédemment afin de pouvoir faire les comparaisons facilement. Un cinquième programme a été ajouté qui met en œuvre davantage de parallélisme OU que les quatre premiers.

IV. 2. 1. Calcul de la suite de Fibonnacci.

Pour simplifier le tableau de résultats (figure IV.27), les nombres de processus et d'inférences étant identiques à ceux de la figure IV.23 ne sont pas répétés.

n	maxgen	maxact	occ	te	perf
2	4	16	632	0,34	38
3	4	16	767	0,52	42
4	5	20	905	0,90	41
5	5	20	1052	1,46	42
6			saturé		

figure IV.27 Calcul de la suite de Fibonnacci sur le réseau.

Les performances obtenues sont voisines de celles obtenues sur un module sans gestion de processus. Cela prouve que la gestion des processus est correctement recouverte par l'activité des processus. Mais cela montre aussi que ce programme ne met en jeu dans le modèle de LAIOS que fort peu de parallélisme. En fait le seul parallélisme est un parallélisme OU qui consiste à essayer à chaque calcul les trois possibilités fib(0), fib(1) et fib(N≥2), l'accélération que l'on peut en attendre ne peut être que modeste.

IV. 2. 2. La concaténation de deux listes.

Pour la même raison que précédemment np et ni n'apparaissent pas au profit de maxgen et maxact dans le tableau des résultats (figure IV.28).

liste	maxgen	maxact	nr	occ	te	perf
[]	2	3	1	92	0,13	23,1
(a)	2	6	2	158	0,23	21,5
(a,b)	2	8	3	225	0,36	19,3
(a,b,c)	2	9	4	307	0,52	17,3
(a,b,c,d)	2	11	5	381	0,70	15,6
(a,b,c,d,e)	2	12	6	475	0,91	14,3
(a,b,c,d,e,f)	2	13	7	567	1,17	12,8
(a,b,c,d,e,f,g)	2	15	8	693	1,45	11,7
(a,b,c,d,e,f,g,h)	2	16	9	779	1,76	10,8
(a,b,c,d,e,f,g,h,i)	2	17	10	900	2,10	9,9
(a,b,c,d,e,f,g,h,i,j)	2	19	11	982	2,49	9,2

figure IV.28 Concaténation de deux listes sur le réseau.

La sensibilité des performances à la longueur des données n'a pas été compensée par le parallélisme. Mais le parallélisme mis en jeu est modeste car l'arbre de recherche est en arête de poisson. Cela apparait clairement avec maxgen qui reste constament égal à deux.

Un autre aspect intéressant de cette simulation provient du nombre de solutions du problème posé. En effet, le temps passé à sortir toutes les solutions est à peu près celui qui est nécessaire pour avoir la plus complexe d'entre elles. La variable maxact qui montre clairement que pratiquement tous les modules de l'arbre travaillent en même temps, ceux du bas dans le calcul de la dernière solution et les autres dans la remontée en pipe line des solutions précédentes. C'est la raison pour laquelle le rapport entre les performances de la figure IV.28 (exécution sur le réseau) et de la figure IV.24 (exécution sur un module) varie de deux à quatre au fur et à mesure que le nombre de résultats augmente.

IV. 2. 3. Inversion naïve d'une liste.

Le tableau des résultats est présenté dans la figure IV.29.

liste	maxgen	maxact	occ	te	perf
[]	2	5	107	0,12	25,8
(a)	2	6	243	0,36	19,3
(a,b)	2	9	361	0,76	17,1
(a,b,c)	2	10	470	1,35	15,5
(a,b,c,d)	2	10	589	2,18	14,2
(a,b,c,d,e)	2	10	701	3,23	13,3
(a,b,c,d,e,f)	2	10	805	4,31	13,1
(a,b,c,d,e,f,g)	2	10	936	5,65	12,9
(a,b,c,d,e,f,g,h)	2	10	1061	7,28	12,5
(a,b,c,d,e,f,g,h,i)			saturée		

figure IV.29 Inversion naïve d'une liste sur le réseau.

L'arbre de recherche est plus complexe et les performances sont meilleures que les précédentes. Toutefois, ces performances diminuent encore avec la longueur de la donnée ce qui prouve que le parallélisme ne parvient pas à compenser le temps perdu à recopier les données. Dans cet exemple le parallélisme reste d'ailleurs modeste comme en témoignent les valeurs de maxgen et maxact qui plafonnent respectivement à deux et dix.

IV. 2. 4. La base de données familiale.

Avec le même programme et les mêmes requêtes qu'au chapitre précédent, la simulation sur le réseau offre les résultats du tableau ci-dessous (figure IV.30).

requête	np	ni	maxgen	maxact	occ	te	perf
parent (a,g)	72	57	7	22	991	0,85	66,7
frère (X,Y)	17	44	6	16	320	1,16	37,8
cousin (X,Y)	77	115	7	24	641	2,55	45,0

figure IV.30 : base de données familiale sur le réseau.

L'intérêt de ce programme est d'utiliser le ET pipe line, ce qui apparait dans le tableau au niveau des colonnes np et ni: dès que le nombre d'inférences est supérieur au nombre de processus, certains processus sont utilisés plusieurs fois.

La même remarque que dans le chapitre précédent doit être faite, la qualité des performances est due en partie à l'utilisation de constantes formelles courtes qui ne sont pas trop pénalisantes à recopier.

IV. 2. 5. Recherche des occurrences dans un arbre binaire.

Tous les exemples précédents utilisent assez peu le OU parallélisme, le choix au niveau des nœuds OU se limitant souvent entre une règle terminale et une règle récursive. Il est donc intéressant de mettre en évidence un programme de test qui utilise un arbre de recherche avec des branches de profondeur supérieure à un issues de nœuds OU.

Le programme suivant recherche les occurrences d'un terme dans un arbre binaire.

```
in (X, (G,X,D), []).
in (X, (G,Y,D), (g,L)) :- in (X, G, L).
in (X, (G, Y, D), (d, L)) :- in (X, D, L).
```

Sur la requête
:- in (a, ((([], b, []), b, ([], a, [])), a, (([], b, []), a, ([], b, [])))), L).
les trois réponses correctes sont
L = [].
L = (g, (d, [])).
L = (d, []).
qui correspondent respectivement aux trois positions de **a** dans l'arbre à la racine, au second niveau en empruntant la branche gauche puis la droite à partir de la racine et enfin au premier niveau en empruntant la branche droite.

Les résultats de cette simulation sont consignés dans le tableau IV.31.

arbre	np	ni	maxgen	maxact	occ	te	perf
a	17	10	4	13	320	0,17	58,7
a b a	37	22	4	26	727	0,38	58,6
a b a a b	57	34	5	35	1138	0,65	52,5
a b a b a b b	77	46	5	55	1584	0,70	65,3

figure IV.31 : recherche d'occurrences dans un arbre binaire.

Le défaut d'un tel programme est d'être un gros consommateur de place mémoire. Par contre, le parallélisme mis en œuvre augmente avec la profondeur de l'arbre. La longueur des données étant divisée par deux en descendant d'un niveau dans l'arbre de recherche, le temps de recopie ne s'avère pas trop pénalisant . En se limitant aux arbres équilibrés, ce programme de test présente la caractéristique d'avoir les performances qui croissent avec la taille du problème, ce qui est la raison d'être d'un réseau multiprocesseur.

IV. 3. LE FUTUR DE LAIOS.

Les exemples traités ci-dessus sont certes de taille modeste. Ils montrent cependant l'aptitude du réseau à faire travailler ses modules en parallèle. Mais ils montrent aussi la nécessité d'améliorer les performances pour que le réseau trouve sa justification.

Tel qu'il est défini actuellement, LAIOS n'a pour limitation de taille -à l'exclusion de toute considération de prix, d'encombrement ou de refroidissement- que la charge du réseau de distribution de nœuds modèles. Sur ce plan, le réseau type utilisé pour simuler la répartition de la charge, avec ses 330 modules est envisageable car le nombre maximum de modules reliés à un bus n'est alors que de vingt. Un tel réseau pourrait desservir six utilisateurs, un sur chaque coté de l'hexagone. Mais encore faudrait-il qu'un aussi grand nombre de processeurs offre un gain substanciel sur les performances.

Les simulations ont confirmé que la complexité d'un algorithme parallèle exécuté sur une

machine multiprocesseur doit être mesurée en fonction non seulement du nombre d'instructions, mais aussi du nombre de transferts de données, de leur taille et du surcoût dû à la gestion des processus. L'accroissement des performances du réseau ne peut passer que par la prise en compte de tous ces paramètres.

En architecture, il est possible d'adjoindre au réseau une mémoire partagée et un réseau de connexions pour les objets de grande taille, symétrique du réseau de distribution des nœuds modèles. Cela permettrait de transmettre ces objets par référence et non plus par valeur. L'utilisation de mémoire cache au niveau de chaque module posera toutefois le problème délicat du maintien de la cohérence.

Comme cela a déjà été évoqué précédemment, le langage devra permettre la programmation d'algorithme de type "divide and conquer" afin d'élargir le domaine du parallélisme exploitable.

Enfin, il est décevant de constater le faible parallélisme mis en jeu dans les programmes de tests classiques, il faut alors demander à l'algorithmique parallèle de dégager les méthodes de programmation efficace des machines multiprocesseurs.

EPILOGUE.

Cette première version de LAIOS présente la mise en œuvre dans un ensemble cohérent de plusieurs concepts modernes en architecture des ordinateurs. Malheureusement, les simulations mettent en évidence des faiblesses qui handicapent les performances. Les versions futures devront donc tenir compte de cet état de fait et intégrer d'autres notions permettant d'y remédier. Cette conclusion se propose donc de résumer les **points forts et les points faibles de LAIOS.**

L'intelligence artificielle représente une demande considérable de moyens de calcul mais elle offre une vision du raisonnement plus naturellement parallèle que l'algorithmique classique. Toutefois, la construction de **machines parallèles orientées intelligence artificielle** reste encore un défi que de nombreux laboratoires de recherche au monde tentent de relever.

Le principe d'un **moteur d'inférences** travaillant sur une **base de connaissances** peut d'autant mieux être généralisé à un ensemble de moteurs d'inférences identiques travaillant sur la même base de connaissances que le travail à effectuer consiste à multiplier les essais pour trouver ceux qui aboutissent. Il faut cependant garantir l'accès aux données partagées, les synchronisations dans les recherches et leur relative indépendance.

Les langages utilisés en intelligence artificielle sont généralement des **langages déclaratifs**. Ces langages se prêtent beaucoup mieux à un parallélisme implicite que les langages impératifs classiques. LAIOS, en choisissant de se baser sur PROLOG, bénéficie de cet avantage. Toutefois, même implicite, le parallélisme exige une étude algorithmique préalable pour être un facteur d'accélération à l'exécution.

Le parallélisme implicite peut découler de deux principes plutôt antagonistes: le non-déterminisme et la stratégie "divide and conquer". La première version de LAIOS exploite le **non-déterminisme** de PROLOG. Du fait de l'introduction de fonctions évaluables, il a été noté combien il serait naturel d'autoriser les fonctions évaluables définies par l'utilisateur et la programmation de stratégies "divide and conquer" par ces fonctions. Cette intégration, en augmentant les possiblités de parallélisme, répondra à la demande apparue lors des simulations.

Le **déroulement dynamique d'un arbre de recherche** est un modèle parfaitement adapté à un réseau de connectique partielle. Il permet la mise en œuvre d'un algorithme de répartition de la charge simple, distribué et performant.

La **hiérarchie de nœuds** permet l'implantation du langage sur le réseau par la définition d'automates implantés directement au niveau de chaque processeur. L'ensemble des classes devra être redéfini et enrichi lors de l'introduction des fonctions utilisateurs.

Les **données autodéfinies** permettent de s'abstraire des instructions machines et de définir un opérateur universel. Elles simplifient considérablement le calcul formel. Elles autorisent le parallélisme sur des modules banalisés.

La structuration des **données en blocs** permet un déplacement facile des données. C'est un choix cohérent avec la **recopie de données**. Ce choix est un point clé dans LAIOS. Il simplifie de façon considérable la synchronisation des processus. Il permet, avec le **principe de localité**, de ne réaliser des échanges de données qu'entre voisins. Il reste un des handicaps les plus évidents pour les performances.

Il semble clair qu'il faille trouver le moyen de distribuer les structures constantes de grande taille sur les processeurs à l'image de ce qui se fait pour les nœuds modèles. Ceci permettrait de ne faire transiter par recopie que des données de courte taille: variables non évaluées, constantes ou pointeurs sur ces structures partagées. Plusieurs choix sont possibles pour introduire ces concepts dans LAIOS. Cette amélioration apparait à la lumière des simulations comme une condition pour que la taille des données d'un problème ne soit pas la cause d'une diminution des performances.

L'architecture des modules en couches permet de rendre les problèmes relatifs à chaque couche orthogonaux à ceux des autres couches.

La définition d'une base de connaissances accessible en lecture seulement au cours du déroulement d'un programme est la garantie de pouvoir utiliser des caches de manière simple et efficace.

L'utilisation d'un **réseau hexagonal** pour la couche exécution permet à la fois le **OU parallélisme** et le **ET pipe line** et semble offrir une connectique suffisante, tout en restant modérée, pour permettre un taux d'utilisation considérable du réseau.

La facilité avec laquelle peut être envisagé un **usage multi-utilisateurs** ainsi que la possibilité de **reconfigurer le réseau sur pannes** sans pertes considérables d'efficacité sont des caractéristiques intéressantes de ce réseau.

L'utilisation de **mémoires à double accès** et d'un **allocateur d'espace** permet la transmission de données de taille quelconque sans synchronisation complexe. Associé au principe de recopie des données, l'allocateur de mémoire peut garantir une gestion efficace de la place mémoire, ce qui est une nécessité avec une exécution dynamique.

D'une manière plus générale, la distribution du travail sur chaque brique de l'architecture du module est envisageable. Cela simplifierait le processeur central et lui permettrait d'être plus performant. Ainsi les mémoires double accès possèderaient-elles à la fois leur arbitre et leur allocateur, le brouillon sa gestion de piles et de tableaux, les mémoires programmes leur cache et leur accès direct au bus de distribution des nœuds modèles ...

BIBLIOGRAPHIE.

[ALMA_85] G.S.ALMASI.
Research in highly parallel computer systems.
I.B.M. Research Report RC11494 (# 51284).
September 25, 1985.

[AND&_87] J.M.ANDERSON, W.S.COATES, A.L.DAVIS, R.W.HON, I.N.ROBINSON, S.P.ROBISON, K.S.STEVENS.
The Architecture of FAIM-1.
COMPUTER, p.55-65.
January 1987.

[ARD&_81] B.W.ARDEN, H.LEE.
Analysis of chordal ring network.
IEEE Transactions on Computers, p. 291-295.
April 1981.

[ARN&_87] M.ARNOUX, G.BECKER, M.C.THOMAS.
Un système de frames expertes en PROLOG.
Séminaire de programmation en logique.
TREGASTEL, 19-21 mai 1987.

[BEK&_86] Y.BEKKERS, B.CANET, L.UNGARO.
Projet MALI (Mémoire Adaptée aux Langages Indeterministes).
Bigre+globule n°50.
1986.

[BEL&_86] J.BELLONE, R.PICCA.
Concurrent Prolog: Modèle de calcul et gestion de la mémoire.
Séminaire de programmation en logique.
TREGASTEL, 21-23 mai 1986.

[BE&T_86] D.BELL, J.TELLIER.
La HM 65231: Outil de communication sans état d'attente.
Minis et micros n°267.
17 Novembre 1986.

[BORG_84] P.BORGWARDT.
Parallel Prolog using stack segments on shared-memory multiprocessors.
International Symposium on Logic programming.
Atlantic City, U.S.A., February 1984.

[BOU&_83] S.BOURGAULT, M.DINCBAS, D.FEUERSTEIN, J.P.LE PAPE.
Lislog : l'an II.
Séminaire de programmation en logique.
PERROS-GUIRREC, 22-23 mars 1983.

[BOU&_87] S.BOURGAULT, J-P.LE PAPE, D.RANSON
LISLOG-C. Une première étape vers l'expression généralisée de contraintes en programmation en logique.
Séminaire de programmation en logique.
TREGASTEL, 19-21 mai 1987.

[CAS&_87] M.CASTAN, A.CONTESSA, E.COUSIN, G.DURRIEU, B.LECUSSAN, M.LEMAITRE, P.NG.
Le processeur de réduction de MaRS, Machine à Réduction Symbolique.
Bigre+Globule n°56.
Novembre 1987.

[CATI_87] E.CATIER.
Des machines pour l'intelligence artificielle.
Electronique industrielle n°125.
1 Mai 1987.

[CHAS_86] J.CHASSIN de KERGOMMEAUX.
Machines abstraites pour l'implantation de PROLOG.
Rapport de recherche n°589.
I.M.A.G. Grenoble, France, Fevrier 1986.

[CHA&_85] J.H.CHANG, D.DeGROOT, A.M.DESPAIN.
AND-Parallelism of Logic Programs based on a static data dependency analysis.
COMPCON.
Spring 1985.

[CHA&_87] D.CHAN, P.DUFRESNE, R.ENDERS.
PHOCUS: production rules, Horn clauses, objects and contexts in a unification-based system.
Séminaire de programmation en logique.
TREGASTEL, 19-21 mai 1987.

[CHO&_85] E.CHOURAQUI, H.FARRENY, D.KAYSER, H.PRADE.
Modélisation du raisonnement et de la connaissance.
Programme de recherche concertée en intelligence artificielle.
Journée Intelligence Artificielle C.N.R.S.
Gif-sur-Yvette, France, 22 mars 1985.

[CHU&_85] Y.CHU, K.ITANO.
A Prolog Direct-execution Machine Organization.
Computer Science Technical Report Series TR-1545.
University of Maryland, August, 1985.

[CIEP_84] **A.CIEPIELEWSKI.**
Towards a computer architecture for OR-parallel execution of logic programs.
Academic Dissertation.
Royal Institute of Technology.
Stockholm, Sweden, May 17, 1984.

[CIE&_84] **A.CIEPIELEWSKI, S.HARIDI.**
Control of activities in the OR-parallel token machine.
International Symposium on Logic Programming.
Atlantic City, U.S.A., February 1984.

[CLA&_84] **K.CLARK, S.GREGORY.**
Notes on systems programming in Parlog.
International conference on fifth generation computer systems.
1984.

[COD&_86] **M.CODISH, E.SHAPIRO.**
Compiling OR-Parallelism into AND-Parallelism.
Third International Conference on Logic Programming.
London, Great Britain, 1986.

[CONE_85] **J.S.CONERY.**
The AND/OR Process Model for Parallel Execution of Logic Programs.
PH.D. Dissertation.
University of California, Irvine, U.S.A. 1983.

[CON&_85] **J.S.CONERY, D.F.KIBLER.**
AND-Parallelism and non determinism in logic programs.
New Generation Computing, Vol. 3.
1985.

[COR&_87] **R.CORNU-EMIEUX, D.LATTARD, G.MAZARE, P.OBJOIS.**
Réseau de cellules asynchrones dédié à la simulation logique: conception et réalisation.
Bigre+globule n°56.
Novembre 1987.

[COS&_87] **M.COSNARD, B.TOURANCHEAU, G.VILLARD.**
Présentation de l'hypercube T20 de F.P.S.
Bigre+globule n°56.
Novembre 1987.

[CRAS_86] **J.Y.CRAS.**
Types et fonctions en PROLOG: une approche ensembliste.
Seminaire de programmation en logique.
TREGASTEL, 21-23 mai 1986.

[CRO&_85] **W.CROWTHER, J.GOODHUE, R.GURWITZ, R.RETTBERG, R.THOMAS.**
The Butterfly (TM) Parallel Processor.
Networks architecture Systems Newsletter.
September, December, 1985.

[DAV&_85] **A.L.DAVIS, S.V.ROBISON.**
The FAIM-1: Symbolic Multiprocessing System.
COMPCON.
Spring 1985.

[DEGR_85] **D.DeGROOT.**
Alternate graph expressions for restricted AND-Parallelism.
COMPCON.
Spring 1985.

[DES&_85] **A.M.DESPAIN, Y.N.PATT.**
AQUARIUS: A High-Performance Computing System for Symbolic/numeric Applications.
COMPCON.
Spring 1985.

[DEVI_85] **M.DEVIN.**
La microprogrammation du système LELISP: une première approche.
Rapport de recherche n°441.
INRIA Sophia Antipolis. September 1985.

[DEW&_87] **L.DEWES, E.MAFETY.**
Modulog 'isa' Prolog.
Séminaire de programmation en logique.
TREGASTEL, 19-21 mai 1987.

[FUC&_87] **K.FUCHI, K.FURUKAWA.**
The Role of Logic Programming in the Fifth Generation Computer project.
New Generation Computing.
OHMSHA, february 1987.

[GIA&_85] **F.GIANNESINI, H.KANOUI, R.PASERO, M.VAN CANEGHEM.**
PROLOG.
InterEditions.
PARIS, 1985.

[GIRA_85] **J.J.GIRARDOT.**
Les langages et les sytèmes LISP, une introduction.
EDITESTS.
1985.

[GUS&_86] S.L.GUSTAFSON, S.HAWKINSON, K.SCOTT.
The architecture of a homogeneous vector supercomputer.
Journal of parallel and distributed Computing, vol. 3, p. 297-304.
1986.

[HAYE_85] F.HAYES-ROTH.
Rule-based systems.
Communications of the ACM. Vol 28, n° 9.
september 1985.

[ICOT_82] I.C.O.T.
Outline of research and development plans for fifth generation computer systems.
May 1982.

[KALE_87] L.V.KALE.
'Completeness' and 'Full Parallelism' of Parallel Logic Programming Schemes.
Report n° UIUCDCS-R-87-1321.
Department of Computer Science,
University of Illinois at Urbana-Champaign; February 87.

[KAWA_84] K.KAWANOBE.
Present status of the fifth generation computer systems project.
ICOT Journal, n° 5.
1984.

[KHAR_86] M.KHAROUNE.
Que demande-t-on aux multiprocesseurs pour exécuter PROLOG?
Bigre+globule n°50.
1986.

[KHA&_86] M.KHAROUNE, A.AGGOUN.
Deux approches du parallélisme ET en PROLOG.
Séminaire de programmation en logique.
TREGASTEL, 21-23 mai 1986.

[KHO&_87] M.KHOLI, M.E.GIULIANO, J.MINKER.
An Overview of the PRISM Project.
A.C.M. Computer Architecture News, vol 15, n°1.
1987.

[KNÖ&_86] B.KNÖDLER, W.ROSENSTIEL.
A Prolog Preprocessor for Warren's abstract instruction set.
EUROMICRO 86, Microarchitectures, developments and applications.
VENICE, Italy, September 15-18, 1986.

[KOYA_86] K.KOYAMA.
Microprogram Control of a Prolog Machine.
SIGMICRO Volume 17, number 1.
March, 1986.

[KOWA_74] R.KOWALSKI.
Predicate Logic as Programming Language.
I.F.I.P. Information Processing.
North Holland Publishing Company, 1974.

[LI_86] P.P.LI.
A parallel excution model for logic programming.
PHD Thesis.
California Institute of Technology; Pasadena, California 91125;
April 24, 1986.

[LIND_84] G.LINDSTROM.
OR-Parallelism on Applicative Architectures.
Second International Logic Programming Conference.
Uppsala, Sweden, August 1984.

[LIN&_84] G.LINDSTROM, P.PANANGADEN.
Stream-based execution of logic programs.
International Symposium on Logic Programing.
Atlantic City, U.S.A., February 1984.

[LI&M_86] P.P.LI, A.J.MARTIN.
The Sync Model: A Parallel Execution Method for Logic Programming.
Report n° 5221 : TR : 86.
Computer Science Department.
California Institute of Technology, Pasadena, U.S.A., March 1986.

[MAGO_85] G.MAGO.
Making Parallel Computation Simple: The FFP Machine.
Proceedings of COMPCON S'85, p. 424-428.
1985.

[MAR&_82] A.MARTELLI, U.MONTANARI.
An efficient unification algorithm.
A.C.M. Transactions on Programming Languages and Systems.
Volume 4 n° 2.
April 1982.

[MEV&_87] A.MEVEL, T.GUEGUEN.
SMALLTALK-80.
EYROLLES.
Paris, 1987.

[MOTO_85] T.MOTO-OKA.
The Japanese Fifth Generation Computer System Project.
IEEE Design and Test.
October 1985.

[MUNT_87] T.MUNTEAN.
SUPERNODE: une architecture parallèle et reconfigurable de Transputers.
Bigre+Globule n°56.
Novembre 1987.

[MYER_82] G.J.MYERS.
Requisites for improved architectures.
Advances in computer architectures. p 53 -100.
Ed J.WILEY & SONS, INC, 1982.

[NAKA_84] H.NAKAGAWA.
AND parallel PROLOG with divided assertion set.
International Symposium on Logic Programing.
Atlantic City, U.S.A., February 1984.

[NAU_83] D.S.NAU.
Expert Computer Systems.
Computer.
February 1983.

[ONA&_84] R.ONAI, H.SHIMIZU, K.MASUDA, M.ASO.
Analysis of sequential Prolog programs.
Technical Report TR-048.
I.C.O.T., Japan, March 1984.

[ONA&_85] R.ONAI, M.ASO, H.SHIMUZU, K.MASUDA, A.MATSUMOTO.
Architecture of a reduction based parallel inference machine: PIM-R.
ICOT Technical Report TR-105.
Maech, 1985.

[PAZ&_87] J.L.PAZAT, B.VAUQUELIN.
MAPP: Un mécanisme d'aide au placement de processus sur une architecture multiprocesseur.
Bigre+globule n°56.
Novembre 1987.

[PFI&_85] G.F.PFISTER, W.C.BRANTLEY, D.A.GEORGE, S.L.MARVEY, W.J.KLEINFELDER, R.P.McAULIFFE, E.A.MELTON, V.A.NORTON, J.WEISS.
The I.B.M. Research Parallel Processor Prototype (RP3): Introduction and Architecture.
Research Report RC11060.
I.B.M. Research Division, Yorkton, U.S.A., March 22, 1985.

[PROL_87] PROLOG II Version 2.4.
Manuel de référence.
PrologIA.
Marseille, 1987.

[RECH_85] LA RECHERCHE n°170
NUMERO SPECIAL: l'Intelligence Artificielle.
octobre 85.

[ROU&_87] **S.ROUX, J.-M.CARGOUET.**
Note technique sur les machines symboliques.
D.R.E.T.
Juillet 1987.

[SAIN_86] **E.SAINT-JAMES.**
LISP comme sa machine virtuelle
Bigre + globule n° 50.
1986.

[SANS_85] **J.P.SANSONNET.**
La machine pour les applications en intelligence artificielle: MAIA.
Materiels et logiciels pour la cinquième génération.
Congrès AFCET INFORMATIQUE.
Paris, France, Mars 1985.

[SCHI_84] **M.SCHINDLER.**
Artificial intelligence begin to pay off with expert systems for engineering.
Electronic Design.
August 9, 1984.

[SHA&_83] **E.SHAPIRO, A.TAKENCHI.**
Object oriented programming in Concurrent Prolog.
New Generation Computing, vol 1, p. 25-48.
1983.

[SHI&_84] **S.SHIBAYAMA, T.TAKUTA, N.MIYAZAKI, H.YOKOTA, K.MURAKAMI.**
A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor.
New Generation Computing, Vol. 2, p. 131-155.
1984.

[SHO&_86] **Y.SHOBATAKE, N.AISO.**
A unification processor based on a uniformly structured cellular hardware.
13th annual international symposium on computer architecture.
TOKYO, Japan, June 1986.

[STOL_87] **S.J.STOLFO.**
Initial Performance of the DADO2 Prototype.
COMPUTER.
January 1987.

[SUZU_85] **N.SUZUKI.**
Concurrent Prolog as an Efficient VLSI Design Language.
COMPUTER, p.33-40.
February, 1985.

[SYRE_85] **J.C.SYRE.**
Une revue de modèles parallèles pour Prolog.
Séminaire de programmation en logique.
Tregastel, France, 1985.

[SYRE_86] **J.C.SYRE.**
Une revue des architectures d'ordinateurs pour les sytèmes logiques.
Congrès R.F.I.A.
Grenoble, France, 1986.

[TAKI_86] **K.TAKI.**
The parallel software research and development tool: Multi-PSI System.
France-Japan Artificial Intelligence and Computer Science Symposium.
1986.

[TAM&_84] **N.TAMURA, Y.KANEDA.**
Implementing Parallel Prolog on a Multi-processor Machine.
International Symposium on Logic Programming.
Atlantic City, U.S.A., February 1984.

[TREL_85] **P.C.TRELEAVEN.**
Parallel Architectures and Languages for AIP.
ESPRIT Technical Week, Project 415.
1985.

[TRE&_86] **P.C.TRELEAVEN, A.N.REFENES, K.J.LEES, S.C.McCABE.**
Computer Architectures for Artificial Intelligence.
Technical Report.
March 1986.

[UEDA_86] **K.UEDA.**
Making Exhaustive Search Programs Deterministic.
Third International Conference on Logic Programming.
London, Great Britain, 1986.

[VIDE_87] **J.M.VIDECOQ.**
Architectures pour l'intelligence artificielle.
Genie logiciel n° 7, pages 27-30.
mars 1987.

[WAH_87] **B.W.WAH.**
New Computers for Artificial Intelligence Processing.
Computer. Special Artificial Intelligence.
January 1987.

[WAH&_86] **B.W.WAH, G.J.LI.**
A Survey on special purpose computer architecture for A.I.
SIGART Newsletter n°96, p.28-46.
April 1986.

[WARR_78] **D.WARREN.**
PROLOG on teh DEC ssystem-1.
1978.

[WARR_83] **D.WARREN.**
An absrtact Prolog instruction set.
Technical Note 303, SRI Project 4776.
October 1983.

[WAR&_84] **D.WARREN, D.BOWEN, L.BYRD, L.PEREIRA.**
C-PROLOG user's manual, version 1.
Department of architecture.
December 3, 1984.

[WEI&_85] **M.WEISER, S.KOGGE, M.McELVANY, R.PIERSON, R.POST, A.THAREJA.**
Status and performance of the ZMOB parallel processing sytsem.
Proceedings of COMPCON S'85, p.81-83.
1985.

[WHIT_85] **C.WHITBY-STREVENS.**
The Transputer.
12th International Symposium on Computer Architecture.
1985.

[WILL_84] **C.WILLIAMS.**
Software tools packages the expertise needed to build expert systems.
Electronic Design.
August 9, 1984.

[WISE_86] **M.J.WISE.**
Prolog Multiprocessors.
Prentice/Hall International Editions.

TABLE DES MATIERES.

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. P. JORRAND, Directeur de recherche
. J.P SANSONNET, Directeur de recherche

Monsieur DUPRAT Jean

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Microélectronique"

Fait à Grenoble, le 12 juillet 1988

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

~~P.O. le Vice Président,~~