Parallélisme dans une machine base de connaissances Prolog

Weldong Dang

THESE

présentée par

Weidong DANG

pour obtenir le titre de **DOCTEUR**

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

spécialité: informatique

PARALLELISME DANS UNE MACHINE BASE DE CONNAISSANCES PROLOG

Date de soutenance: Le 7 Janvier 1987

Composition du jury:

Mr. J.MOSSIERE	Président
Mr. G.BERGER SABBATEL	Examinateurs
Mr. B.COURTOIS	
Mr. D.HERMAN	
Mr. Ph.JORRAND	
Mr. J.ROHMER	

Thèse préparée au sein du laboratoire: IMAG/TIM3

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Daniel BLOCH

Année 1987

Vice - Présidents : René CARRE
Jean-Marie PIERRARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BESSON Jean	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
BUYLE-BODIN Maurice	ENSERG
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHIAVERINA Jean	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Mar	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Pierre	ENSIEG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSEEG
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
PAUTHENET René	ENSIEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET PIERRE	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS 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)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
PLA Fernand
GHIBAUDO Gérard
HAMAR Sylvaine
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET Marcel
CARRE René
FRUCHART Robert
JORRAND Philippe
LANDAU Ioan
MARTIN

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BINDER Gilbert
BONNET Roland
BORNARD Guy
CALMET Jacques
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
JOUD Jean Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MERMET Jean
MUNIER Jacques
SENATEUR Jean Pierre
SUERY Michel
TEODOSIU
WACK Bernard

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

E.N.S.E.E.G

BERNARD Claude
CHATILLON Catherine
CHATILLON Christian
COULON Michel
DIARD Jean-Paul
FOSTER Panayotis
HAMMOU Abdelkader
MALMEJAC Yves
MARTIN GARIN Régina
SAINTFORT Paul
SARRAZIN Pierre
SIMON Jean-Paul
TOUZAIN Philippe
URBAIN Georges

E.N.S.E.R.G

BOREL Joseph
CHOVET Alain
DOLMAZON Jean-Marc
HERAULT Jeanny

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

BOIS Daniel
DARVE Félix
MICHEL Jean-Marie
ROWE Alain
VAUCLIN Michel

E.N.S.I.M.A.G

BERT Didier
COURTIN Jacques
COURTOIS Bernard
DELLA DORA Jean
FONLUPT Jean
SIFAKIS Joseph

E.F.P.G

CHARUEL Robert

C.E.N.G

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

Laboratoires extérieurs

C.N.E.T

DEMOULIN Eric
DEVINE
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M.MERMET
Directeur des Etudes et de la formation: Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

PROFESSEURS DE 1ère CATEGORIE

COINDE Alexandre	Gestion
GOUX Claude	Métallurgie
LEVY Jacques	Métallurgie
LOWYS Jean-Pierre	Physique
MATHON Albert	Gestion
RIEU Jean	Mécanique-Résistance des matériaux
SOUSTELLE Michel	Chimie
FORMERY Philippe	Mathématiques Appliquées

PROFESSEURS DE 2ème CATEGORIE

HABIB Michel	Informatique
PERRIN Michel	Géologie
VERCHERY Georges	Matériaux
TOUCHARD Bernard	Physique Industrielle

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

BISCONDI Michel	Métallurgie
DAVOINE Philippe	Géologie
FOURDEUX Angeline	Métallurgie
KOBYLANSKI André	Métallurgie
LALAUZE René	Chimie
LANCELOT Francis	Chimie
LE COZE Jean	Métallurgie
THEVENOT François	Chimie
TRAN MINH Canh	Chimie

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

Je tiens à remercier :

- Monsieur Mossière, Professeur à l'ENSIMAG, qui m'a fait l'honneur d'accepter de présider le jury et d'être le rapporteur de cette thèse.
- Monsieur Herman, Professeur à l'INSA de Rennes, d'avoir bien voulu s'intéresser à mon travail et d'être le rapporteur.
- Monsieur Jorrand, Directeur de recherche au CNRS, pour ses conseils et ses encouragements pendant ces années et pour sa participation au jury.
- Monsieur Rohmer, Chef de service de l'Intelligence Artificielle de Bull, qui m'a témoigné sa confiance en participant au jury.
- Monsieur Courtois, Chargé de recherche au CNRS, Directeur du Laboratoire TIM3, de m'avoir accepté dans son laboratoire pour préparer cette thèse et d'avoir dirigé ce travail.

Je tiens à remercier tout particulièrement :

- Monsieur Berger Sabbatel, Chargé de recherche au CNRS, qui m'a guidé tout au long de ce travail, en particulier pour la rédaction de cette thèse, et m'a accordé sa confiance et son amitié.

Je tiens à remercier aussi :

- Monsieur Anceau, Ingénieur chez Bull, Professeur à l'ENSIMAG, qui a dirigé la première année de ce travail.
- Monsieur Muntean, Directeur de recherche au laboratoire de génie informatique à l'IMAG, ses conseils ayant beaucoup contribué à la réalisation de cette thèse.
- Madame Amielh et Messieurs Marchal, Guyot, Jerraya, Jamier, Souai, Rougeaux, Bourcier, ... de l'équipe d'Architecture des Ordinateurs du Laboratoire TIM3, pour leurs sympathies et conseils.
- Mes amis Jean-Christophe Ianeselli et Christophe Roche. Leurs remarques et conseils m'ont été très profitables.

Enfin, je remercie également ma femme Yu qui m'a soutenu tout au long de cette thèse, et ma mère pour tous ses encouragements.

A ma mère et ma famille.

A Yu, qui a fait preuve de beaucoup de patience.

RESUME

Cette Thèse aborde divers problèmes du parallélisme dans le projet OPALE, qui consiste à concevoir et à réaliser une machine base de connaissances reposant sur Prolog. Un modèle d'interprétation parallèle de Prolog est d'abord défini. Ce modèle se base sur la notion de processus. Ensuite, en introduisant un type d'architecture nommé "architecture orientée processus", l'exécution du modèle dans un environnement multi-processeurs est étudiée. Une expérimentation par simulation en langage parallèle OCCAM, en vue de valider le modèle et son exécution dans une architecture multi-processeurs, est decrite. Enfin, un système Prolog, influencé par les notions développées dans cette thèse, avec la capacité de la manipulation des clauses extérieures à la mémoire centrale est présenté.

mots clés: machine base de connaissances,
Prolog,
parallélisme,
architecture multi-processeurs.

ABSTRACT

This thesis describes how the issue of parallelism was approached in the OPALE project, which aims at the design and development of a knowledge base machine based on Prolog. We first define a parallel Prolog interpretation model, this model relies on the notion of process. By introducing a type of architecture named "processes oriented architecture", the execution of the model in a multiprocessor environment is then studied. An experiment using simulation in the parallel language OCCAM, in order to validate the model and its execution in a multiprocessor machine, is described. Finally, we present a Prolog system, influenced by the ideas developed in this thesis, with the capacity of manipulation of the clauses in secondary memory.

Key words: knowledge base machine,
Prolog,
parallelism,
multiprocessor architecture.

TABLE DES MATIERES

PARALLELISME
DANS UNE MACHINE BASE DE CONNAISSANCE PROLOG

PRESENTATION

PRESENTATION

Cette thèse est préparée dans le contexte du développement de l'ordinateur de "nouvelle génération", ou "cinquième génération". L'ordinateur nouvelle génération est caractérisé par sa capacité de traitement des connaissances. En plus du calcul numérique et du contrôle de processus dans l'industrie, on veut que la machine puisse effectuer l'opération humaine essentielle: le raisonnement. Le raisonnement par ordinateur se fait par un mécanisme d'inférence reposant sur une base de connaissances. On prévoit aussi une amélioration fondamentale de l'interaction homme-machine en permettant à la machine de "voir", de "lire", d'"écouter" et de "parler".

Le projet OPALE dans le cadre duquel cette thèse est préparée, vise à la définition et à la réalisation d'une machine gérant les bases de connaissances, on la nomme donc "machine base de connaissances". Le langage Prolog a été choisi comme point de départ. Ce langage est considéré tant comme le formalisme de spécification des connaissances que comme un mécanisme de manipulation des connaissances. Deux problèmes principaux sont étudiés: la sélection des connaissances par un opérateur de filtrage et le parallélisme dans la machine.

Cette thèse aborde divers problèmes autour du parallélisme. Nous allons définir un modèle d'interprétation parallèle de Prolog. Ce modèle repose sur la notion de processus et s'adapte particulièrement aux bases de connaissances qui ont un grand volume de connaissances à traiter. En partant du modèle, nous allons déduire l'architecture qui supporte le mieux ce modèle en introduisant la notion d'architecture orientée processus. Sous l'hypothèse de l'architecture orientée processus qui a une caractéristique multi-processeurs, le problème de la distribution des processus a été étudié et expérimenté. L'expérimentation sur le modèle par simulation en langage OCCAM

montre que les parallélismes exploités dans le modèle et un environnement multi-processeurs accélèrent effectivement le traitement. Le développement du parallélisme influence aussi le langage Prolog lui même. Nous présenterons également la contribution apportée au langage Prolog.

Le premier chapitre est une présentation générale et préliminaire. Le langage Prolog est d'abord présenté avec des exemples simples. Ensuite vient une présentation de l'évolution de systèmes de gestion des données et des connaissances. Puis sont dégagés le but et les principaux problèmes de systèmes de bases de connaissances. Enfin sont présentées les grandes lignes du projet OPALE et de la thèse.

Dans le deuxième chapitre, les différents problèmes concernant Prolog et le parallélisme sont dégagés et les différentes approches pour une exécution parallèle de Prolog sont présentées. L'analyse des expériences acquises nous permet de définir notre modèle dans le contexte base de connaissances.

Le chapitre III contient une partie importante de cette thèse. Le modèle d'interprétation parallèle de Prolog est défini. Le modèle contient un algorithme de base qui traite les clauses non-récursive et un algorithme de traitement des clauses récursives. Tous ces algorithmes sont spécifiés à l'aide de la notion de processus.

Dans le chapitre IV, nous discutons l'architecture de la machine. Nous définissons d'abord un type d'architecture multi-processeurs nommé "architecture orientée processus". Ensuite nous présentons les principaux problèmes de ce type d'architecture à travers un exemple de langage concurrent orienté objet. Enfin nous discutons l'architecture de la machine OPALE. L'intérêt est concentré sur le problème de la relation entre le modèle et l'architecture.Une expérimentation sur l'algorithme de dispatching de processus est présentée avant la conclusion.

Le chapitre V présente l'expérimentation effectuée en langage parallèle OCCAM en vue de la

validation du modèle. Le système qu'on réalise et les exemples sont décrits. Les résultats de la simulation montrent que le modèle dans une machine multi-processeurs accélère effectivement le traitement.

Le dernier chapitre, le chapitre VI, présente la contribution apportée au langage Prolog. Un système Prolog avec la capacité d'accès aux connaissances en mémoire secondaire par un algorithme du filtrage est présenté. La réalisation de cette maquette nous permet d'envisager la réalisation d'une version logicielle de la machine OPALE.

Enfin, nous donnons la conclusion de la thèse.

CHAPITRE I

INTRODUCTION

CHAPITRE I

INTRODUCTION

L'intelligence artificielle est un des domaines de l'informatique. Le but de l'intelligence artificielle est de construire des systèmes de plus en plus intelligents, dans le sens qu'une machine ait la capacité humaine de:

- Perception: acquérir des connaissances par ses organes de vision, de lecture,...
- Apprentissage: enrichir sa base de connaissances au fur et à mesure de l'évolution de l'environnement dans lequel elle se situe.
- Résolution: déduire en se basant sur des connaissances accumulées.
- Communication: échanger des connaissances avec d'autres systèmes.
- Planification: générer les actions successives à suivre pour atteindre un but suivant les connaissances accumulées.
- Action : accomplir une action dans le monde extérieur.

L'intelligence artificielle peut être aussi définie restrictivement comme "la capacité de résolution de problèmes en appliquant des règles d'inférence à des bases de connaissances". On constate bien que dans toutes les fonctionnalités citées ci-dessus, les "connaissances" jouent un rôle majeur.

Si on voit un ordinateur conventionnel comme une unité centrale faisant des opérations en fonction des instructions et des données stockées dans sa mémoire, pour la machine de nouvelle génération, c'est un mécanisme d'inférence fonctionnant selon les connaissances stockées dans une base de connaissances.

La machine de nouvelle génération, qui est le but de plusieurs projets ambitieux (FJCS, ESPRIT, ALVEY...), peut être schématisée par certains niveaux des recherches présentées dans la Fig.I.1.

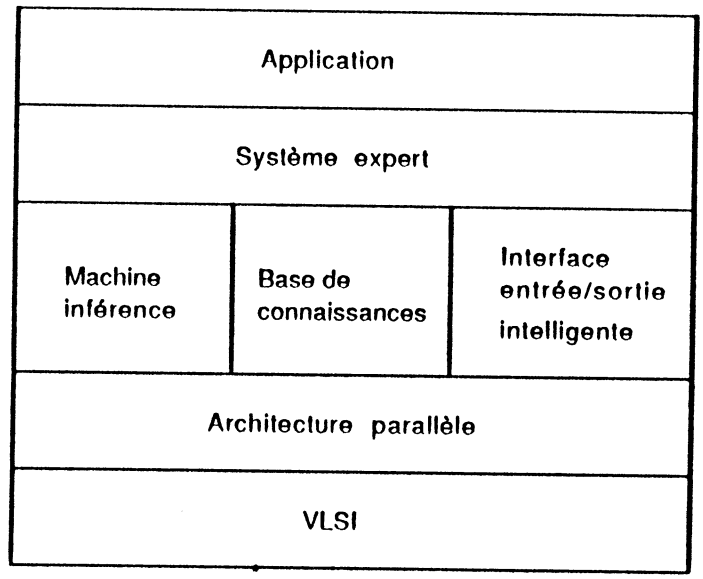

Fig.I.1

Les applications principales prévues dans la prochaine décennie sont des systèmes experts basés sur des connaissances. La machine de type calcul scientifique sera remplacée par la machine à inférence (M.I.), et la machine base de connaissances prendra en charge la gestion des connaissances en coopérant avec M.I. L'interface intelligent entrée/sortie améliore essentiellement la communication homme/machine, il fournira le moyen d'accepter les connaissances humaines comme son, image, langage naturel etc. Les supports technologiques restent toujours en "VLSI" mais avec une intégration plus importante.

En se basant sur ces hypothèses, l'architecture de la machine nouvelle génération sera parallèle car la technologie du VLSI prévue pour les années 90 ne peut pas satisfaire les besoins en vitesse. Il est généralement admis par les informaticiens que le développement du parallélisme fournira une accélération en exécution des programmes.

Le projet OPALE fait partie de cette tendance et s'est fixé pour but la construction d'une

machine base de connaissances. Ce chapitre est une présentation générale du langage Prolog, le point de départ du projet; des bases de connaissances correspondant aux fonctionnalités envisagées par la machine; et de la thèse.

I.1 PROLOG

I.1.1 Le langage

Basé sur l'idée de la programmation en logique fondée par Kowalski [Kol74], le premier interpréteur Prolog a été réalisé par Colmerauer [Col77]. Aussitôt après, beaucoup d'études ont été faites, et en 1980, Prolog a été choisi comme langage de base pour le projet japonais d'ordinateurs de cinquième génération.

Un programme Prolog est composé de deux parties: une base de faits et un groupe de règles de déduction.

Exemple I.1: routage

Un graphe de route est présenté dans la Fig.I.2, les noeuds a,b,c représentent des villes, la fleche a$\overset{x}{\text{---}}$>b représente la route uni-directionnelle reliant les villes a et b avec une distance de x kilomètres.

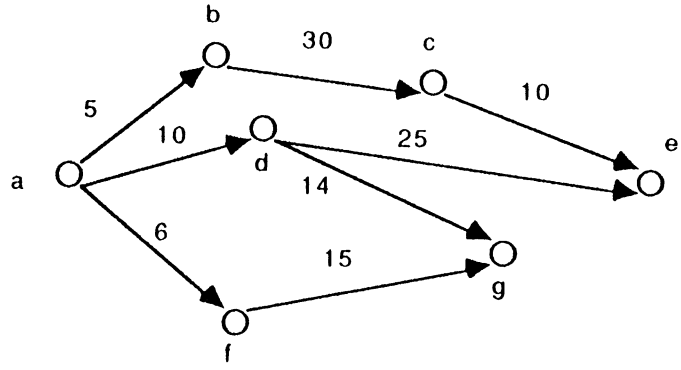

Fig.I.2

Une route peut être exprimée par une clause unaire. Si on veut aller d'une ville à une autre, soit il existe une route directe, soit il existe des villes intermédiaires par lesquelles on

peut passer pour arriver à destination.

```
route(a,b,5).        route(d,g,14).
route(b,c,30).       route(d,e,25).
route(c,e,10).       route(a,f,6).
route(a,d,10).       route(f,g,15).

aller(X,Y,DIS) :- route(X,Y,DIS).
aller(X,Y,DIS) :- route(X,Z,D1), aller(Z,Y,D2), DIS is D1+D2.
```

Différentes questions peuvent être posées à ce programme:

Q1: Existe il une route reliant a et f ?

Q2:quelles sont les villes distantes de 10 kilomètres ?

Q3: Peut-on aller de a à e ?

Les questions et les réponses du système sont les suivantes:

Questions	Réponses du système
:- route(a,f,X).	X=6
:- route(X,Y,10).	X,Y=a,d; c,e
:- aller(a,e,DIS).	DIS=45; 35

Les données manipulées en Prolog sont essentiellement des arborescences sous forme préfixée (Fig.I.3).

L'opération unique en prolog est l'unification, qui consiste à comparer deux arborescences et à les rendre égales en établissant des substitutions pour les variables. Dans l'exemple suivant (Fig.I.4), les substitutions [X/c(b,d)] et [Y/a] sont générées.

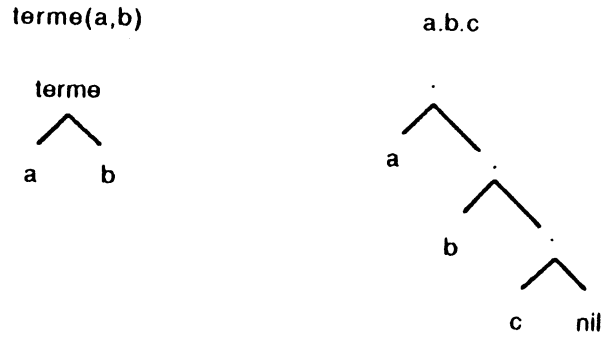

Fig. I.3

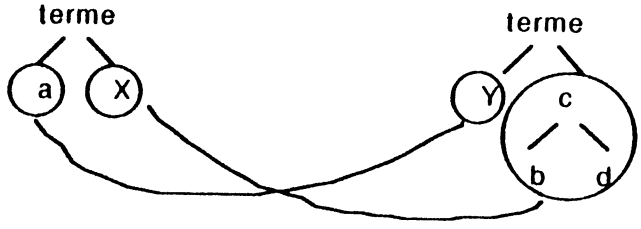

Fig. I.4

I.1.2 base théorique

Une caractéristique essentielle de Prolog est qu'il a une base mathématique solide. Le langage est fondé sur la logique du premier ordre. Cette dernière peut être définie comme un système formel reposant sur un langage de l'objet, l'interprétation des formules du langage et une théorie de preuve.

A. Langage de l'objet:

-Alphabet:
* a,b,c... pour constantes;
* x,y,z... pour variables;
* connecteurs logiques: ¬, =>, &, V, <=>;
* symboles de fonction: f, g, h...
* symboles de prédicat: P, Q, R...

- Une variable est dite liée si elle est dans la portée d'un qualificateur, sinon elle est dite libre.

- Formule bien formée (FBF):

  * terme: une constante ou une variable est un terme.

    si f est un symbole fonctionnel n-aire et t1...tn des termes, alors f(t1,...tn) est un terme.

  * formule atomique: si P est un symbole de prédicat à n places et t1...tn des termes, alors P(t1,...tn) est une formule atomique.

  * littéral: si F est une formule atomique alors F et ¬F sont des littéraux.

  * FBF: une formule atomique est une FBF;

    si F et G sont des FBF, alors ¬F, F&G, FVG, F=>G, F<=>G sont des FBF;

    si x est une variable et F une FBF, alors (∃ x)F, (∀x)F sont des FBF.

- Une FBF est dite prenexe si tous les qualificateurs se trouvent devant la formule. Il existe une méthode systématique de transformation d'une FBF en forme prenexe.

- Une clause est une FBF qui ne contient que des disjonctions de littéraux. Il existe une méthode systématique qui transforme une FBF en une clause. Une clause ¬A1V...V¬AnVB1V...VBm est équivalente à A1&...&An=>(B1V...Bm). Quand m=0 ou m=1, la clause est dite clause de Horn.

B. Sémantique: Interprétation

Une interprétation d'un groupe de FBF consiste en une spécification d'un ensemble non vide E, à partir duquel chaque constante et variable est associée à une valeur. Une fonction n-aire est assignée par E^n ->E. Un prédicat est assigné par une relation E^n et une valeur de vérité.

Chaque FBF peut avoir une valeur de vérité, la valeur est calculée en respectant les règles suivantes:

- Si F est vraie alors ¬F est fausse.
- F1&F2 est vraie ssi F1 et F2 sont vraies.
- F1VF2 est vraie ssi F1 ou F2 est vraie.
- $(\forall x)F(x)$ est vraie si tous les éléments x de E vérifient F.
- $(\exists x)F(x)$ est vraie s'il existe une valeur x appartenant à E qui vérifie F.

Un modèle d'un ensemble de FBF est une interprétation dans laquelle toutes les FBF sont vérifiées vraies. Une FBF est dite conséquence logique d'un groupe de FBF W ssi elle est vérifiée vraie dans tous les modèles du groupe W.

C. Syntaxe: théorie du premier ordre

Une théorie du premier ordre est constituée d'un schéma d'axiomes et de deux règles d'inférence. L'approche syntaxique concerne la déduction, c'est à dire la dérivation d'une nouvelle FBF à partir d'un ensemble de FBF. Le schéma d'axiomes est un ensemble d'axiomes qui est vérifié vrai dans toutes les interprétations. Les deux règles de déductions sont les suivantes:

- Modus ponens: A et A=>B alors B.
- Généralisation: A alors $(\forall x)A$.

Basé sur un ensemble d'axiomes, le mécanisme d'inférence peut aboutir à d'autres formules appelées théorèmes. Une formule qui n'est pas démontrable dans une théorie peut être ajoutée comme axiome, et ceci donne une nouvelle théorie.

Le principe de résolution, introduit par Robinson [Rob65], est une règle de déduction

Le principe de résolution, introduit par Robinson [Rob65], est une règle de déduction supplémentaire. Elle consiste à générer des nouvelles FBF à partir d'un ensemble de FBF. Elle s'applique à la forme clausale. L'exemple suivant donne une explication:

Si C1: $\neg$P(a,b,c) V Q(a,Z) et C2: P(X,Y,Z) V R(X,Y) alors

C3: Q(a,c) V R(a,b) avec des substitutions [X/a,Y/b,Z/c].

I.1.3 Les caractéristiques

Prolog a des caractéristiques très interéssantes pour plusieurs aspects. C'est la raison pour laquelle le projet OPALE l'a pris comme point de départ.

D'abord, l'unification est un concept très puissant. Elle peut être considérée comme un appel d'une procédure dirigé par "pattern matching", une activation de processus, un passage de messages, une vérification ou une recherche associative.

Ensuite, au niveau de la représentation de connaissances, la logique mathématique est un formalisme pour modéliser des pensées humaines. L'expression des faits élémentaires et des règles de déduction en Prolog a fourni une méthodologie simple et puissante qui facilite le transfert de connaissances de l'humain à l'ordinateur.

Enfin, en tant que langage de programmation, Prolog a des caractéristiques très différentes de celles des langages classiques.

- Langage de très haut niveau : il s'agit de déclarativité. On spécifie "ce" qu'il faut faire, et non pas "comment" il faut le faire.
- uniformité : Le calcul de la logique du premier ordre est un sur-ensemble du calcul relationnel, donc Prolog peut être utilisé comme un langage de programmation, de définition, et de manipulation de données pour les bases de données.

- Parallélisme : Plusieurs formes de parallélisme peuvent être dégagées pour l'interprétation des langages logiques. Ceci permet l'exécution dans un environnement parallèle et concurrent.

I.2 Base de connaissances

I.2.1 De "bases de données" à "bases de connaissances"

A. Bases de données: Les bases de données sont nées pour répondre au besoin de stockage de grands volumes d'informations, le but étant d'assurer une indépendance maximale entre la description logique des données et leur représentation physique.

Depuis les années soixante, plusieurs modèles sont proposés. Parmis eux, les plus célebres sont les modèles réseaux, hiérarchique et relationnel; le plus réussi est le modèle relationnel [COD70]. Il a fourni un modèle pour représenter les informations sous forme de tableaux, et une algèbre relationnelle définissant des opérations sur les tableaux. Le stockage physique des données peut être structuré indépendamment de leur représentation logique. En résumé, un système de gestion de base de données a les fonctionnalités suivantes:

- Fournir un modèle de données permettant de modéliser le monde réel.
- Améliorer l'indépendance logique et physique.
- Fournir un langage de haut niveau à l'utilisateur.
- Optimiser les accès à la base.
- Améliorer l'intégrité et la confidentialité.

Dans une base de données, les informations stockées sont les suivantes:

1) Les données élémentaires qui modèlisent le monde extérieur. Les types de données sont souvent simples, par exemple l'atome, le chiffre et la chaîne de caractères.

2) Les contraintes d'intégrité qui représentent les conditions que doivent respecter les données élémentaires pour assurer la cohérence des données.

B. Base de données déductive: La base de données déductive repose sur le principe de la logique. Une base de données peut être modélisée par la logique du premier ordre en suivant deux approches différentes:

Base de données = interprétation d'une théorie du premier ordre.

Base de données = théorie du premier ordre.

Dans la première approche, les contraintes d'intégrité constituent une théorie qui doit être vérifiée par la base de données, c'est à dire que la base de données doit être un modèle des contraintes d'intégrité. Des requêtes sont des formules à évaluer. L'évaluation d'une requête suit l'approche interprétation pour voir si la requête est une conséquence logique dans le modèle.

Dans la deuxième approche, des requêtes sont considérées comme des théorèmes à prouver. L'évaluation est entreprise par une approche syntaxique, c'est à dire la génération d'une nouvelle formule à partir du système actuel. Dans cette approche, il est possible d'ajouter dans la base de données, en plus des données élémentaires et des contraintes d'intégrité, des règles de déduction. Avec ce troisième élément, une base de données peut contenir implicitement des informations. Une recherche sur des informations implicites invoque une opération de déduction qui applique les règles de déduction, à la base de données qui contient les informations explicites.

C. Base de données généralisée: Une autre tendance de l'évolution des bases de données est de manipuler des données à structure complexe. En effet, jusqu'à maintenent les bases de données étaient développées pour des applications de gestion (personnel, financière, production...). Des nouvelles applications en CAO (en architecture, VLSI...) et les nouveaux types de données à stocker (son, image...), exigent la manipulation d'objets complexes. Ceci a amené une évolution des bases de données en bases de données généralisées.

D. Base de connaissances: Dans les années 80, les systèmes experts ont eu beaucoup de succès, et on estime qu'ils seront l'application principale de l'intelligence artificielle dans la

prochaine décénnie. Un système expert est basé sur une représentation particulière des connaissances qui constitue une base de connaissances, et un moteur d'inférence qui explore ces connaissances.

Un effort a été fait tant du côté du système expert que du côté des bases de données pour profiter des avantages des deux types de systèmes. L'idée initiale est d'installer une interface entre les deux, par exemple entre Prolog et une base de données relationnelle.

Etant donné que l'étude formelle sur les "connaissances" est encore loin de fournir un formalisme général [Ros85], chaque base de connaissances concrète peut prendre un formalisme particulier, propre à son système (logique, objet etc). Donc, on ne peut donner qu'une présentation de la philosophie par une comparaison des systèmes de gestion de base de données (SGBD), systèmes experts (SE) et systèmes de gestion de base de connaissances (SGBC):

BUT:

SGBD: recherche des informations.

SE : inférence sur des connaissances.

SGBC: gestion des connaissances à fin de déduction.

STRUCTURE DE DONNEES MANIPULEES:

SGBD: atomique.

SE : structure complexe mais unique, propre à chaque système.

SGBC: structures complexes et multiples, possède une certaine généralité.

INFORMATIONS STOCKEES:

SGBD: données volumineuses et contraintes d'intégrité.

SE : données, règles de productions et méta-règles.

SGBC: données volumineuses, contraintes d'intégrité, règles de déduction, objets complexes.

En dehors de ces différences, la table I.1 donne d'autres caractéristiques représentatives des trois systèmes.

I.2.2 Base de connaissances Prolog

La base de connaissances Prolog (BCP) est un type de base de connaissances qui repose sur le langage Prolog. Ceci veut dire que la représentation de connaissances est sous la forme logique, et les fonctionnalités citées ci-dessus sont réalisées aussi par Prolog.

FONCTIONALITES	SGBD	SE	SGBC
méthode d'accès	oui	non	oui
langage de requête	oui	non	oui
accessible par langage naturel	non	oui	?
explication	non	oui	?
partage des informations	oui	non	oui
protection	oui	non	oui
sécurité	oui	non	oui
raisonnement incertain	non	oui	?
heuristique	non	oui	?

Table I.1

La logique est considérée comme une des méthodes les plus importantes pour la représentation des connaissances [Lau83]. D'autres formalismes, comme ceux orientés objets, sont faciles à transformer ou convertir en forme logique [Nil80]. Quant au mécanisme d'exploitation des connaissances, les études menées au cours de ces années, surtout celles de Prolog concurrent [Sha81a], ont prouvé la possibilité d'utiliser Prolog comme langage système. Un exemple de gestion de connaissances au niveau d'acquisition est donné ci-dessous. Il montre comment Prolog réalise le contrôle de redondance et de cohérence.

Exemple I.2:

Une base de données est présentée par des clauses bd(_). Des contraintes d'intégrité sont représentées par bd(ic(clause,PG-->PD,message)), PD représente la contrainte qui doit être vérifiée fausse si PG est vérifiée vraie dans la base de données. Le prédicat "demo" peut être considéré comme un interpréteur Prolog qui vérifie son argument dans un ensemble de clauses acquises.

```
%%%%%%%% contrôle de déductibilité
assimil(Input) :- demo(Input),
                 write('input is deducible from knowledge base').

%%%%%%%% contrôle de redondance
assimil(Input) :- bd(X), nonredondante(X,Input),fail.
nonredondante(X,Input) :- retract(bd(X)),
                          assert(bd(Input)),
                          demo(X),!,
                          retract(bd(Input)).
nonredondante(X,Input) :- retract(Input),
                          assert(X).

%%%%%%%% contrôle de contradiction
assimil(Input) :- assert(bd(Input)),
                  bd(ci(Input,IC,Mesg)),
                  ic_trans(IC,ICR),
                  Check_IC=..[demo,ICR],
                  Check_IC,
                  write('input conflicts with ic'),
                  write(Mesg),
                  retract(Input).
```

```
ic_trans((ICP-->ICQ),(ICP,not(ICQ))).
ic_trans((ICA,ICB),(ICAM,ICBM)) :- ic_trans(ICA,ICAM),ic_trans(ICB,ICBM).
```

Prolog est encore en forte évolution, les principales recherches s'effectuent sur l'environnement de programmation Prolog [ChM85], le parallélisme [KFT84], les facilités pour la construction de systèmes experts [ClM82], et les machines Prolog[OAS85,NKH85,DDP85]. Les résultats de ces recherches permettront d'avoir un langage performant pour la gestion de bases de connaissances.

I.3 Le sujet de la thèse

La thèse est préparée dans le cadre du projet OPALE qui a été lancé en 1982 [BeN82]. Le projet OPALE (Organisation PArallele et LogiquE) vise à la conception et la réalisation d'une machine base de connaissances reposant sur le langage Prolog. L'idée initiale est d'utiliser Prolog tant comme un modèle de données pour la représentation des connaissances, que comme un mécanisme d'inférence pour exploiter les connaissances. Les recherches menées s'appuient principalement sur deux axes:

- Unification: l'unification a été décomposée en deux parties, pré-unification et association. Cette décomposition permet d'envisager un opérateur spécifique réalisable en VLSI qui pourra effectuer la pré-unification entre un ensemble de buts et un flux de données. Cet opérateur sera comme un filtre de données situé devant les disques sur lesquels sont stockées les clauses afin de sélectionner les connaissances.

- Architecture parallèle: le parallélisme est considéré comme une issue principale pour des performances satisfaisantes. Il s'agit de la définition d'un modèle parallèle pour l'interprétation de Prolog, celui-ci devant tenir compte de la stratégie de recherche exigée par le filtre dans un environnement de traitement ensembliste. La définition du modèle parallèle doit être effectuée

interactivement avec le développement de l'architecture qui le supporte.

Pour l'ensemble du projet et les recherches sur la premiere partie, il convient de se reporter aux publications suivantes: [IAN85], [BDI85], [BDI84]. Cette thèse aborde essentiellement la deuxième partie.

En analysant les besoins d'une machine base de connaissances Prolog, et le mécanisme d'inférence de Prolog, la stratégie de recherche classique (de gauche à droite et en profonteur d'abord) a été abandonnée. Une nouvelle stratégie basée sur un traitement ensembliste a été développée. Cette stratégie fait partie intégrante du modèle d'interprétation parallèle d'OPALE.

Pour avoir une uniformité avec le langage Prolog, une méthode de coopération a été développée et implémentée autour d'un interpréteur Prolog. Ceci permet à un interpréteur Prolog classique d'avoir un moyen de gérer des clauses en mémoire secondaire.

Basée sur le modèle développé, l'architecture de la machine a été étudiée. Une architecture multi-processeurs a été proposée. Une expérimentation sur le modèle entier a été réalisée en langage OCCAM. Il s'agit de la définition d'un algorithme de dispatching qui gère les processus parallèles. Cette expérimentation a donné des résultats satisfaisants.

L'idée générale du modèle parallèle pour l'interprétation de Prolog dans OPALE est due à G.Berger Sabbatel. L'étude de l'algorithme d'unification a été effectuée par S.C. Ianeselli.

La contribution de l'auteur a porté plus particulièrement sur les points suivants :

- préciser, optimiser, implémenter et évaluer les algorithmes d'interprétation parallèle.
- étudier les problèmes liés à la communication inter-processeurs, et à l'allocation des processeurs aux processus d'évaluation, et préciser sur ce point l'architecture d'Opale.
- étudier le problème de l'interprétation des clauses récursives.

CHAPITRE II

PROLOG ET PARALLELISME

CHAPITRE II

PROLOG ET PARALLELISME

Dans le passé, l'amélioration de la vitesse du calcul a été réalisée principalement par l'avancement de la technologie du circuit intégré. Cette dernière ne peut plus satisfaire le besoin de l'informatique de nouvelle génération (Fig.II.1). C'est la raison pour laquelle les chercheurs se tournent vers le développement du parallélisme.

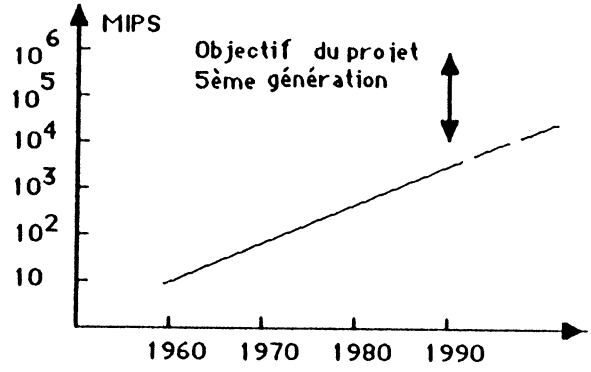

Fig.II.1

Le parallélisme est considéré comme une issue générale pour accélérer l'exécution d'un programme. La possibilité d'expression du parallélisme est un critère important d'un langage de programmation. Le développement du parallélisme pour Prolog peut s'effectuer sur trois niveaux:

- Le langage: En tant que langage de programmation, l'introduction de parallélisme est très importante car il s'agit de la possibilité de résolution des problèmes concurrents d'une part, et du point de départ pour le développement de la machine langage d'autre part.

- Le modèle d'interprétation: Il existe plusieurs types de parallélismes quant à l'interprétation de Prolog:

* OU-parallélisme qui consiste à évaluer les différentes clauses alternatives en même temps.
* ET-parallélisme qui consiste à vérifier les prédicats dans la queue d'une même clause.
* Parallélisme d'accès qui consiste à diviser les clauses ayant un même entête en plusieurs groupes et à évaluer un but simultanément dans chaque groupe. C'est le cas quand les clauses sont stockées en mémoire secondaire.
* "Stream" parallélisme qui consiste à évaluer plusieurs buts sur un flux de données.

Un modèle de l'interprétation parallèle consiste à développer ces parallélismes par des processus parallèles communiquant, ou par des opérateurs spéciaux en vue de dataflow.

- Architecture: Il s'agit du développement de la machine parallèle supportant le langage et le modèle d'interprétation.

Ce chapitre présente les recherches principales dans ces branches. Une conclusion est retenue à la fin du chapitre.

II.1 Prolog - langage avec parallélisme

II.1.1 Parlog

PARLOG est un langage de programmation logique parallèle developpé par CLARK [ClG83]. Dans ce langage, AND-parallélisme et OU-parallélisme peuvent être spécifiés explicitement dans des clauses.

Chaque clause doit être déclarée soit sous **and-relation**, soit sous **or-relation**. Dans une and-relation, les différentes clauses alternatives doivent être séparées explicitement, par les

operateurs "." et ";", afin de distinguer la vérification parallèle ou séquentielle de ces clauses. Dans chaque clause, les littéraux dans la queue sont connectés par des connecteurs "," et "&" qui spécifient l'évaluation de ces littéraux en parallèle ou séquentiellement. La vérification d'un prédicat concernant une and-relation n'obtient qu'une seule solution. La notion des modes entrée/sortie pour des variables et de "garde" est adoptée pour faciliter le contrôle de processus.

Exemple II.1 : définition d'une and-relation.

```
relation append(?,?,^ )
append([],Y,Y).
append([U|X],Y,[U|Z]) :- append(X,Y,Z)
```

Dans cet exemple on définit une and-relation. Le symbole "?" énonce une variable en mode entrée et "^" en mode sortie. Le séparateur "." signifie que deux clauses seront vérifiées en parallèle.

Dans une or-relation, toutes les clauses alternatives seront appelées en parallèle et toutes les solutions seront acquises à travers une **expression ensembliste**. Mais la vérification des littéraux dans la queue d'une clause est toujours séquentielle. L'expression ensembliste joue un rôle d'interface entre deux types de relation.

Exemple II.2: définition d'une or-relation.

```
or-relation employe
employe(departement1,smith)
employe(departement1,jones)
employe(departement2,brown)
:::
```

C'est la définition d'une or-relation. Un appel par le biais d'expression ensembliste {<Y> :

employe(departement1,Y)} lance une évaluation parallèle qui effectue l'unification du but "employe(departement1,Y)" avec toutes les clauses spécifiées dans la or-relation. Y sera liée avec [<smith>,<jones>] dans cet exemple.

II.1.2 Prolog concurrent

Prolog concurrent a été developpé par E.Y.Shapiro [Sha83a]. La sémantique opérationelle associée au Prolog concurrent donne un langage très différent de Prolog classique.

Une clause peut être écrite sous trois formes:

(1) P :- G1,...,Gn | B1,...,Bm.

(2) P :- B1,...,Bm.

(3) A.

où " | " est le symbole de "garde", dans la deuxième clause le symbole " | " se trouve implicitement avant B1.

Dans ce langage, un but à vérifier est un processus. La seule opération possible consiste à réduire un processus à d'autres processus. La clause (1) signifie que "pour réduire le processus P, d'abord réduire G1,...,Gn à vide, et puis réduire P à B1,...,Bm". La clause (3) signifie que "le processus A est réduit à vide". La clause (2) est un cas particulier de la clause (1) où les Gi sont déjà vides. L'exécution d'un programme consiste à réduire un ensemble de processus, le but étant d'arriver à un ensemble vide.

Dans un système Prolog concurrent, tous les processus sont exécutés en parallèle. Le contrôle de processus se fait par des variables en mode "lecture seule" spécifiée par le symbole "?" attaché au nom de la variable. C'est à dire que un processus contenant une variable en mode "lecture seule" ne peut être vérifié que si cette variable est liée. Donc, un processus contenant une variable libre en mode "lecture seule" doit attendre un autre processus qui lie cette variable avec un terme pour son

exécution.

Exemple II.3 :

programme : process1(terme).

process2(terme).

requête : process1(X?),process2(X).

Pour cette requête, les réductions de process1 et de process2 sont lancées en parallèle, mais process1 ne peut commencer sa réduction qu'après que process2 donne X liée à "terme".

II.1.3 KL1

Le langage KL1 (kernal language version 1) [KFT84] a été spécifié par les chercheurs de l'ICOT (Institut of new COmputing Technology). Ce langage a combiné les notions principales de Parlog et de Prolog concurrent. En plus, un nouveau type de contrôle a été introduit par la variable en mode "écrit en avant". Quand une variable de ce type est liée par une unification, la substitution peut être communiquée avant d'atteindre la "garde" de la clause. Ceci permet de déclencher un autre processus contenant cette variable en lecture seule dès que l'unification réussit, au lieu d'attendre l'exécution de la "garde" qui communique les substitutions au processus concerné.

KL1 a été développé dans le cadre du projet japonnais de cinquième génération. Il est considéré comme le langage de base pour la machine Prolog parallèle. Le développement de son successeur, KL2, a été commencé. L'évolution du projet nous permet d'avoir plus d'informations sur l'ensemble du projet, tant au niveau du langage qu'au niveau de la machine.

II.2 Modèle d'interprétation parallèle

II.2.1 Modèle dataflow

Un modèle dataflow pour Prolog a été décrit par [Kac84]. Ce modèle transforme un

programme Prolog en un graphe composé de quatre types d'opérateurs:

- UN pour effectuer l'unification,
- AND pour relier les littéraux dans la queue d'une clause,
- OR pour relier les différentes clauses alternatives d'un but,
- UT pour les clauses unaires.

Exemple II.4 :

a(X,Y) :- b(X,Y),c(X),d(Y).

b(orange,apple).

b(orange,lemon).

b(plum,apple).

c(orange).

d(lemon).

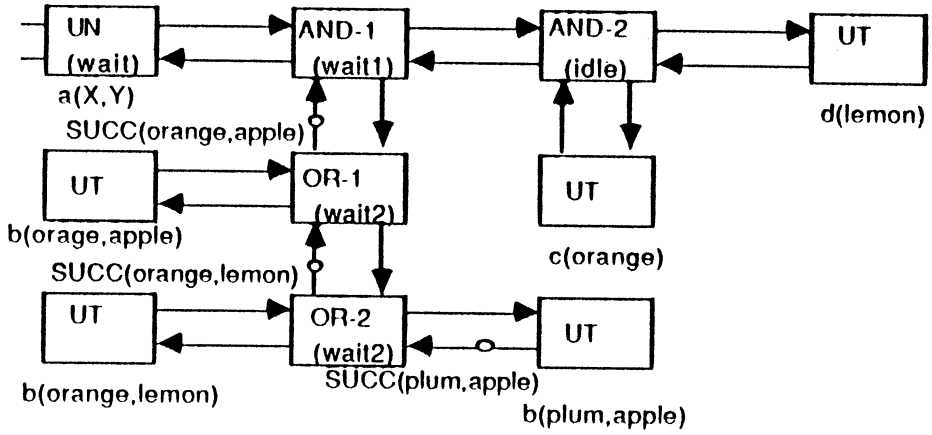

Fig.II.2

Le programme est représenté sous forme de graphe dans la Fig.II.2. Les messages de contrôle (token) sont:

- DO (vérifier un littéral),
- REDO (chercher une autre alternative),

- SUCC (succès),
- FAIL (échec).

Ces messages circulent dans le graphe avec les substitutions générées quand l'unification réussit. La Fig.II.2 montre que pour la requête ?- a(X,Y) lancée, le noeud AND-1 attend un message; OR-1 a reçu un message SUCC(orange,apple) et le renvoie pour AND-1; OR-1 et OR-2 ont stocké chacun un message qui sera retourné dès que cela sera nécessaire (REDO).

[UmT83] a établi un autre modèle dataflow. Ce modèle utilise cinq types d'opérateurs:

1) Unificateur (Fig.II.3) qui a deux entrées, l'entrée gauche stocke simplement tous les messages qui arrivent , l'arrivée d'un message à l'entrée droite déclenche une unification entre ce message et un des messages accumulés par l'entrée gauche. Si l'unification réussit, les deux messages sont renvoyés par deux sorties, sinon ils seront détruits.

2) Copieur (Fig.II.4) qui a une entrée et plusieurs sorties. Il copie tous les messages qui arrivent et les renvoie sur chaque sortie.

3) Collecteur (Fig.II.5) qui a plusieurs entrées mais une seule sortie. Il renvoie les messages arrivés un par un sur la sortie.

4) Entrée (Fig.II.6) et Sortie (Fig.II.7) qui se présentent toujours en couple. Ils gèrent les messages entrées/sorties pour une clause.

Avec ces cinq opérateurs, un programme Prolog peut être transformé en un graphe. Un appel à une clause correspond à l'envoi d'un message. L'unification est effectuée par l'unificateur, et la gestion des messages est réalisée par l'ensemble des opérateurs.

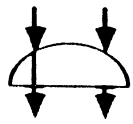

Fig.II.3

Fig.II.4

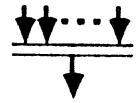

Fig.II.5

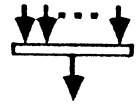

Fig.II.6

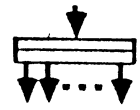

Fig.II.7

Exemple II.5 : Une clause représentée par un graphe dataflow.

La clause gp(X,Y) :- p(X,Z),p(Z,Y). est représentée par le graphe dataflow dans la figure.II.8.

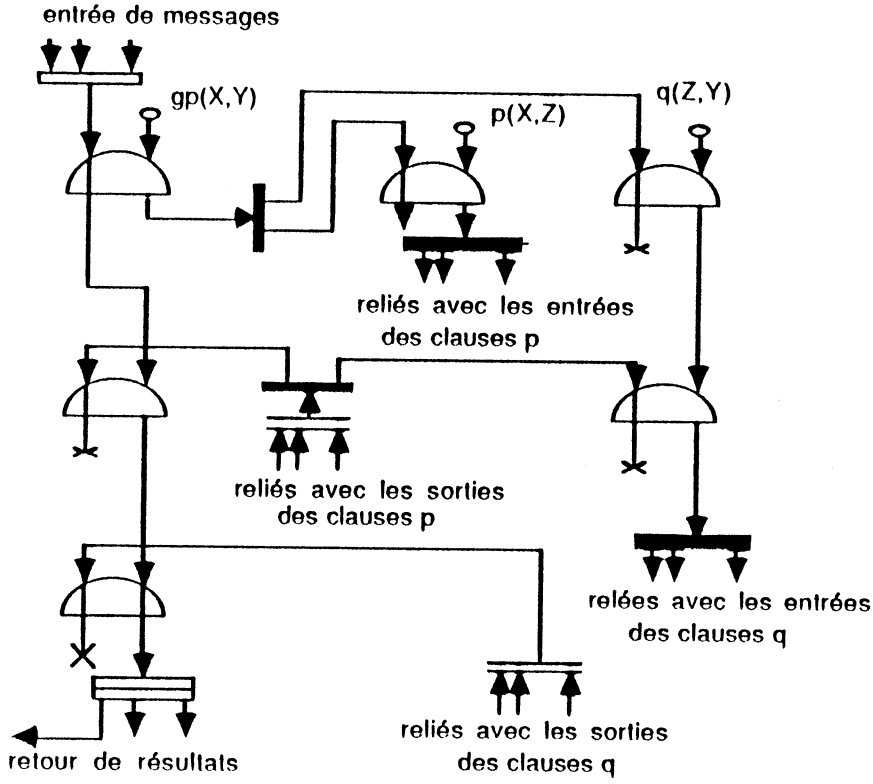

Fig.II.8

Le langage Prolog a une caractéristique non-déterministe. L'unification est une opération bidirectionnelle. Ces deux propriétés impliquent que la représentation du programme Prolog sous la forme de dataflow est compliquée. La complexité de l'unification est la difficulté principale pour réaliser un modèle avec des opérateurs simples. Le gain dû au parallélisme risque d'être pénalisé à cause de la gestion des messages et celle de la communication.

II.2.2 Modèle de processus

L'idée principale du modèle de processus est de modéliser une résolution en un groupe de processus parallèles communicants. Ce type de modélisation a été établi par [CoK81]. Dans son modèle, un but à vérifier correspond à un **OU-processus**, il crée des sous-processus nommés **ET-processus** chacun correspondant à une clause candidate pour la vérification du but. Les ET-processus, à leur tour, créent des OU-processus pour chaque prédicat de la queue de la clause. Un ET-processus ne réussit que si tous ses processus-fils réussissent; tandis qu'un OU-processus ne réussit que si un des ses processus-fils réussit. Les substitutions générées par l'unification sont transmises par des messages. Comme Prolog peut être modélisé par un arbre ET/OU, les relations entre les processus ont une structure correspondante.

Dans ce type de modèle, les principaux problèmes sont le contrôle de la prolifération des processus et la réduction de la communication entre les processus. Différents algorithmes sont proposés pour la gestion de substitutions [CiH83] [Bor84] [Lin84]. Ces algorithmes consistent à développer la technique évitant la copie totale des substitutions générées de processus père aux processus fils.

La technique en vue de l'utilisation du ET-parallélisme a été étudiée par [CoK85]. L'exécution parallèle des littéraux dans la queue d'une clause est basée sur une analyse dynamique en temps d'interprétation selon les relations production/consommation des variables dans la clause. Pour avoir un maximum de ET-parallélisme, les littéraux sont ré-ordonnés. Chaque backtrack entraîne un nouveau réordonnancement.

II.2.3 Modèle de multi-Prolog machine

Comme cela a été décrit auparavant, une interprétation de Prolog peut être vue comme un parcours dans un arbre ET-OU. Les ET-noeuds représentent les buts à vérifier quand la vérification du but actuel est réussie, les OU-noeuds représentent les points de backtrack si un échec est

rencontré. La trace de vérification est conservée dans une pile. Le modèle de multi-Prolog machine est basé sur une répartition dynamique d'arbre à parcourir.

Exemple II.6: une répartition d'un programme.

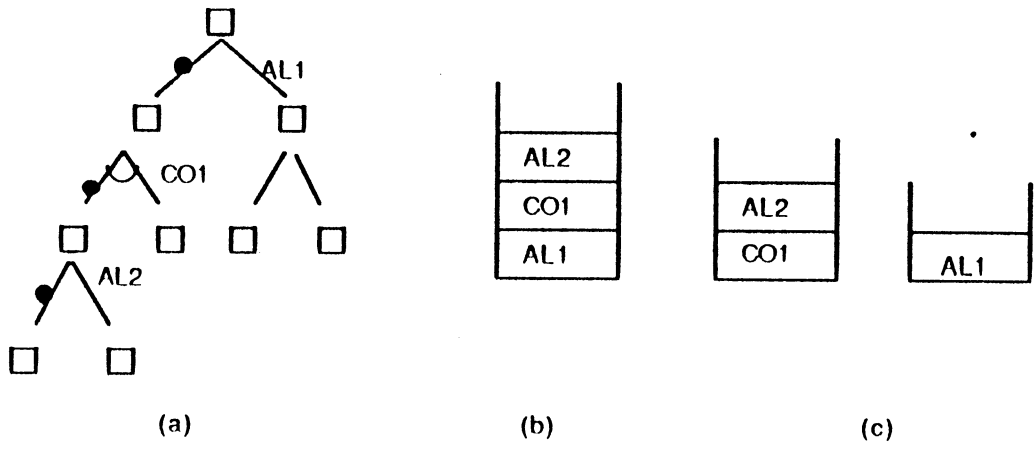

Fig.II.9

La Fig.II.9 (a) montre l'arbre à parcourir d'un programme en exécution. Le point représente la trace d'exécution. ALx représente une alternative (un point de choix), COx la continuation. (b) est la pile d'évaluation avant la répartition. (c) donne les deux piles après répartition.

Le modèle suppose une mémoire commune contenant la définition du programme et certaines machines qui exécutent chacune Prolog séquentiellement. Quand il y a une machine libre, une machine en exécution d'un programme est arrêtée pour le lancement d'un processus de répartition. Le processus de répartition, selon la pile d'évaluation, coupe le programme en exécution en deux parties indépendantes. Une partie reste dans la machine initiale. L'autre partie, avec toutes les informations nécessaires à la vérification, est transférée dans la machine libre. Après ceci, deux machines commencent leurs exécutions indépendamment en accédant à la mémoire commune.

II.3 Architecture

L'étude de l'architecture de la machine Prolog consiste en la définition et en la réalisation des processeurs ou des machines supportant un langage Prolog parallèle ou un ou plusieurs modèles d'interprétation parallèle. Les principales recherches peuvent être regroupées dans deux catégories: (1) la machine basée sur l'idée de machine abstraite Prolog de Warren [War77]; et (2) La machine basée sur des modèles de processus. Ce sont deux approches complémentaires.

II.3.1 La machine abstraite Prolog et sa réalisation

La machine abstraite Prolog a été initialement spécifiée par Warren [War77]. La machine est constituée par des registres et des piles pour la gestion de résolution, et un opérateur faisant l'unification. Un programme Prolog est compilé en une séquence d'instructions relativement simples. La Fig.II.10 monte une structure générale des instructions correspondant à une clause.

Suivant le modèle de la machine abstraite, deux machines au moins sont mises en oeuvre matériellement. L'une est la machine HPM (High speed Prolog Machine) [NKH85], le parallélisme utilisé dans cette machine est le "pipelining" de la phase de recherche d'instruction, du décodage et de l'exécution. L'autre est la machine PLM1 [DDP85] développée dans le cadre du projet Aquarius [DeP85]. Cette machine a pris en compte le ET-parallélisme en se basant sur une analyse statique des programmes Prolog.

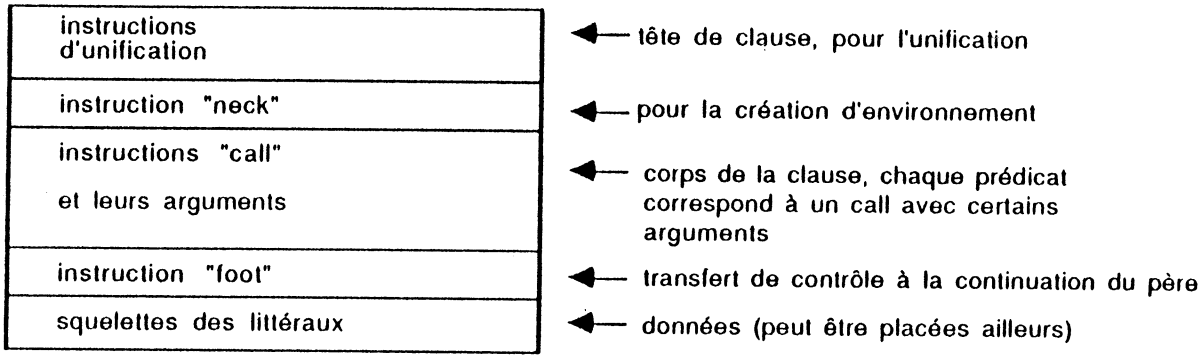

Fig.II.10

L'architecture de HPM est donnée ci-dessous dans la figure II.11.

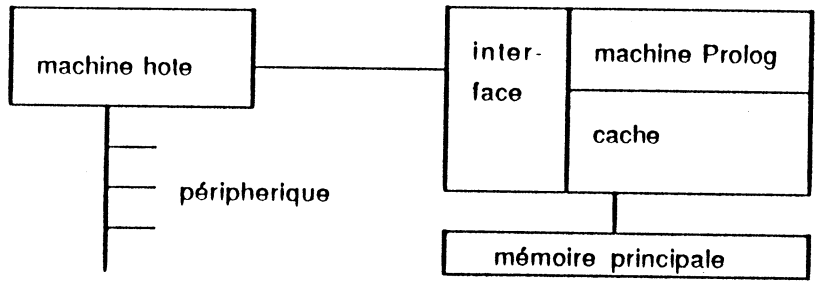

Fig.II.11

II.3.2 La machine PIM-R

La machine PIM-R (Parallel Inference Machine based on Reduction) repose sur le principe de réduction [OAS85]. En effet, la vérification d'un but peut être vue comme un processus de réduction du but par un groupe de clauses unifiant le but; l'effet de la réduction est le remplacement du but par des nouveaux buts spécifiés dans la queue d'une clause. La terminaison de l'exécution a lieu quand le but est réduit à vide.

L'architecture de la machine est organisée autour des modules d'inférence (IM) et des modules de mémoire structurée (SMM). Différents modules sont reliés par deux réseaux de communication (Fig II.12). Les processus de réduction sont alloués dynamiquement dans IM et sont en mesure d'accéder à SMM où sont stockées les données. Cette machine supporte le OU-parallélisme de Prolog et le ET-parallélisme de Prolog Concurrent.

II.4 Conclusion

L'expérience du parallélisme montre que le gain obtenu atteint difficilement deux ordres de grandeur à cause de l'overhead de synchronisation [Lus84]. Le développement du parallélisme pour Prolog est certainement une recherche encore à poursuivre. Les langages Prolog parallèles ne

sont pas encore mis en oeuvre. Leurs implémentations sur des machines parallèles influeront certainement sur la définition du langage. La manipulation du parallélisme au niveau de l'interprétation semble une approche plus intéressante actuellement, car un mécanisme élémentaire d'évaluation améliorera essentiellement la performance de Prolog.

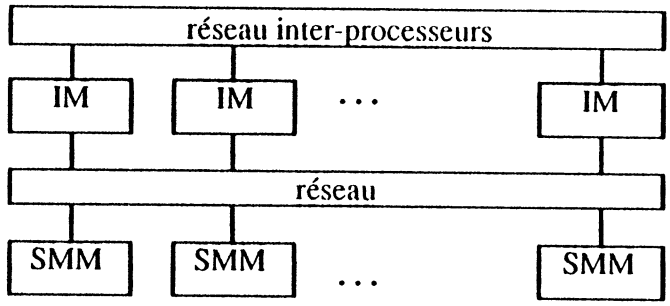

Fig.II.12

Il est à noter que, au niveau de la recherche sur l'architecture parallèle de la machine, la définition de matériels spécialisés doit reposer sur l'avancement du langage Prolog parallèle et du modèle d'interprétation parallèle. Nous croyons que la bonne approche vers l'architecture adaptée a Prolog est d'expérimenter d'abord sur des machines paralléles générales. C'est dans cette philosophie que le concept "architecture orientée processus" est proposé dans la suite de cette thèse.

CHAPITRE III

LE MODELE PARALLELE D'OPALE

CHAPITRE III

LE MODELE PARALLELE D'OPALE

Le modèle d'interprétation de Prolog dans OPALE est développé dans le contexte base de connaissances ; sa particularité est due au stockage des informations en mémoire secondaire. Dans ce chapitre, les spécificités du contexte sont d'abord dégagées. Ensuite les algorithmes de base, qui reposent sur la notion de processus, sont spécifiés. Les algorithmes pour le traitement des clauses récursives sont donnés comme cas particuliers des algorithmes de base. Enfin quelques remarques sur le modèle sont présentées.

III.1 Le contexte

Une base de connaissances Prolog présente des différences importantes par rapport à un système Prolog:

- Dans la programmation Prolog, l'ordre des clauses ayant même tête et même nombre d'arguments est important car l'interpréteur vérifie un but en respectant l'ordre d'apparition des clauses candidates. Les opérateurs de contrôle, par exemple la coupure "/", modifient aussi l'espace de recherche en suivant cet ordre. Dans un système de base de connaissances, les règles sont utilisées essentiellement pour la définition des informations implicites. Leur rôle est de guider la dérivation. Donc, l'ordre d'apparition des clauses est considéré non significatif.

- La modification de programme est une opération avec effet de bord; cette opération est par sa

nature très différente des opérations de recherche de données et de résolution de problèmes. Donc un modèle d'interprétation ne considère pas les effets de bord ; les modifications sur les connaissances sont prises en compte par un autre sous système.

- Les clauses unaires doivent avoir une cardinalité importante comme les relations de base dans une base de données relationnelle.

- Le taux d'échec de l'unification sera très élevé, car l'unification joue ici un rôle de sélection de données. Plus de 90% des données sont rejetées dans une base de connaissances, tandis qu'il n'y a en moyenne qu'environ 50% d'échecs de l'unification dans un programme Prolog typique.

- Les clauses seront moins complexes que dans le cas de programmes Prolog, en effet ces clauses ont une fonctionnalité différente d'un système de programmation.

- La récursivité est souvent utilisée en programmation en Prolog. L'utilisation de clauses récursives représente une amélioration nécessaire aux bases de connaissances par rapport aux bases de données classiques. Le problème se trouve au niveau de l'implémentation, car l'inefficacité de traitement des clauses récursives est inacceptable. Il est donc nécessaire de développer des algorithmes ad-hoc performants.

Les différences citées ci-dessus déterminent des caractéristiques d'OPALE qui constituent le contexte dans lequel le modèle parallèle d'OPALE doit être développé. Les principaux problèmes sont le stockage des clauses en mémoire secondaire et l'utilisation du filtre. Ces deux problèmes sont abordés ci-après.

III.1.1 Système de stockage

Le grand volume de données exige l'utilisation de la mémoire secondaire pour stocker les clauses. Le disque est considéré comme le support physique le plus adapté actuellement. Le système

de stockage est un niveau inférieur dans OPALE, il accepte les commandes issues du modèle d'interprétation, les exécute et retourne les données que le modèle demande. La figure III.1 exprime le fonctionnement du système.

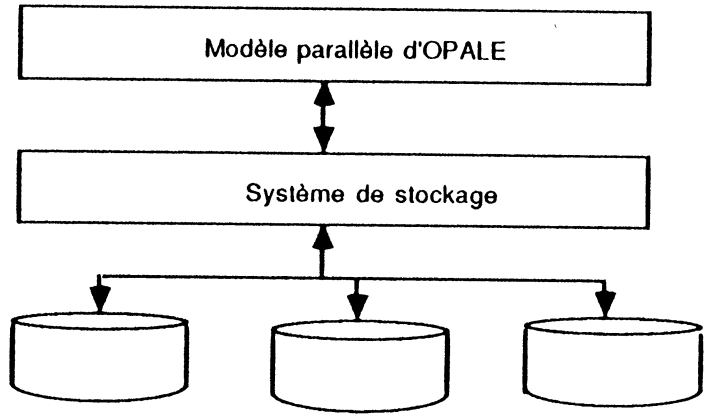

Fig.III.1

Le système de stockage fournit les fonctionnalités suivantes:

- Organisation et distribution des clauses sur les disques.
- Indexation.
- Modifications de la base: insertion, suppression, modification.
- Filtrage, qui est le sujet ci-dessous.

Dans beaucoup de bases de données déductives, le stockage des clauses est divisé en deux parties: les données élémentaires dans la "base de données extensionnelle", et les informations générales (règles) dans la "base de données intensionnelle". Cette séparation permet d'installer le système autour d'une base de données relationnelle et d'appliquer une méthode de compilation de requêtes en profitant des mécanismes existants d'optimisation, de contrôle de concurrence, etc. Dans OPALE, au niveau du stockage, les clauses unaires et les règles ne sont pas distinguées. Car d'une part, nous croyons qu'un ensemble de clauses définit une entité complète au niveau sémantique, ce qui facilite l'organisation du stockage et la modification des connaissances. D'autre part, une méthode d'interprétation est adoptée, qui fournit la possibilité de stocker ses clauses ensemble. Nous pensons que les inconvénients d'interprétation seront compensés par la souplesse,

la possibilité d'appliquer des fonctions heuristiques etc.

L'accès aux disques se fait par blocs séquentiels et demande un temps relativement long (de l'ordre de 20 à 30 ms), ce qui nécessite une méthode d'accès sans retour arrière. Dans ce cas là, l'optimisation des accès aux disques, en nombre et en durée, va être un critère de performance déterminant. Donc, plutôt que de chercher les solutions les unes après les autres comme dans Prolog, un maximum d'informations doit être extrait pour chaque accès.

Le stockage des clauses en mémoire secondaire nécessite qu'une approche orientée traitement ensembliste soit utilisée. L'approche orientée traitement ensembliste explore les solutions en respectant une stratégie de recherche en largeur d'abord (breadth first). Une méthode évitant la redéfinition de la stratégie de recherche de Prolog est de temporiser les solutions extraites d'un accès au disque, et de laisser l'interpréteur les traiter les unes après les autres. Dans un contexte base de connaissances, la redéfinition de la stratégie de recherche est nécessaire pour que le système puisse se baser sur le traitement ensembliste.

III.1.2 Filtrage

Le filtrage est une technique utilisée dans les machines base de données. Il consiste à insérer un opérateur matériel du type automate fini entre le disque et la mémoire centrale, comme moyen de pré-selectionner les données. Il est à noter que le filtrage se fait dans un temps compatible avec la vitesse du disque. L'utilisation du filtre permet de minimiser le nombre d'informations inutiles amenées dans la mémoire centrale, ce qui réduit non seulement l'espace de travail mais aussi le temps de traitement. Dans le projet OPALE, un opérateur de filtrage a été développé [Bel84] [Ian85]. Cet opérateur réalise une partie importante de l'unification, ce qui décharge beaucoup les traitements en aval. Le principe est brièvement décrit ci-dessous.

A. L'algorithme d'unification: Dans un contexte base de connaissances Prolog, l'unification joue un rôle de sélection des données, les substitutions générées par l'unification sont considérées

comme des données sélectionnées. L'unification est une opération assez complexe. Il est impossible de réaliser cette opération pendant le temps de transfert des données lues à partir d'un disque, c'est à dire que le temps de lecture des données (qui dépend de la vitesse du disque) et le temps de traitement de ces données (sélection par unification) ne sont pas compatibles, ce dernier étant non linéaire, et souvent bien plus long que celui du premier.

Pour éviter un transfert de toutes les données lues sur disque dans la mémoire primaire pour réaliser le traitement (unification), on décompose l'unification en deux sous opérations, la pré-unification et l'association. Nous donnons une explication informelle ci-après: la pré-unification compare deux termes en substituant les variables apparaissant dans les termes, sans vérifier la cohérence des substitutions; l'association vérifie la cohérence des substitutions générées par la pré-unification et complète l'unification. La pré-unification réalise une pré-sélection des données par un opérateur se situant entre les disques et la mémoire centrale. Les données qui ne satisfont pas la pré-unification sont rejetées. Ceci diminue énormement les transferts inutiles.

Exemple III.1:

Soient les termes terme(X,X) et terme(a,b). La pré-unification réussit et génère X<-a et X<-b, l'association vérifie que ces substitutions ne sont pas cohérentes car a et b ne sont pas unifiables, donc l'unification échoue.

Deux termes a(X,X) et a(b(Z,c),b(d,Y)), la pré-unification génère simplement X<-b(Z,c) et X<-b(d,Y), l'association complète l'unification en générant Z<- d et Y<- c.

La pré-unification est une opération assez simple, telle qu'il est possible de la réaliser par un opérateur câblé, dans un temps compatible avec la vitesse du disque.

B. Filtre: Le filtre est un opérateur réalisant la pré-unification entre un ensemble de buts et un flux de données issu du disque représentant les clauses élémentaires ayant le même nom de littéral.

Le filtre reçoit un ensemble de buts compilés, déclenche la lecture du disque et produit les substitutions. Nous donnons une explication avec l'exemple ci-dessous. Une présentation formelle peut être trouvée dans [Bel84] [Ian85].

Exemple III.2:

Supposons que les buts à vérifier soient les suivants:

but 1: terme(a,X0)

but 2: terme(X1,m(1))

but 3: terme(b,n(i,X2))

Ces buts seront compilés dans un automate montré en Fig.III.2, (a) représente une structure englobante de tous les termes, à chaque noeud est liée une liste de valeurs qui se trouvent dans cette position, à chaque valeur est lié un vecteur (b) qui désigne l'apparition de la valeur dans les buts. Quand un flux de données arrive, le filtre parcourt l'automate, et s'il arrive jusqu'au bout, la pré-unification est réussie.

Supposons que les clauses unaires lues du disque soient: terme(a,m(1))<-; terme(a,n(i,q))<-....

On commence par l'état 1, "terme" est bien trouvé, on charge le vecteur de travail avec 111 qui indique que les buts 1,2 et 3 sont vérifiés à ce moment là, et on avance en direction de "fils"; le deuxième élément a est trouvé dans l'état 2, on fait un "et" logique entre le vecteur de travail et le vecteur lu actuellement, le vecteur de travail devient 110. Si on continue cette démarche pour les données suivantes, pour la première clause lue du disque, les buts 1 et 2 sont vérifiés en obtenant des substitutions but1: X0<-m(1) et but2: X1<-a.

III.2 Les algorithmes de base

III.2.1 Généralités

Dans OPALE, il n'y a pas d'opérateur parallèle au niveau du langage. Le parallélisme est déterminé automatiquement par le système. Nous croyons que la spécification de parallélisme au niveau utiliseur n'est pas une bonne approche pour une base de connaissances. Par contre, la détection automatique du parallélisme présente des avantages de souplese, de modifiabilité, d'extensibilité etc.

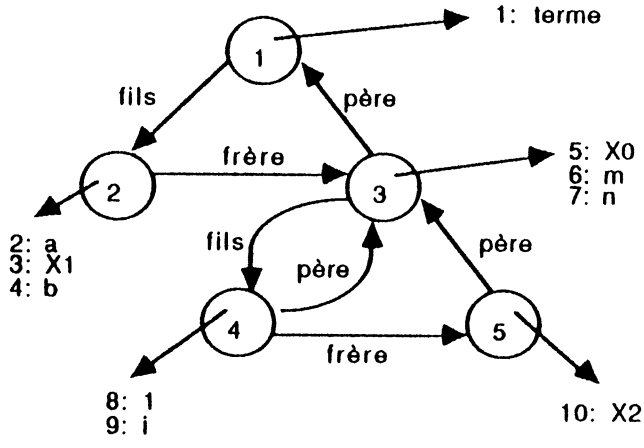

Fig.III.2

Avant une explication sur le modèle, les définitions suivantes sont données:

- Paquet de clauses: un ensemble de clauses ayant même tête et même nombre d'arguments.
- Substitution: un couple variable/valeur, la valeur étant liée à cette variable.
- Solution: un ensemble de substitutions générées pour un but.
- Environnement: un ensemble de substitutions générées pour une clause.

Le modèle est constitué par des processus. Une vérification est modélisée par un réseau de processus présenté dans la Fig.III.3.

- ET-processus: Ils sont attachés à chaque littéral dans la queue d'une clause et organisés en pipe-line. Un ET-processus reçoit les environnements produits par son prédécesseur, et applique des substitutions aux variables pour générer les buts à vérifier. Le ET-processus crée des R-processus (processus de recherche) par l'intermédiaire desquels les buts sont vérifiés.

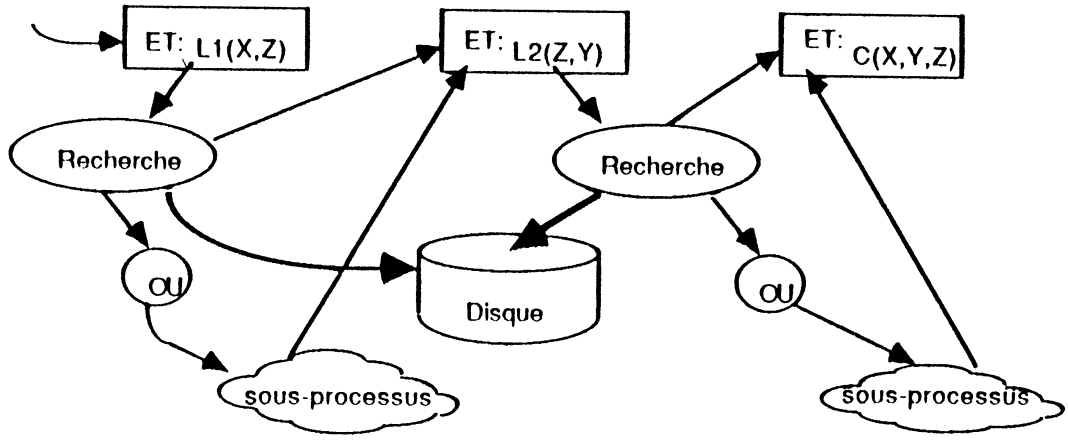

Fig.III.3

- R-processus: Un R-processus reçoit un groupe de buts à vérifier. Il doit d'abord accéder au dictionnaire du système pour avoir l'adresse disque nécessaire à la recherche. Ensuite, après avoir envoyé les commandes de filtrage aux processeurs disques correspondants, il attend les réponses qui contiennent les substitutions générées au cours de l'unification. Si une clause non unaire vérifie un but, un OU-processus sera créé. Dans le cas de clause unaire, les substitutions sont envoyées directement vers le successeur du ET-processus.

- OU-processus: Ils créent un ensemble de ET-processus correspondant chacun à un prédicat dans la queue de la clause. Le nom du ET-processus successeur est envoyé à ses ET-processus fils pour qu'ils puissent transférer directement les résultats au ET-processus successeur.

III.2.2 Les algorithmes

L'agorithme de base traite les cas généraux. Quand une clause récursive est rencontrée, l'algorithme spécifié dans la prochaine section sera appelé. Les algorithmes sont spécifiés en pseudo

Pascal. Certaines primitives et fonctions sont d'abord définies ci-dessous:

- recevoir(<données>): recevoir des données issues d'un processus, le processus émetteur est ignoré.

- envoyer(<processus_id>,<données>): envoyer des données à un processus qui porte un identificateur <processus_id>.

- créer(processus): créer un processus_fils et lui transférer des paramètres.

- attendre: attendre la terminaison de ses processus_fils.

- filtre(<buts>,<adresse_disque>): créer un processus d'accès disque qui effectue le filtrage sur le flux de données issu du disque. Les résultats seront retournés par envoi de messages.

Trois fonctions sont définies:

- adresse_disque = index(<but>): en accédant au mécanisme d'indexation de la base de connaissances, les adresses disque concernant la vérification des buts sont retournées.

- buts = appliquer(<squelette>,<environnement>): appliquer un ou plusieurs environnements à un squelette d'un littéral pour construire des buts à vérifier.

- "U" : l'opération U représente l'évaluation d'un ensemble d'environnements avec un autre pour vérifier leur cohérence. Elle constitue de nouveaux environnements et correspond souvent à la jointure de l'algèbre relationelle.

L'algorithme III.1: ET-processus

```
ET-processus(Squelette)
BEGIN
  recevoir(Suc_processus);     /* nom du successeur ET-processus */
  recevoir(Env);    /* premier environnement */
  WHILE (Env != End_env)
   BEGIN
     Buts=appliquer(Env,Squelette);
     créer(R-processus(Buts,Suc_processus,Env));
     recevoir(Env);
```

```
    END;
  attendre;
  envoyer(Suc_processus,End_env);
END.
```

L'algorithme III.2: R-processus

```
R-processus(Buts,Suc_processus,Env)
BEGIN
  filtre(Buts,index(Buts));
  recevoir(Solutions);
  WHILE (Solutions != End_sol)
    BEGIN
      New_env = Env U Solutions;
      IF (Queue=Vide) THEN envoyer(Suc_processus,New_env)
      ELSE créer(OU-processus(Suc_processus,New_env,Queue));
      recevoir(Solutions);
    END;
END.
```

L'algorithme III.3: OU-processus

```
OU-processus(Suc_processus,Env,Queue)
BEGIN
  Nb=nombre de littéraux dans la Queue;
  FOR i=1 TO Nb DO
      Num_proc[i]=créer(ET-processus(Queue[i]));
  FOR i=1 TO Nb-1 DO
      envoyer(Num_proc[i],Num_proc[i+1]);
  envoyer(Num_proc[Nb],Suc_processus);
  envoyer(Num_proc[1],Env);
```

envoyer(Num_proc[1],End_env);

attendre;

END.

La requête utilisateur invoque un processus initial:

OU-processus(Utilisateur,End_env,Requête)

Ce processus permet de lancer la vérification. L'Utilisateur est le nom du processus d'utilisateur qui a lancé la requête.

III.2.3 Un exemple

Un simple exemple de relation généalogique ci-dessous montre le fonctionnement du modèle spécifié.

Exemple III.3

grandpère(X,Y) :- père(X,Z), parent(Z,Y).

parent(X,Y) :- père(X,Y).

parent(X,Y) :- mère(X,Y).

père(jean,pierre). mère(denise,alain).

père(jean,anne). mère(jeanne,christine).

père(laurent,nicole). mère(anne,nicole).

père(pierre,julie). mère(christine,pierre).

individu(jean,50,masculin).

individu(julie,8,féminin).

individu(nicole,14,féminin).

:::

Le réseau de processus pour l'évaluation de la requête

grandpère(jean,X),individu(X,Age,_),plusgrand(Age,7)

est représenté dans la figure III.4.. L'évaluation du prédicat "plusgrand" n'invoque pas de recherche sur le disque car c'est un prédicat évaluable. Dans la figure, pour les ET-processus et R-processus, les littéraux et les buts sont donnés respectivement; tandis que pour les OU-processus, les clauses correspondantes ne sont pas dessinées.

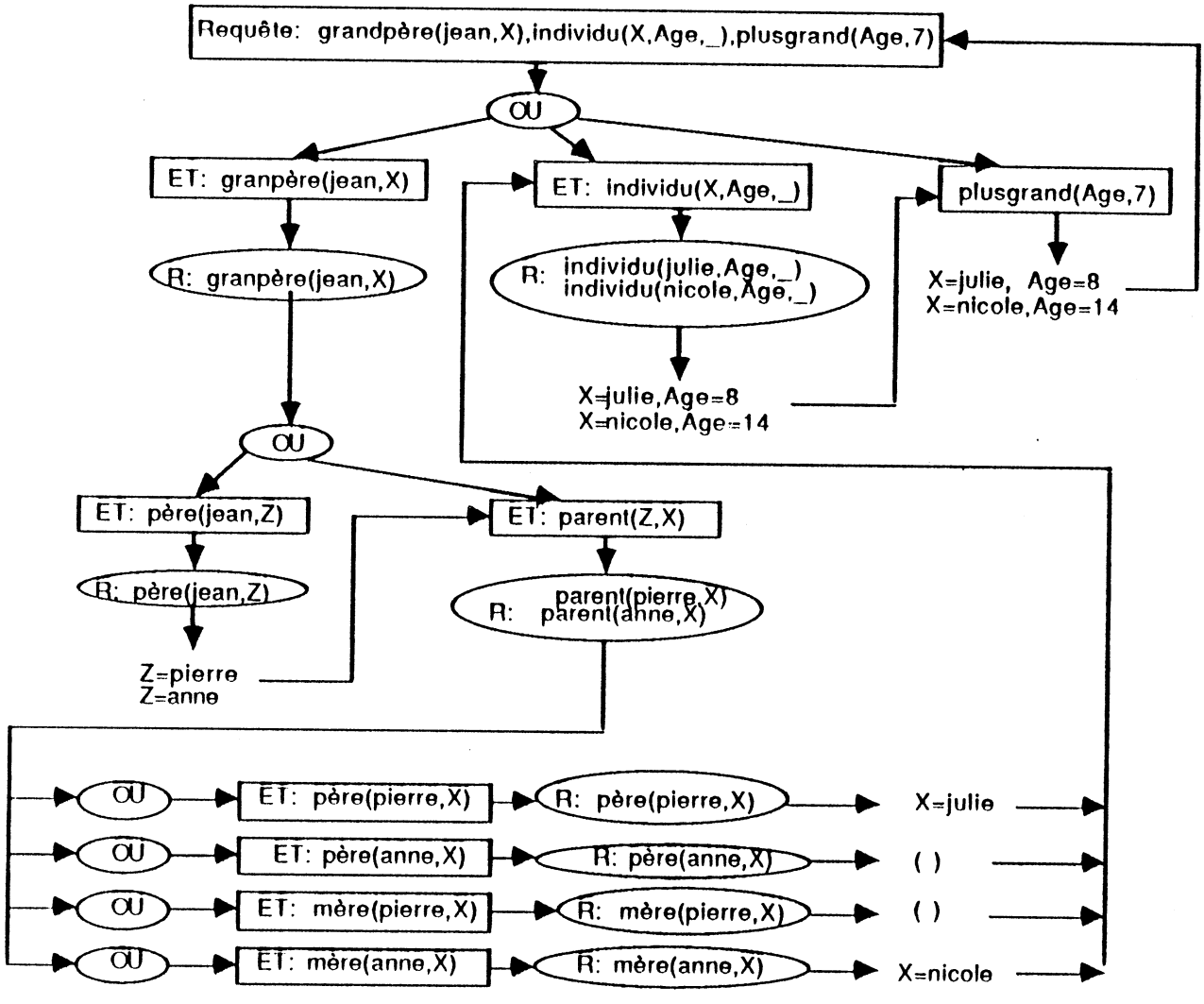

Fig. III.4

III.2.4 Optimisation de l'algorithme de base

L'algorithme de base peut être optimisé facilement en utilisant la méthode de

ré-ordonnancement des littéraux dans la queue d'une clause [War81]. Cette méthode consiste à évaluer d'abord des littéraux qui produisent le moins de solutions. Ceci correspond à des techniques classiques utilisées dans le traitement de requêtes dans les bases de données relationnelles, qui effectuent d'abord des projections et des sélections. Dans le modèle présenté à la section précédente, le OU-processus peut réaliser cette optimisation.

La deuxième optimisation envisageable est de diminuer les transferts inutiles entre les processus.

Exemple III.4:

La clause suivante met en évidence différents types de variables:

```
r(X,Y,Z) <- r1(V,U,Z)
            r2(X,V,Z)
            r3(X,Y);
```

La variable U n'apparaît que dans r1, elle est inutile pour l'évaluation de ses successeurs r2 et r3. La variable V n'apparaît que dans r1 et r2, elle n'a pas d'effet pour l'évaluation de r3.

Pour éviter des accès et des transmissions inutiles entre processus, nous utilisons un descripteur pour chaque processus permettant d'indiquer les types de variables. Les variables dans la queue d'une clause sont divisées en 6 types:

- V.P. (variable principale): variable libre qui sera utilisée dans la suite de l'évaluation. Ce processus est le producteur des substitutions de la variable, ces substitutions doivent être transférées pour son successeur.

- V.S. (variable secondaire): variable libre qui ne sera pas utilisée dans la suite de l'évaluation, il est inutile de générer les substitutions de la variable. Les substitutions sont jetées par le module association.

- V.I. (variable immédiate): variable liée, sa valeur se trouve dans le descripteur. Ce processus applique la valeur au squelette du prédicat pour construire les buts à vérifier. Pendant le filtrage des données, cette variable est équivalente à une constante.

- V.F. (variable filtrée): variable liée par un filtrage de données dans un processus prédécesseur, elle sera utilisée dans la suite de l'évaluation. Ce processus est le consommateur des substitutions générées par un de ses prédécesseurs. Les substitutions doivent être appliquées au squelette du prédicat pour construire les buts à vérifier. Les substitutions avec lesquelles les buts sont construits et vérifiés doivent être transférées à son successeur.

- V.T. (variable temporaire): variable liée par un filtrage de données dans un processus prédécesseur. Elle ne sera pas utilisée dans la suite de l'évaluation. Comme le type précédent, mais les substitutions seront jetées après la construction des buts.

- V.C. (variable à copier): variable qui n'apparaît pas dans ce prédicat, elle est liée par un processus prédécesseur et les substitutions doivent être recopiées pour son successeur.

En fonction de la description des variables dans un prédicat, un processus sait comment construire des buts (avec des variables des types V.I, V.T et V.F), quelles sont les variables pour lesquelles les substitutions doivent être générées (du type V.P), et quelles sont les variables dont les substitutions doivent être transférées à son successeur (des types V.P, V.F, V.C). Les substitutions pour une variable du type V.S ne seront pas générées par le filtre, les substitutions pour une variable du type V.T seront rejetées (ne pas transmettre pour son successeur). La génération de substitutions et la transmission inutile dans chaque étape de traitement sont évitées grâce au descripteur de variables.

Dans l'exemple III.4, pour la requête "r(terme,X1,X2)", les descripteurs pour chaque prédicat seront ceux présentés dans la Fig.III.5 :

III.3 Les algorithmes pour des clauses récursives

III.3.1 Les clauses récursives

Une clause est dite récursive si son évaluation invoque elle-même directement ou par intermédiare d'une autre clause. Dans l'exemple I.1, la clause

aller(X,Y):- route(X,Z),aller(Z,Y).

est un exemple typique de clause récursive. Les clauses récursives sont utilisées très souvent dans un système de résolution de problèmes. Les langages Prolog, Lisp et certaines bases de données déductives supportent ce type de clauses.

Prédicat	V	U	X	Y	Z
r1	V.P	V.S	—	—	V.P
r2	V.T	—	V.I	—	V.F
r3	—	—	V.I	V.P	V.C

Fig.III.5

Le traitement de clauses récursives pose des problèmes dans le contexte des bases de données déductives, ou bases de connaissances à grand volume de données [RoL85,Mai85]. Pour la méthode de compilation, les requêtes doivent être compilées en des expressions contenant seulement les relations de base reliées par des opérateurs jointure, donc il faut trouver une formule générale et l'algorithme d'évaluation approprié pour traiter la récursivité. Malheureusement, il n'est pas toujours possible de transformer une requête concernant la récursivité en une formule générale. Donc la compilation des clauses récursives complexes est très coûteuse [Han85].

Quant à la méthode interprétative, le problème semble moins difficile. Mais si on applique directement les algorithmes III.1-3 aux clauses récursives, le nombre de processus va être très élevé. Il sera beaucoup plus raisonnable de développer un algorithme approprié pour ceci. Dans la suite de cette section, nous donnons d'abord quelque définitions; ensuite un algorithme de traitement et la démonstration de sa complétude et de sa terminaison sont présentés; et enfin, nous montrons comment l'intégrer dans les algorithmes de base du modèle parallèle d'OPALE.

Nous distinguons d'abord deux types de clauses récursives: les clauses récursives directes et les clauses récursives mutuelles. Les clauses récursives directes sont celles dont la queue contient

une seule ou plusieurs fois le prédicat de tête. Un exemple de clauses récursives directes est le suivant:

R(X,Y) :- A(X,V1),R(V1,V2),B(V2,V3),R(V3,V4),C(V4,Y).

Les clauses récursives mutuelles sont celles dont l'évaluation d'une clause doit appeler l'autre et vice versa. Un exemple de ce type de recursivité est présenté ci-après:

R(X,Y) :- S(X,Z),A(Z,Y).
S(X,Y) :- B(X,Z),R(Z,Y).

Certaines clauses récursives mutuelles peuvent être transformées en des clauses récursives directes [Han85]. Cette transformation intervient aussi dans les problèmes concernant l'assimilation de connaissances [MKK83] [Kow74b] et le processus de transformation est souvent compliqué. Dans cette thèse, nous discutons seulement le traitement des clauses récursives directes.

Dans la suite du chapitre, pour simplifier l'explication de l'algorithme, quand les arguments des prédicats sont absents, par exemple A,B et E, nous considérons que les variables du prédicat sont toutes libres et leur évaluation retournera toutes les solutions. Pour des cas où on veut un sous ensemble de solutions correspondant à une ou plusieurs variables du prédicat liées avec un groupe de termes, le symbole |R| est utilisé et nous l'appelons "la recherche sur R".

Nous considérons un paquet de clauses récursives de la forme générale suivante:

R :- E. (III-1.1)

R :- A_1,R,B_1,R,C_1... (III-1.2)

R :- A_2,R,B_2,R,C_2... (III-1.3)

:::

R :- A_n,R,B_n,R,C_n... (III-1.n+1)

où E,A_i,B_i,C_i... sont des clauses unaires; la clause (III-1.1) s'appelle la clause de sortie; dans la queue de chaque clause récursive, les littéraux ayant le même prédicat que la tête s'appellent les points récursifs; quand il n'existe qu'une clause récursive dans le paquet, la clause est dite récursive unique, s'il en existe plusieurs, les clauses sont dite clauses récursives multiples.

III.3.2 Les algorithmes

Nous commençons par un exemple avec une clause récursive unique à deux points récursifs:

R :- E.

R :- A,R,B,R,C.

Le traitement est détaillé par étape et représenté par la figure III.6.

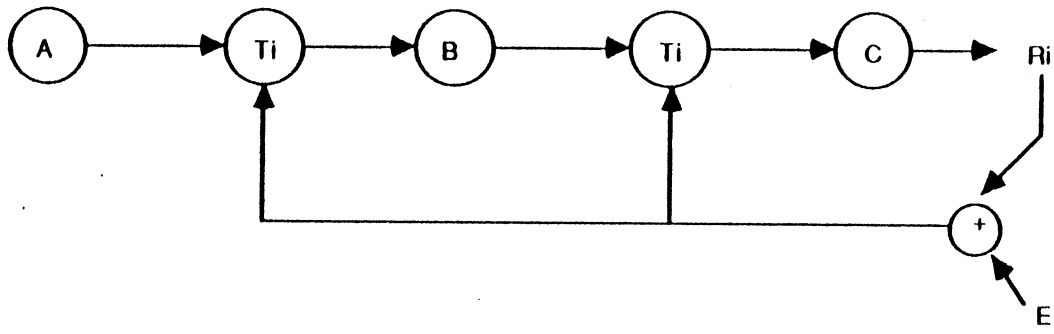

Fig.III.6

étape 1: effectuer la recherche sur E et on obtient le premier groupe de solutions R_0, transférer R_0 dans Résultat, Résultat<-R_0. Cette étape correspond à l'évaluation du but R par la clause de sortie.

étape 2: évaluer A,E,B,E,C; on obtient le premier groupe de solutions partielles R_1 en éliminant les solutions doubles. Cette étape correspond à la résolution du but R par la clause

récursive et les points récursifs générés sont résolus ensuite par la clause de sortie.

étape 3: concaténer R_1 et E, cette opération correspond à l'union de l'algèbre relationnelle, on l'exprime par T_1<- R_1+E; évaluer ensuite A,T_1,B,T_1,C; on obtient le deuxième groupe de solutions partielles R_2; éliminer les solutions doubles dans R_2.

étape 4: effectuer la différence de R_2 et R_1, c'est à dire D <- R_2-R_1; si D=vide alors terminer, sinon aller à l'étape 3 avec R_2 au lieu de R_1.

étape 5: effectuer la recheche sur R_n et concaténer les solutions $|R_n|$ et Résultat, c'est à dire Résultat <- Résultat+$|R_n|$, Résultat contient toutes les solutions.

Les étapes 3-4 sont répétées jusqu'à ce que R_{i+1} - R_i devienne vide, c'est à dire qu'il n'y ait plus de solutions nouvelles générées. Dans le cas ou il y a plus ou moins de points récursifs, les points récursifs sont toujours d'abord remplacés par la queue de la clause de sortie E pour la génération du premier groupe de solutions partielles R_1; et ensuite remplacés par R_i+E.

L'algorithme III.4: le traitement de la clause récursive unique R:-A,R,B,R,C.

1) structure de données:

(a) un entier i pour le nombre d'itérations.

(b) quatre tampons:

-Résultat pour les résultats finaux;

-R_t pour résultats partiels;

-R_i pour résultats partielles antérieurs;

-T_i pour la concaténation de E et R_t.

2) initialisation:

(a) effectuer la recherhe sur E, mettre les solutions obtenues dans Résultat,

Résultat <-|E|.

(b) i<-0.

(c) T_0<-E.

(d) évaluer la clause récursive en remplaçant les points récursifs par T_0,

R_1<-A,T_0,B,T_0,C; et éliminer les solutions doubles dans R_1.

3) itération:

(a) i<-i+1.

(b) concaténer les solutions partielles, T_i <- R_i+E.

(c) évaluer la clause récursive en remplaçant les points récursifs par T_i, R_t<-A,T_i,B,T_i,C;

et éliminer les solutions doubles dans R_t.

(d) effectuer la différence R_t-R_i, si R_t-R_i=vide alors aller en 4), sinon R_i <- R_t et aller en 3.(a).

4) final:

(a) effectuer la recherche sur R_i et concaténer les résultats, Résultat <- $|R_i|$ + Résultat .

(b) fin.

L'algorithme présenté ci-dessus est équivalent à la résolution par réfutation, telle qu'elle est utilisée dans le langage Prolog. Donc il est complète, c'est à dire qu'il retourne toutes les solutions possibles; et il termine dans toutes les cas.

Théorème III.1: L'algorithme III.4 est complet et terminable.

Démonstration:

Complétude: L'algorithme III.4 peut être modélisé par la fonction suivante:

$$R_0 = \{\ \}$$

$$R_i = F(x,y \mid x,y \in \{\ E + R_{i-1}\ \}) = \Lambda,x,B,y,C. \qquad (1<=i).$$

Avec cette fonction, on peut calculer R_i de chaque étape. Le résultat final est obtenu par

$$\text{Résultat} = |E| + |\lim_{i->\infty} R_i|$$

En appliquant cette fonction, nous pouvons obtenir les résultats partiels dans différentes étapes:

$R_1 = F(x,y \mid x,y \in \{\ E\ \})$

$= \{\ \Lambda EBEC\ \}$

$R_2 = F(x,y \mid x,y \in \{\ E,\Lambda EBEC\ \})$

$= \{\ \Lambda EBEC, \Lambda EB\ \Lambda EBEC\ C, \Lambda\ \Lambda EBEC\ BEC, \Lambda\ \Lambda EBEC\ B\ \Lambda EBEC\ C\ \}$

$R_3 = F(x,y \mid x,y \in \{\ E, \Lambda EBEC, \Lambda EB\Lambda EBECBEC, \Lambda\Lambda EBECBEC, \Lambda\Lambda EBECB\Lambda EBECC\ \})$

$= \{\ \Lambda EBEC, \Lambda EB\ \Lambda EBEC\ C, ..., \Lambda EB\ \Lambda\Lambda EBECB\Lambda EBECC\ C,$

:::

$\Lambda\ \Lambda\Lambda EBECB\Lambda EBECC\ BEC, ..., \Lambda\ \Lambda\Lambda EBECB\Lambda EBECC\ B\ \Lambda\Lambda EBECB\Lambda EBECC\ C\ \}$

Si on définit la profondeur P de résolution par l'application au maximum P fois de la clause récursive pour un point récursif et les points récursifs générés par la résolution, on trouve que le résultat R_i retourné par la fonction F correspond à toutes les solutions obtenues en profondeur i par la résolution par réfutation. Les résultats finaux correspondent à une sélection sur toutes les

solutions. Puisque la résolution par réfutation est complète, l'algorithme III.4 est complet aussi.

Terminaison: La condition de terminaison est suffisante. Quand R_{i+1} est équivalent à R_i, l'application de l'algorithme ne génère aucune nouvelle solution, le calcul à ce moment là atteint l'état stable.

La condition de terminaison est nécessaire. Puisque Ri est monotone, c'est à dire que

$$R_i \subseteq R_{i+1}$$

Si R_{i+1} n'est pas équivalent R_i, ça veut dire qu'il y a encore des nouvelles solutions, donc il est possible que l'application de l'algorithme génère de nouvelles solutions.

Une modification simple de l'algorithme III.4 s'applique au cas de clauses récursives multiples. Le principe est décrit dans la figure III.7.

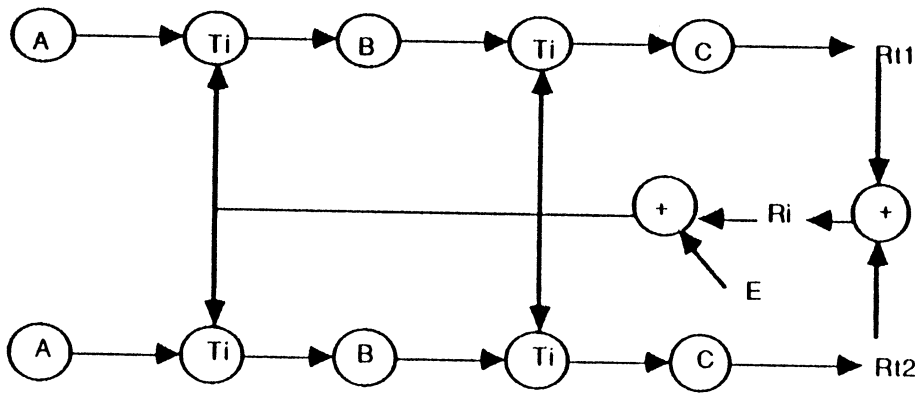

Fig. III.7

L'algorithme III.5: Le traitement de n clauses récursives multiples.

Nous considérons les clauses (III-1.1) - (III-1.n+1).

1) structure de données:

(a) un entier i pour le nombre d'itérations.

(b) n+4 tampons:

- Résultat pour les résultats finaux.

- R_{t1} à R_{tn} pour les résultats partiels de chaque clause.

- R_t pour l'ensemble des résultats partiels.

- R_i pour l'ensemble des résultats partiels antérieurs.

- T_i pour la concaténation de R_t et E.

2)initialisation:

(a),(b) et (c) comme dans l'algorithme III.4.

(d) évaluer R_{tj} <- $A_j,T_0,B_j,T_0,C_j,\ldots$, (1<=j<=n) pour chaque clause récursive et concaténer de R_{t1} à R_{tn} dans R_t, R_t<- $R_{t1}+R_{t2}+\ldots+R_{tn}$; éliminer les solutions doubles dans R_t et R_1<-R_t.

3) itération:

(a) i <- i+1.

(b) effectuer la concaténation T_i<- R_t+E.

(c) évaluer R_{tj}<- $A_j,T_i,B_j,T_i,C_j,\ldots$, (1<=j<=n).

(d) concaténer R_{tj}, (1<=j<=n), dans R_t, R_t<-$R_{t1}+R_{t2}+\ldots+R_{tn}$; éliminer les solutions doubles dans R_t; effectuer R_t-R_i, si R_t-R_i=vide alors aller en 4), sinon R_i<- R_t et aller en 3.(a).

4) final:

(a) effectuer la recherche sur R_i et concaténer les résultats,

Résultats <- $|R_i|$ + Résultats.

(b) fin.

III3.3 L'optimisation

Les algorithmes III.4 et III.5 peuvent être optimisés pour réduire les évaluations repétitives. Pour des clauses récursives qui ont une forme plus simple que celle de (III-1.1) à (III-1.n+1), l'évaluation peut être optimisée en évitant la génération de toutes les solutions, nous verrons un exemple d'une clause récursive unique avec un seul point récursif.

Nous discuterons l'optimisation sur l'algorithme III.4. Le même principe s'applique à l'algorithme III.5. Dans l'étape i, $T_i=R_i+E$, on peut aussi dire que $T_i=T_{i-1}+D$ où $D=R_i-R_{i-1}$. Donc l'évaluation de A,T_i,B,T_i,C peut être reécrite comme

$$A,\{T_{i-1}+D\},B,\{T_{i-1}+D\},C \qquad \text{(III-2)}$$

L'évaluation de (III-2) est équivalent à des calculs relationnels, la concaténation "+" correspond à l'union, l'évaluation successive de A,B correspond à la jointure. Puisque la jointure est distributive par rapport à l'union, (III-2) peut être décomposée en quatre formules:

$$A,T_{i-1},B,T_{i-1},C \qquad \text{(III-3.1)}$$

$$A,T_{i-1},B,D,C \qquad \text{(III-3.2)}$$

$$A,D,B,T_{i-1},C \qquad \text{(III-3.3)}$$

$$A,D,B,D,C \qquad \text{(III-3.4)}$$

Parmi ces formules, (III-3.1) est identique à R_{i-1} qui est déjà évaluée; la partie de (III-3.2), A,T_{i-1},B est aussi évaluée dans la dernière boucle. Donc pour éviter de refaire la même évaluation dans chaque boucle, seulement D, c'est à dire les nouvelles solutions partielles générées dans la boucle actuelle, est transféré à chaque point récursif. Le premier point récursif jette T_i dans chaque boucle, il n'évalue que A,D,B. Le deuxième point récursif garde T_i pour évaluer (III-3.2) dans la

prochaine boucle. A la fin d'une boucle, il concatène T_i et D pour construire T_{i+1}.

Il est à noter que dans le cas de la clause à un seul point récursif, l'évaluation faite dans chaque boucle utilise seulement les solutions partielles dans D qui ne sont pas encore essayées, ceci diminue énormément les opérations inutiles. L'algorithme optimisé est donné dans la prochaine section.

Deuxièmement, nous discutons l'optimisation du type de clauses récursives qui est le plus utilisé. Les exemples suivants représentent ce type de clauses.

ancêtre(X,Y) :- père(X,Y).
ancêtre(X,Y) :- père(X,Z), ancêtre(Z,Y).

aller(X,Y) :- route(X,Y).
aller(X,Y) :- route(X,Z), aller(Z,Y).

Nous prenons la forme générale suivante:

R(X,Y) :- E(X,Y). (III-4.1)
R(X,Y) :- E(X,Z),R(Z,Y). (III-4.2)

L'évaluation de ce type de clauses récursives correspond au calcul de la fermeture transitive de la relation E. Les solutions possibles peuvent être exprimées par les formules ci-dessous:

E(X,Y).
E(X,X1),E(X1,Y).
E(X,X1),E(X1,X2),E(X2,Y).
E(X,X1),E(X1,X2),E(X2,X3),E(X3,Y).
:::

S'il y a une variable, soit X, étant liée, on peut évaluer ces formules de gauche à droite en substituant X avec ce qui est lié. Si c'est la variable Y qui est liée, on peut les évaluer de droite à gauche de même façon. Donc on peut déterminer que la concaténation des solutions se fait toujours par la variable libre. Ceci permet d'éviter la génération de toutes les solutions possibles. L'algorithme III.6 ci-dessous est issu de [Han85].

L'algorithme III.6: le traitememt de la clause R:-E,R.

1) structure de données:

Trois tampons: Résultat, T et D.

2) initialisation: Effectuer la recherche sur E et mettre les résultats dans Résultat et D,

Résultat <- |E|, D<- |E|.

3) itération:

(a) effectuer l'évaluation E,D; mettre les résultats dans T, c'est à dire T <- E,D.

(b) sélectionner les solutions dans T qui n'appartiennent pas à Résultat, mettre les résultats sélectionnés dans D.

(c) si D=vide alors aller en 4), sinon ajouter D dans Résultat, Résultat<-Résultat +D, et aller en 3.(a).

4) fin.

III.3.4 L'intégration aux algorithmes de base

Les algorithmes présentés ci-dessus peuvent être intégrés naturellemment dans le modèle utilisé avec l'algorithme de base III.1-III.3. Nous considérons que les R-processus, avant de créer les OU-processus pour une clause qui unifie un but, détectent s'il s'agit d'une clause récursive. Puisque nous considérons seulement les clauses récursives directes, la détection de clauses récursives devient un simple examen des prédicats dans la queue de la clause. L'algorithme du R-processus avec la détermination des clauses récursives utilise une nouvelle fonction prédicat(), cette fonction teste son argument et retourne un couple nom du prédicat et nombre des arguments:

- (Nom_prédicat,Nb_arg) = prédicat(terme).

L'algorithme III.7: R-processus avec la détection de clauses récursives.

```
R-processus(Buts,Suc_processus,Env)
BEGIN
  filtre(Buts,index(Buts));
  recevoir(Solutions);
  WHILE (Solutions != End_sol)
    BEGIN
      New_Env = Env U Solutions;
      IF (Queue=Vide) THEN envoyer(Suc_processus,New_Env)
      ELSE BEGIN
            Recursive=FALSE;
            Nb=nombre de littéraux dans la queue;
            FOR i=1 TO Nb DO
              IF (prédicat(Queue[i])=prédicat(buts)) THEN Récursif=True;
            IF Récursif
            THEN BEGIN
                  M_proc=créer(M-processus(but,Suc_processus,Env));
                  send(M_proc,M_proc);
                 END
            ELSE créer(OU-processus(Suc_processus,New_Env,Queue);
          END;
      recevoir(Solutions);
    END;
END.
```

Pour évaluer les clauses récursives, nous introduisons deux types de processus, M-processus (processus maître) et C-processus (processus de concaténation). La figure.III.8 montre le

fonctionnement de ces processus. Dans la figure, le M-processus est créé par le R-processus, tous les autres processus sont créés par le M-processus. Le M-processus extrait toutes les clauses récursives concernant l'évaluation d'un but et organise un pipeline pour chaque clause récursive. L'environnement Vide est transmis au début de chaque cycle pour le premier ET-processus de chaque clause, ce qui permet de lancer une évaluation dans chaque clause. L'ensemble des nouvelles solutions partielles D est transféré dans chaque cycle du M-processus à tous les points récursifs, c'est à dire aux C-processus.

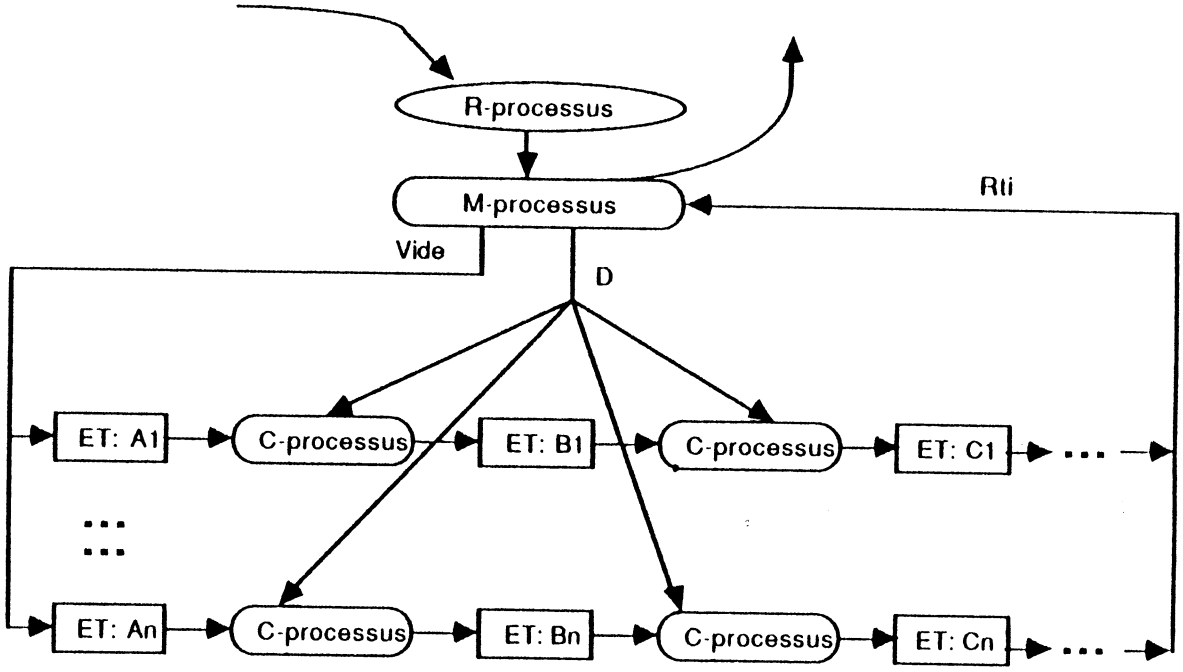

Fig. III.8

Le C-processus ajoute les résultats partiels D reçus du M-processus dans son tampon et vérifie la cohérence avec des environnements issus de son prédecesseur. Cette dernière opération est une sous opération du R-processus.

Le M-processus regroupe les résultats partiels produits par chaque clause et effectue la sélection de ces résultats pour engendrer D. Il envoie D aux C-processus. La communication avec un niveau supérieur est aussi assurée par le M-processus.

Dans la spécification des algorithmes du M-processus et du C-processus, l'opérateur "+"

signifie la concaténation de deux ensembles de solutions, ce qui correspond à l'union en algèbre relationnelle. Une fonction Select(E1,E2) est utilisée, elle sélectionne les éléments qui appartiennent à E1 mais pas à E2.

- E = Select(E1,E2): sélectionne e où e∈E1 et e∉ E2.

L'algorithme III.8: M-processus.

```
M-processus(But,Suc_processus,Env)
BEGIN
  /* recherche toutes les clauses récursives concernant le but */
  recevoir(M_proc);                          /* nom du M-proc lui même */
  créer(R-processus(But,M_proc,Vide));       /* cherche des clauses récursives */
  recevoir(R_clauses);          /* R_clause est un tableau qui contient le paquet de clauses */
  Qe=la queue de la clause de sortie;

  /* création des sous processus */
  Nb_point_rec=0;
  Nb_clause=nombre de clauses récursives;
  FOR i=1 TO Nb_clause DO
   BEGIN
    Nb=nombre de littéraux de la queue de la clause R_clause[i];
    FOR j=1 TO Nb DO
     IF (prédicat(but)=prédicat(Queue[j]))
      THEN BEGIN          /* point récursif */
             Num_proc[j]=créer(C-processus(Queue[j]));
             Point_rec[Nb_point_rec]=Num_proc[j];
             Nb_point_rec=Nb_point_rec+1;
           END;
```

```
        ELSE Num_proc[j]=créer(ET-processus(Queue[j]));
    Premier_proc[i]=Num_proc[1];
    FOR j=1 TO Nb-1 DO
        envoyer(Num_proc[j],Num_proc[j+1]);    /* nom du processus successeur */
    envoyer(Num_proc[nb],M_proc);        /* le successeur du dernier processus de chaque
                                            clause est le M-processus lui même */
 END;

/* initialisation de l'évaluation */
créer(R-processus(M_proc,Qe,Vide));      /* recherche sur la clause de sortie */
recevoir(D);                             /* D=E */
Résultat=Vide;

/* l'évaluation */
WHILE (D != Vide)
 BEGIN
   FOR i=1 TO Nb_clause DO envoyer(Premier_proc[i],Vide); /* déclencher recherche */
   FOR i=1 TO Nb_point_rec DO envoyer(Point_rec[i],D);    /* pour tous C-processus */
   FOR i=1 TO Nb_clause DO
      BEGIN recevoir(Env);     /* résultats issus de chaque clause */
            R_t = R_t + Env;
      END;
   D = Select(R_t,Résultat);    /* éliminer les doubles */
   envoyer(Suc_processus,ID);     /* envoyer les résultats */
 END;

/* final */
/* pour terminaison des fils */
```

```
    FOR i=1 TO Nb_clause DO envoyer(Premier_proc[i],End_env);
    /* détection la fin des fils */
    FOR i=1 TO Nb_clause DO recevoir(Env);
    envoyer(Suc_processus,End_env);
END.
```

L'algorithme III.9: C-processus

```
C-processus(Squelette)
BEGIN
  recevoir(Suc_processus);     /* du M-processus */
  recevoir(D);                 /* du M-processus */
  recevoir(Env);               /* du ET-processus prédécesseur */
  T = { } ;
  WHILE (Env != Vide)
    BEGIN
      T = T + D;
      New_Env = (T U Env) + (T U T);     /* "U"=joiture et "+"=union */
      envoyer(Suc_processus,New_Env);
      recevoir(D);          /* du M_processus */
      recevoir(Env);        /* du ET-processus prédécesseur */
    END;
  envoyer(Suc_processus,End_env);
END.
```

Une petite modification portant sur le C-processus permet de prendre en compte la première optimisation mentionnée dans III.2.5.3..

L'algorithme III.10: C-processus optimisé

```
C-processus(Squelette)
BEGIN
  recevoir(Suc_processus);
  recevoir(D);
  recevoir(Env);
  T = { } ;
  WHILE (Env != Vide)
    BEGIN
      New_Env = (T U Env) + (T U D) + (Env U D);
      T = T + D;
      envoyer(Suc_processus,New_Env);
      recevoir(D);
      recevoir(Env);
    END;
  envoyer(Suc_processus,End_env);
END.
```

La deuxième optimisation abordée dans III.2.5.3. peut être réalisée par le M-processus comme un cas particulier. Considérons les clauses (III-4.1) et (III-4.2), l'algorithme III.6 exige seulement un accès disque pour retirer toutes les clauses unaires E. La génération de toutes les solutions consiste en une opération semblable à la jointure dans les bases de données relationnelles. L'insertion de ce cas particulier dans l'algorithme III.8 est simple, donc nous ne présentons pas d'algorithme du M-processus avec cette optimisation.

III.3.5 Discussion

Les algorithmes de traitement de clauses récursives présentés ci-dessus permettent d'éviter la

prolifération des processus, donc diminuent le coût de gestion des processus. Mais leur traitement dans un contexte base de connaissances à grand volume est encore coûteux, car d'une part ils ne peuvent pas profiter totalement de la capacité de filtrage de données qui amène normalement une amélioration de performances, d'autre part, les solutions partielles générées dans l'algorithme III.8 contiennent toutes les solutions possibles pour une requête avec toutes les variables libres, ce qui est nécessaire pour la génération de toutes les solutions d'une requête avec des variables liées.

Dans une base de connaissances, l'espace de recherche peut être limité par des contraintes particulières sur une requête. Ces contraintes sont soit spécifiées dans le système par des experts, soit données explicitement par des utilisateurs; elles permettent de réduire les calculs inutiles. Dans cette optique, nous croyons que l'utilisation de systèmes basés sur des connaissances, comme systèmes experts, pour la recherche des solutions est une approche intéressante vers un traitement plus efficace.

III.4 Discussion

III.4.1 Le parallélisme

Le modèle présenté ci-dessus adopte une stratégie de recherche en largeur d'abord (breadth first). Cette stratégie s'applique aux clauses unaires lors de filtrage et aux règles lors de l'exploitation du OU-parallélisme. L'utilisation de cette stratégie conduit à une approche ensembliste qui augmente la granularité du traitement.

Dans le modèle, l'exploitation du OU-parallélisme se limite à l'évaluation parallèle des clauses non-unaires. Le parallélisme d'accès, qui consiste à accéder les clauses sur différentes unités de disques est un type de OU-parallélisme aussi, son exploitation se fait à un niveau inférieur, c'est à dire au niveau du système de stockage.

Le ET-parallélisme vérifie en parallèle les littéraux dans la queue d'une clause. L'évaluation en

pipeline dans le modèle n'est pas un ET-parallélisme dans ce sens là. Grâce au traitement ensembliste, ou à une stratégie de recherche en largeur d'abord, l'évaluation en pipeline présente effectivement une forme de parallélisme. On peut le considérer comme un autre type de ET-parallélisme, ou l'appeler parallélisme pipeline.

L'unification parallèle, c'est à dire l'unification de différents arguments d'un prédicat simultanément, n'est pas prise en compte dans beaucoup de systèmes Prolog parallèles, car l'analyse de programmes Prolog montre qu'elle présente peu d'intérêt [DKM84]. Dans un contexte de bases de connaissances, pour un grand volume de clauses unaires, c'est une alternative si le stockage se fait par argument (colonne) et non par clause (La machine base de données Delta utilise ce principe). Dans ce cas là, une recherche peut être réalisée par unification dans différents arguments en parallèle, plus vérification de cohérence pour générer les résultats définitifs.

III.4.2 Dataflow ou réduction

Le mécanisme de dataflow consiste à compiler un programme en un graphe et l'exécution est dirigée par les données. Son application à Prolog présente des difficultés à cause de la complexité de l'unification. Le mécanisme de réduction consiste à réduire un but par la règle de reécriture définie pour le but. Son inefficacité dans un contexte de bases de connaissances est due à la répétition d'applications de processus de réduction à un grand volume de clauses unaires.

Le modèle d'OPALE évite cet inconvénient du dataflow en appliquant une méthode d'interprétation, dont le principe est basé sur la réduction, pour établir le flux de données dans le temps d'exécution. Une fois que le flux de données est installé, le fonctionnement est dirigé par les données. Donc, on peut considérer que le modèle est une combinaison de dataflow et de réduction.

III.4.3 La granularité

La définition d'un modèle parallèle doit prendre en compte différents critères. Le plus

important est le granule du parallélisme. En général, un petit granule augmente le taux de parallélisme ce qui diminue le temps d'exécution. Mais l'augmentation du taux de parallélisme implique une gestion du parallélisme (synchronisation, dispatching...) plus lourd. Ce phénomène peut être schématisé par les courbes présentées dans la Fig.III.9. Donc, un système parallèle doit chercher un compromis pour la meilleure performance.

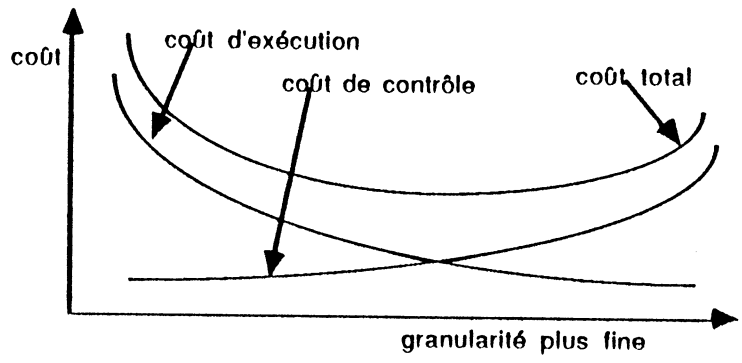

Fig. III.9

Trois possibilités existent pour la définition de granule du parallélisme: une clause, un bloc disque à taille fixée et toutes les clauses dans un paquet. Nous pensons que le granule de taille fixée correspondant à une piste du disque (16K-32K octets) est la meilleure solution car la rapilité du système est déterminée principalement par l'accès au disque. Ceci est cohérent avec les travaux de Boral et Dewitt [BoD82] sur la machine base de données Direct qui utilise la technique de dataflow. Un compromis est à rechercher en fonction du temps d'acces, de l'efficacité de l'indexation et de la taille moyenne des ensembles de données.

CHAPITRE IV

ARHITECTURE ORIENTEE PROCESSUS ET MACHINE OPALE

CHAPITRE IV

ARCHITECTURE ORIENTEE PROCESSUS ET MACHINE OPALE

Nous avons défini un modèle d'interprétation parallèle de Prolog dans le contexte base de connaissances pour OPALE. Ce modèle doit être supporté par une machine parallèle appropriée pour une exécution efficace. Dans ce chapitre, nous définirons une architecture multi-processeurs adaptée aux traitements du modèle développé. Aprés une présentation très rapide de l'architecture parallèle existante, nous dégagerons d'abord l'implication des systèmes reposant sur la notion de processus. Ce type de système reflète les caractéristiques des systèmes d'intelligence artificielle. La notion d'architecture orientée processus est ensuite présentée avec un exemple de langage concurrent orienté objet. Enfin nous décrirons l'architecture d'OPALE en nous basant sur les notions développées.

IV.1. Architecture Parallèle

Le développement de l'architecture parallèle consiste à utiliser plusieurs processeurs coopérant pour augmenter la puissance de calcul. Les machines parallèles existantes peuvent être classifiées selon différents critères [Maz78] [DDS80]. Nous considérons qu'une classification selon la relation entre le programme exécuté et la structure de la machine permet de mieux dégager le problème pour notre étude. Nous considérons les trois cas suivants :

- Architecture orientée algorithme : L'architecture de la machine correspond directement à l'algorithme. C'est le cas des machines systoliques [Kun82]. L'architecture de la machine est fixée

pour un algorithme spécifique. Le programmeur effectue la distribution du programme dans les différents composants de traitement. Ce type d'architecture fournit de meilleures performances, mais manque de généralité.

- Architecture orientée calcul : L'architecture de la machine correspond à la nature des calculs effectués dans le programme. Ce type de machine est composé de différents composants correspondant chacun à un type de calcul spécifique. C'est le cas des machines vectorielles, par exemple Cray [Rus78]. Dans ce genre de machine, la distribution du programme dans le système est effectuée par le compilateur. Dans le cas du Cray, le compilateur reconnaît les calculs vectoriels. Ce type de calcul est effectué dans l'unité vectorielle, tandis que les autres sont effectués par l'unité scalaire.

- Architecture orientée processus : L'architecture de la machine est organisée dynamiquement, la reconnaissance du parallélisme et la distribution des programmes dans le système est effectuée en temps d'exécution. Les processeurs de traitement peuvent être des unités indépendantes. La communication entre tous les processeurs doit être assurée.

En résumé, le premier type d'architecture est du type SIMD (Single Instruction stream Multiple Data stream) avec processeurs spécialisés et contrôle centralisé. Ce type de machine est orienté vers une application spécifique, par exemple le traitement d'image. Le deuxième type, basé sur la machine classique Von-neumman avec extension, est orienté calcul numérique. Le troisième type présente le cas général. Sa caractéristique essentielle est l'indépendance de l'architecture vis à vis du langage de programmation. Ceci rend la machine extensible, générale et tolérante aux pannes. Ce type de machine s'adapte mieux aux applications en intelligence artificielle, et sera développé dans la section suivante.

IV.2. Architecture orientée processus

IV.2.1. L'implication de modèles de processus

IV.2.1.1. La notion de processus

Beaucoup de systèmes parallèles en intelligence artificielle reposent sur la notion de processus. On peut citer ici les différents projets concernant Prolog : [CoK81], [CiH83], [Got84], [OAS85], [Tan85], ... Ceci est dû au fait que les opérations en intelligence artificielle sont beaucoup plus complexes qu'en calcul numérique, donc il est très difficile de décomposer les opérations en des granules plus petits. Par exemple, une unification prend de 0,1 milli seconde à 1 milli seconde sur M68000 d'après notre interpréteur qui va être décrit dans le chapitre VI. Ceci veut dire qu'une unification demande de 100 à 1000 instructions dans le cas du M68000.

La notion de processus à été introduite pour la description des systèmes d'exploitation. Les études théoriques remontent jusqu'en 1973 [Hor73]. En 78, Hoare utilisait la notion de "processus communicant" comme une base de programmation, et donnait naissance à CSP [Hoa78]. Le processus devient ainsi une notion de programmation .

Les processus des systèmes d'exploitation sont utilisés pour décrire le comportement des entités dans un système informatique. Un processus peut être défini par "un programme en exécution" et sa description est souvent compliquée. Par exemple, dans le système UNIX [Lio77], un processus peut accepter une interruption, demander des ressources, effectuer une entrée/sortie ; il peut être actif, "swappé" en mémoire secondaire, etc. La description d'un processus a besoin de 306 octets (sur PDP 11) pour les informations suivantes :

- informations générales : nom de l'utilisateur, "directory" actuel,...
- état : priorité, en mémoire centrale/secondaire,...
- interruption :
- environnement d'exécution : espace de travail, copies des registres,...
- création/destruction des processus : processus parent...
- informations de compte : temps d'exécution, temps d'exécution des fils,...
- :::

Par contre, dans un langage de programmation, tel qu'OCCAM [Inm84], le processus devient simple. Il existe trois types de processus de base:

affectation : par exemple Var : = 5

entrée : par exemple Canal ? Var

sortie : par exemple Canal ! 5

où Canal est le "support" de communication spécifié par le programmeur.

Un programme OCCAM est constitué à partir de ces trois types de processus avec des constructeurs SEQ (séquentiel), PAR (parallèle); ALT (alternative), IF, WHILE etc... La communication peut être traitée comme un processus et réalisée par des canaux de communication spécifiés par le programmeur.

La notion de processus dans les modèles d'interprétation en intelligence artificielle cités ci-dessus est différente. Les caractéristiques particulières peuvent être les suivantes :

- La communication est une fonctionnalité, ou bien une opération qu'un processus peut accomplir. La communication elle-même n'est pas un processus.
- Les processsus sont des unités minimales "schedulables" par un système, tous les processus sont en parallèle, les contraintes sur des processus sont retenues par la communication.
- Les opérations complexes demandées par des processus, par exemple les entrées/sorties, sont réalisées par des serveurs du système en passant des messages. C'est à dire que le système est organisé par un ensemble de processus. Le seul moyen d'interaction entre processus, ou bien le seul moyen de faire quelque chose à l'extérieur d'un processus, est de communiquer par passage de messages.
- Un processus demande une communication par le nom du processus récepteur. Il n'existe pas de moyen de communication fixé comme canal en OCCAM.

Au niveau du système, le nombre de processus est souvent limité dans le système d'exploitation (50 dans UNIX). Dans le langage OCCAM, il est très élevé car chaque énoncé est un processus. Le nombre de processus peut être déterminé en temps de compilation. Le nombre de processus dans les modèles d'interprétation en intelligence artificielle est imprévisible, car les systèmes ont souvent la caractéristique non-déterministe.

Nous pouvons définir la notion de processus ci-dessous :

Définition IV.1 : Processus : Un processus est une unité minimale "schedulable" qui a un ensemble de variables internes exprimant l'état interne du processus, et un ensemble de fonctions qui peuvent être appliquées aux variables internes pour changer l'état du processus. Un processus peut recevoir les messages envoyés par d'autres processus en parallèle et peut envoyer des messages aux processus qu'il connaît.

D'après cette définition, un processus peut avoir les états suivants :

- Actif en exécution.
- Actif en attente.
- Attente pour envoyer un message.
- Attente pour recevoir un message.

La transition d'états est donnée dans le figure IV.1.

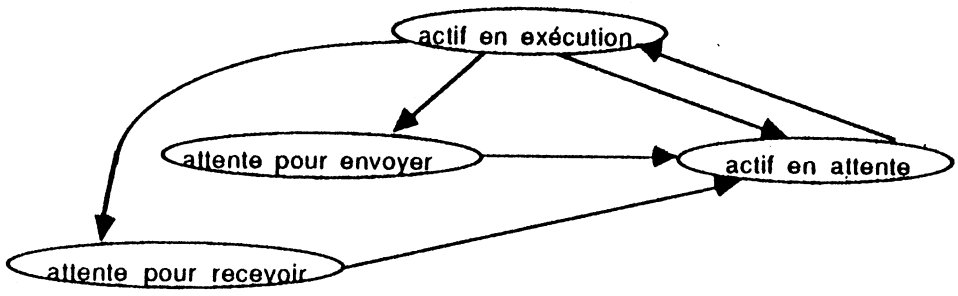

Fig.IV.1.

Puisque un processus peut être en attente, un processeur doit accepter plusieurs processus simultanément pour augmenter le taux d'utilisation. Donc le processeur supportant ce type de processus doit assurer au minimum les fonctionnalités suivantes :

1) Fournir le moyen de communication
   - envoyer un message.
   - recevoir un message.

2) Effectuer la gestion de processus dans le processeur :
   - fournir le contexte du travail : l'espace du travail,...
   - organiser les files d'attente pour l'envoi de messages et l'allocation de proceseur.

Les fonctionnalités ci-dessus exigent un granule matériel, c'est à dire un processeur, du type machine Von-neumman, aux entrées/sorties près. Donc un processeur doit avoir au moins une mémoire locale et certains dispositifs logiciels pour assurer les traitements propres à chaque processus, la communication inter-processeur et la gestion de processus dans le processeur.

IV.2.1.2. La gestion de processus dans un système

En se basant sur la notion de processus définie dans la section précédente, nous pouvons définir le système de processus :

Definition IV.2. : Système de processus : Un système de processus est constitué par un ensemble de processus. A l'initialisation, le système contient les processus du système et un processus utilisateur initial. Le système évolue lors de la création de processus par le processus utilisateur initial .

Nous constatons que la gestion de processus fait intervenir les opérations suivantes :

1) Fournir les services du système :

- création d'un processus.
- détection de l'état d'un processus.
- destruction d'un processus.
- :::

2) Effectuer le "scheduling" :

- distribuer les processus dans le système.
- éviter les interblocages.
- équilibrer la charge de chaque processeur.
- fournir l'espace de travail quand la mémoire locale d'un processeur n'est plus suffisante.
- assurer la transmission de messages entre les processeurs.
- :::

Il est évident que toutes ces fonctions n'ont pas besoin d'être installées dans tous les processeurs. Donc les serveurs spécialisés du système sont nécessaires. En outre, les données en intelligence artificielle sont souvent des entités sémantiquement bien définies, donc l'utilisation d'une mémoire intelligente (mémoire gérée par microprocesseur) permettra de faciliter la manipulation des objets complexes, puis la programmation.

IV.2.2. Architecture orientée processus

Vue l'implication des modèles de processus, chaque machine destinée à ce type d'application doit être organisée en adaptant le comportement de chaque modèle particulier. Mais en général, l'architecture orientée processus peut être représentée par une collection de processeurs reliés par un réseau de communication (Figure IV.2.).

Dans la figure IV.2., les processeurs de contrôle réalisent les services au niveau du système,

ils sont chargés de recevoir les travaux demandés par utilisateurs, de distribuer les processus dans le système, et de collectionner les résultats de traitements pour les retourner aux utilisateurs.

Les processeurs de traitement assurent les traitements de chaque processus, ils peuvent accéder aux informations partagées du système en communiquant avec les processeurs de gestion de l'information partagée, et demander les ressources de stockage par l'intermédiaire de processeurs de gestion de mémoire de travail.

Le réseau de communication est un composant critique du système. Le bus, multi-bus, cross-bar, réseau ADM et IADM [Fen79] ou n-cube [Pea77] sont des exemples qui peuvent assurer la communication entre les processeurs. Puisque le nombre de processeurs peut être de l'ordre de 1000, une connexion complète ne sera pas réaliste. Donc le choix du réseau est très important puisque de la configuration du système dépendent les connexions permises par le réseau.

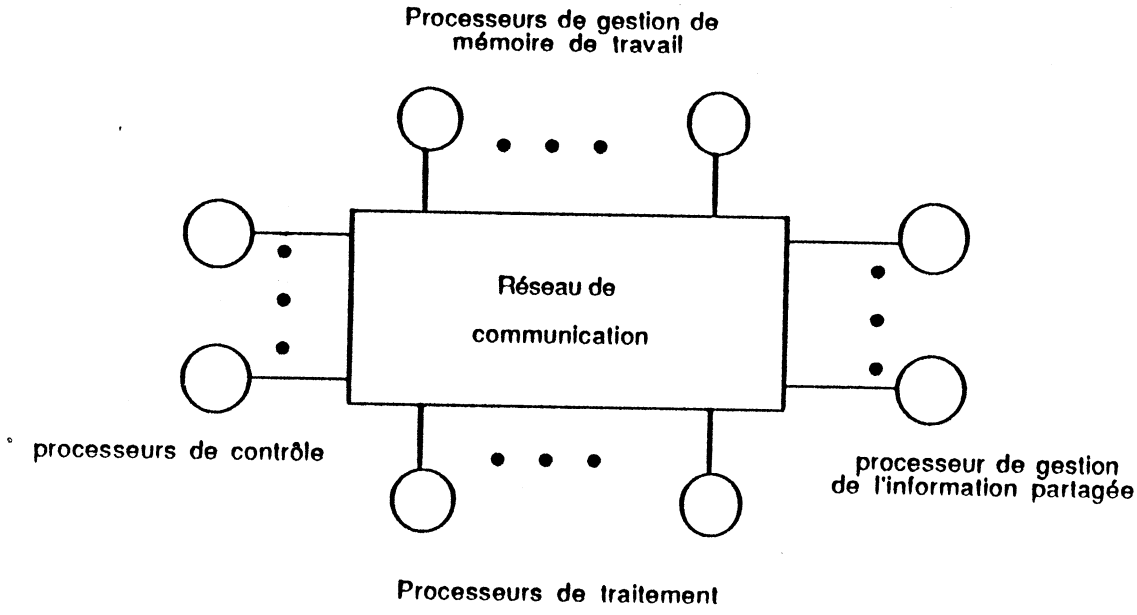

Fig.IV.2

Les principaux problèmes dans ce genre de machines sont les suivants :

- Comment distribuer le programme et les données dans le système ? Par exemple, dans un

système Prolog distribué, [WaD84] installe les principaux prédicats évaluables dans chaque site d'un réseau local. Les programmes utilisateurs sont répartis dans différents sites (processeurs de traitement).

- Comment représenter le programme et les données dans la machine ? [Cra85] résume les différentes techniques proposées pour l'exécution du OU-parallélisme pour Prolog. [War85] présente une autre réprésentation de la pile d'évaluation pour avoir un contrôle plus souple en faveur de l'exécution parallèle de Prolog.

- Comment et quand invoquer et contrôler le parallèlisme ? Il s'agit de l'évaluation à la demande (lazy evaluation) ou bien l'évaluation dès que possible (eager evaluation). Le projet ORBIT [YaN84] a adopté la première, mais beaucoup d'autres systèmes ont utilisé la deuxième [OAS85] [Got84].

- Comment reconnaître et profiter de la localité des programmes et données ? La localité du calcul et de la communication est une propriété naturelle existant dans un programme. L'utilisation d'un cache est un bon exemple dans le cas d'une machine séquentielle. Mais dans le cas d'une machine parallèle, puisqu'il est très difficile de prévoir la localité, une prise en compte au niveau de la distribution des processus est difficile à réaliser. Le système multi-processeurs Cm* [JCD78] a montré qu'il est difficile de profiter de la localité des programmes impératifs.

- Comment reconfigurer le système pour une performance optimale ? La reconfiguration dépend principalement du réseau de communication. Le réseau détermine l'efficacité de l'accès aux mémoires et de la transmission entre processeurs. C'est la raison pour laquelle un grand effort à été effectué dans ce domaine. [OAS85] et [Tan85] ont utilisé les réseaux à routage, [OAS85] a adopté un réseau intelligent avec tampons dans les nœuds du réseau.

- Comment réaliser le contrôle pour éviter le goulot d'étranglement ? Dans un système multi-processeurs du type MIMD, un contrôle centralisé présente le goulot d'étranglement. Les

contrôles hiérarchiques et décentralisés sont des alternatives possibles [Maz78].

Ces problèmes sont des grands sujets qu'on rencontre dans le développement de machines orientées langages en intelligence artificielle. Même une discussion partielle dépasse la capacité de cette thèse. Nous proposons aux lecteurs de se reporter aux publications suivantes pour des discussions plus détaillées [Veg84] [Mag85] [TBH82]. Nous présentons simplement dans la section suivante un exemple du langage concurrent orienté objet concernant certains aspects de ces problèmes.

IV.2.3. Un exemple : langage concurrent orienté objet

Les langages orientés objet sont considérés comme une des trois approches principales en intelligence artificielle pour le calcul de nouvelle génération. L'approche orientée objet est utilisée dans les domaines des systèmes experts [Roc84], bases de connaissances [SIS85, CaD86], langages de commande [Sno83] etc... Le développement de langages parallèles ou concurrents orientés objet a été récemment lancé [Dan86] [YMS86].

Nous présentons dans cette section quelques aspects développés par l'auteur au cours de la préparation de cette thèse [Dan86], et discutons de l'architecture en vue de mieux supporter le langage.

IV.2.3.1. Modèle Acteur et Langage Orienté Objet

Des programmes concurrents consistent en un ensemble de processus parallèles, ces processus devant communiquer et se synchroniser pour effectuer un travail en commun. Dans un langage orienté objet (LOO), le parallèlisme découle naturellement de la notion d'objet. On peut considérer qu'un objet est un processus ayant un état interne, et son activation est déclenchée par la réception de messages.

Cette notion se conforme bien au modèle acteur pour le traitement réparti [Hew77] [Mac79], qui a donné naissance au langage PLASMA. Certains langages, comme SMALLTALK [Gor83], FORME [Cor84] et MERING [Fer84], ont introduit le parallèlisme pour résoudre les problèmes concernant la concurrence. Mais de par le non-déterminisme du modèle acteur, le problème de synchronisation par transmission de messages est difficile à modéliser avec les mécanismes existants dans les langages, de sorte que quelques problèmes, comme le calcul systolique, ne peuvent pas être réalisés naturellement en LOO.

- IV.2.3.1.1. La Gestion de Messages Dans le Modèle Acteur

Le modèle acteur et les LOO se développent interactivement. Du point de vue du parallélisme, on peut considérer que les LOO sont basés sur le modèle acteur. Un objet est équivalent à un acteur, ce dernier est un processus indépendant et se déroule parallèlement avec d'autres acteurs. Il contient un environnnement représentant l'état interne, et un ensemble de procédures opérant sur l'environnement lorsque l'acteur est activé. L'activation d'un acteur est déclenchée par la réception d'un message envoyé par d'autres acteurs. Durant l'activation, selon la spécification du message, l'acteur peut modifier son état interne, créer d'autres acteurs, ou envoyer des messages aux autres acteurs.

A noter que dans ce modèle, la transmission des messages est totalement asynchrone, c'est à dire que les deux acteurs participant à la transmission n'ont pas besoin d'attendre l'un et l'autre. L'émetteur transmet le message, en supposant que les messages envoyés arrivent toujours au récepteur, et que le récepteur a un tampon suffisant pour la réception. Seulement la réception d'un message influe sur le calcul. Cela entraine le non-déterminisme du modèle : l'ordre des messages envoyés chez l'émetteur peut être différent de celui du récepteur, et quand plusieurs messages arrivent à un acteur, le choix d'un message est totalement aléatoire.

Dand le modèle acteur, la transmission de messages est le seul moyen d'interaction entre des objets. En considérant la transmission d'un message comme un moyen de synchronisation, la

réception d'un message doit être un énoncé apparaissant explicitement dans le programme, il a pour rôle de :

1) spécifier le moment de la réception d'un message : chez l'émetteur, l'envoi d'un message ne peut être réalisé que si le récepteur exécute cet enoncé ; chez le récepteur, si l'émetteur n'est pas prêt à envoyer le message, le processus est suspendu jusqu'à ce que l'émetteur soit prêt à lui envoyer le message.

2) spécifier la forme du message (type, nombre d'arguments...), et éventuellement comment le traiter (affecter la valeur reçue à une variable par exemple).

3) spécifier la source du message (canal, porte, processus...).

On constate bien que cette approche est contradictoire avec l'idée initiale de la manière dite générique, c'est à dire que l'objet récepteur à la capacité d'appliquer une procédure de traitement en fonction de l'état de l'objet et du message reçu. Donc on cherche une solution compatible en opérant seulement sur la réception de messages.

On peut considérer que, attaché à chaque acteur, il y a un arbitre implicite gérant les messages reçus. Notre solution consiste donc à rendre cet arbitre explicite et à donner la possibilité de contrôler la gestion de messages par programmation. Nous verrons ci-dessous avec des exemples les mécanismes de synchronisation existants de ce point de vue.

IV.2.3.1.2. Partage de Ressources

SMALLTALK, et d'autres LOO, ont introduit la notion de valeur retournée : à chaque envoi de message, l'objet émetteur attend une réponse du récepteur avec une valeur . On peut considérer que c'est un mécanisme de synchronisation réalisé par le biais de l'arbitre associé : après un envoi d'un message, l'objet émetteur suspend son activité en attendant la réponse. A ce moment là, tous les messages reçus par l'arbitre seront temporisés sauf la réponse issue du récepteur. Ce contrôle est réalisé implicitement par l'interpréteur du langage SMALLTALK.

Grâce à ce mécanisme, SMALLTALK a pu établir une classe sémaphore, qui ne retourne une valeur pour le message "wait" que si un message "signal" a été reçu. En utilisant la classe sémaphore, un problème concernant le partage de ressource par plusieurs objets peut être facilement résolu en LOO. Des exemples ont été présentés dans [Gor82] [Dan86].

IV.2.3.1.3. Les Messages Retardés

Le langage MERING a introduit un moyen de synchonisation appelé "messages retardés", la réception d'un message change l'état d'un objet, mais l'effet est réalisé par des procédures d'attachement, qui ne seront pas invoquées par la réception d'un message mais par certaines conditions représentant l'état de l'objet.

Exemple IV.1 : calcul dataflow.

Problème : Un calcul est défini par la Fig. IV.3: quand les données X et Y arrivent, le calcul est effectué et le résultat X + Y est envoyé à R1; le même principe s'applique ensuite pour R1 et Z.

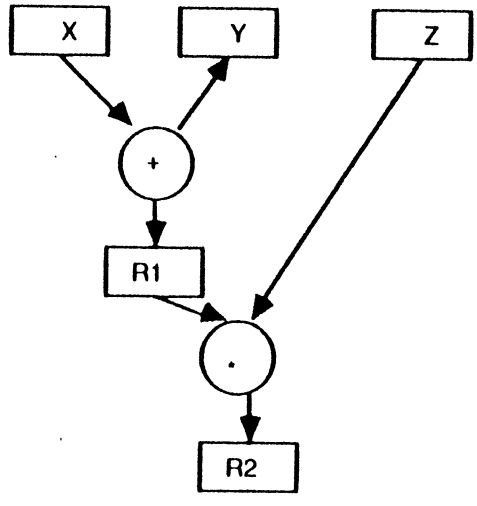

Fig.IV.3

Programme :

```
[delayed  when [AND [X known]
                    [Y known]]
          do [R1 <= X + Y ]]
[delayed  when [AND [R1 known]
                    [Z known]]
          do [R2 <= R1 * Z ]]
```

Le message retardé peut être aussi considéré comme un mécanisme de synchronisation : un groupe d'objets opère sur un objet par transmission de messages. Quand tous les messages arrivent à l'objet récepteur, ce dernier exécute l'action correspondante. Ceci est équivalent à une spécification d'un groupe de réception des messages. L'ordre d'arrivée de ces messages n'est pas pris en compte dans ce mécanisme. On verra dans la prochaine section que ceci présente un inconvénient pour certains types de calculs synchrones.

IV.2.3.2. Un Mécanisme de Synchronisation

IV.2.3.2.1 Le Problème des Systèmes Systoliques

Nous commençons par un problème typique de calcul synchrone.

Problème : Multiplication de deux matrices IN[m, n] et A[n, n], IN X A.

Rappelons que KUNG [Kun 82] a défini une architecture systolique comme une machine spécialisée pour effectuer le calcul. Supposons n = 3 ; la configuration de la machine est donnée en Fig.IV.4, chaque nœud contient une valeur A[i, j]. Il reçoit une valeur IN[k, i] (k=1..m) et une somme intermédiaire S. Il s'agit de calculer Snew = S + IN[k, i] * A[i, j], et de transmettre Snew et

IN[k,i] pour ses voisins.

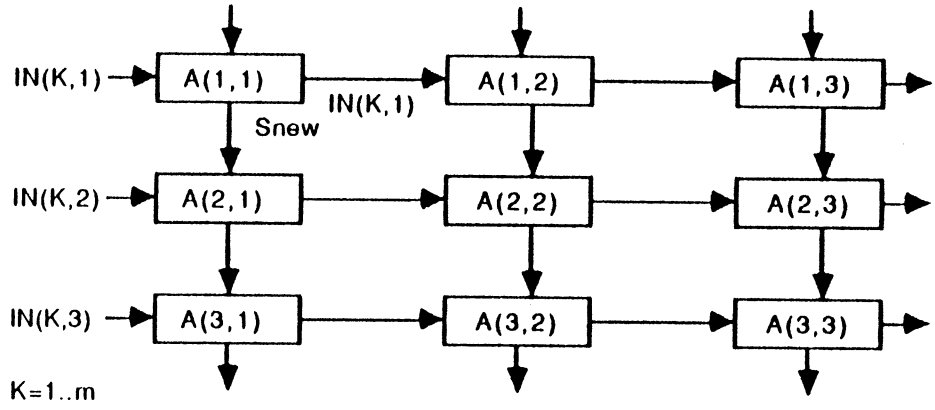

Fig.IV.4

Le calcul est totalement synchrone. Avec les mécanismes existants, la synchronisation entre deux messages issus de différentes sources dans un noeud de calcul est difficile à réaliser : on ne peut pas empêcher l'envoi d'un message contenant IN[k+1,i] pour que IN[k,i] ne puisse pas être effacé avant l'arrivée de S. Nous aborderons ce problème dans la suite du chapitre avec un nouveau mécanisme de synchronisation.

IV.2.3.2.2 L'objet Message_manager

Comme on l'a indiqué auparavant, dans le modèle acteur, on suppose qu'il existe un arbitre implicite qui gère les messages reçus. Nous établissons une classe du système message manager qui joue le rôle de gestion des messages. Un objet message-manager peut être attaché à chaque objet processus soit par l'implémentation, soit par la programmation. Dès qu'un objet est attaché à un message_manager, tous les messages à destination de l'objet sont filtrés par lui. Il sélectionne les messages pour l'objet en fonction des messages qu'il en a reçu. En transmettant des messages au message_manager, un objet peut programmer l'ordre de réception des messages, temporiser des messages etc.

Comme la classe sémaphore, cette classe ne peut pas être exprimée par un programme

SMALLTALK ou un autre langage. Nous en donnons la définition suivante :

Message manager instance protocole

susp_until Mesg :

Suspendre l'objet, mettre tous les messages à destination de l'objet dans un tampon jusqu'à ce que Mesg arrive ; puis transmettre Mesg, et les autres messages.

no_more Mesg :

Mettre les messages Mesg reçus dans un tampon jusqu'à ce que "more Mesg" arrive.

more Mesg :

Annuler l'effet de "no_more Mesg".

A noter que l'on suppose que l'objet a une priorité de communication avec cet arbitre, c'est à dire que les messages envoyés à la destination "self" seront traités avant tous les autres.

Une implémentation de la classe message_manager a été réalisée en Prolog-concurrent [Sha83a] [Dan85]. Quelques exemples ont été testés sur cette maquette.

IV.2.3.2.3 Les Applications

Avec ce mécanisme de synchronisation, en utilisant aussi ceux qui existent déjà (sémaphore et procédures d'attachement), le problème de la synchronisation entre des objets peut être résolu naturellement. Dans l'exemple IV.3, nous utilisons "condition" pour exprimer les procédures d'attachement mentionnées en IV.2.3.1.3.

Exemple IV.2 : Spécification de l'ordre de la réception des messages dans un objet, par exemple dans un ordre M1, M2, M3.

Instance protocole

M1

: : : :

self susp_until M2

M2

: : : :

self sup_until M3

M3

: : :

Exemple IV.3 : calcul systolique.

Programme :

Instance protocole

S Vals

Stam <= Vals,

Flags <= true,

self no_more S Vals.

IN Valb

INtam <= Valb,

FlagB <= true,

self no_more IN Valb.

Condition

```
IF
   [ FlagS AND FlagB ]
THEN
   [ Snew <= Stam + INtam*A,
    suit_gauche INtam,
    suit_bas snew,
    FlagS <= false,
    FlagB <= false,
    self more IN Valb,
    self more S Vals
   ]
```

IV.2.3.3. La méthode d'implémentation

Puisque les langages concurrents orientés objet sont issus du modèle acteur, l'architecture multi-processeurs s'adapte bien pour l'implémentation de ce type de langages. Nous verrons dans cette section comment les programmes peuvent s'identifier naturellement à l'architecture orientée processus, et influencer la conception de matériels.

IV.2.3.3.1. La création et distribution des objets

Comme on a indiqué auparavant, un objet est équivalent à un processus. Les objets créés peuvent être distribués dans différents processeurs de traitement. La création des objets peut être considérée comme un service du système réalisé par des processeurs de contrôle, car une fois qu'un objet est crée, il faut l'allouer à un processeur. Donc l'allocation des objets et la création des objets sont des opérations liées et doivent être assurées dans un même type de processeur.

On suppose qu'un objet du système "CreateSomething" résidant dans les processeurs de contrôle accepte les demandes de création des objets. En recevant un message de demande de

création d'un objet, l'objet "CreateSomething" crée l'objet demandé en suivant les paramètres de la la demande. Ensuite, l'objet créé est passé à l'objet du système "Scheduler". L'allocation de processeur pour l'objet créé se fait selon la charge du système par "Scheduler" qui contient les informations globales du système.

L'évolution du système sera connue par "Scheduler" si la création des objets et la destruction des objets (y compris si un objet se suicide) sont notées dans un centre d'informations géré par "Scheduler". La disparition d'un objet doit être communiquée à l'objet "Scheduler".

IV.2.3.3.2 La communication

Chaque processeur doit se munir d'un mécanisme de gestion de communication. La réception des messages doit pouvoir être réalisée en parallèle avec le traitement dans le processeur. Dans le cas de "message_manager" présenté dans la section précédente, la gestion de communication peut être assurée par deux files d'attente par objet. Une file pour temporiser des messages dans le cas suivant :

1) L'objet est actif, il ne peut pas traiter tout de suite les messages arrivant.

2) L'objet attend la réponse d'un objet avec lequel il a eu la communication. Il ne traite pas d'autres messages dans cet état.

3) L'objet est suspendu par "self susp_utill Mesg" et le message arrivant n'est pas le message spécifié.

4) L'objet est libre, plusieurs messages suspendus par "no_more Mesg" sont libérés par "more Mesg". Il ne peut pas les traiter tous à la fois.

Une autre file à pour effet la temporisation des messages spécifiés par "no_more Mesg" , tous

les messages reçus correspondant à Mesg sont amenés dans cette file jusqu'à ce que "more Mesg" arrive.

IV.2.3.3.3. Le partage de connaissances

Une propriété essentielle des langages orientés objet est la façon de partager des connaissances par le mécanisme de l'héritage. Dans le cas de l'environnement réparti, l'héritage peut être modifié comme un lien implicite correspondant à la relation génétique lors de la création des objets.

Une autre façon de réaliser le partage des connaissances est la "délégation" [Lie86]. Au contraire d'un lien implicite du mécanisme de l'héritage, la délégation est une méthode explicite et programmable. Dans chaque objet, on doit spécifier comment traiter un message inconnu : le déléguer à un autre objet ou l'ignorer.

Dans le cas d'un environnement réparti, l'implémentation de ces deux mécanismes nécessite que le nom d'objet puisse être manipulé comme une variable contenue dans le message. Ceci implique qu'au niveau du langage, la notion du nom d'objet manipulé par programmation est une approche qui facilite la réalisation.

Le partage de connaissances est non seulement une méthode évitant la multiplication des procédures de traitement et des données, mais aussi une méthode de régularisation de la charge du système. Par exemple, un processeur de contrôle s'occupe de certains services du système et il y a trop de messages à traiter par les objets dans ce processeur, qui donc devient le "point chaud" du système; dans ce cas une partie des services qu'il doit rendre peut être transmise à un autre processeur, les messages demandant ces services peuvent être délégués et donc réalisés ailleurs ; tous ces changements sont transparents aux objets utilisateurs et permettent d'éviter le changement de destination au niveau du système.

IV.3. La machine OPALE

IV.3.1. Généralité

La machine OPALE vise à supporter les bases de connaisances reposant sur Prolog. Les machines bases de connaissances manipulent de grands volumes de données, donc ont les nécessités particulières suivantes :

- Extensibilité : Les bases de connaisances sont des systèmes évolutifs dans le sens que le volume de données stockées augmente avec une croissance monotonique. Donc la capacité de traitement dans le système doit augmenter facilement en répondant aux besoins dès que nécessaire. Une autre raison est que les machines bases de connaissances sont des systèmes qui nécessitent beaucoup de matériels (surtout la mémoire secondaire par exemple). Ceci correspond à un investissement relativement important. L'extensibilité permet de construire un système minimal relativement moins cher, et d'ajouter des composants dès que possible et nécessaire.

- Tolérance aux pannes : Une base de connaissances est souvent intégrée dans un système comme serveur pour fournir des connaisances à plusieurs machines de traitement (par exemple machines à inférences), elle joue un rôle central dans le système englobant. Une panne de la machine bases de connaissances peut bloquer le système entier. Donc la tolérance aux pannes est une propriété nécessaire. En plus, le problème de sécurité des données est très important car elle contient les informations partagées.

- Traitement et communication intensive : Les machines bases de connaissances manipulent de grands volumes de données. Leurs processeurs de traitement doivent avoir une mémoire locale relativement importante. La communication entre différents processeurs n'est pas seulement les messages de synchronisation et quelques paramètres, mais aussi beaucoup de résultats intermédiaires. Le volume de communication est important. La communication est probablement le goulot d'étranglement. Un système de communication adéquat permettra d'améliorer sensiblement les performances.

L'organisation du système en reposant sur une philosophie d'architecture orientée processus répond bien à toutes ces nécessités. La modularité de l'architecture orientée processus permet d'ajouter facilement tous types de composants au système. L'utilisation dégradée en cas de panne d'un composant est assurée puisque tous les composants ont plusieurs exemplaires. La faute d'un composant est localisée, donc elle ne peut pas se propager ailleurs car chaque composant fonctionne indépendamment.

Au niveau de la technologie, l'évolution vers les compilateurs de silicium [Anc83] nous permet de réaliser les circuits spécialisés avec un coût modéré. Donc des processeurs du disque comportant des opérateurs filtres en VLSI sont envisagés dans la machine. Le disque winchester de petite taille est considéré comme l'unité de stockage appropriée dans OPALE. Ceci permet de décentraliser les données avec une répartition plus fine, et donc fournit la possibilité d'exploitation de l'accès en parallèle et évite le point critique dans un système.

Une vue globale de l'architecture d'OPALE est présentée dans la figure IV.5. Dans ce chapitre, nous traitons les problèmes concernant la distribution de processus dans le système et dégageons les problèmes de la conception des différentes parties du système. En analysant la nature des calculs effectués dans chaque processus, nous discutons d'abord les processeurs qui supporteront différents types de processus ; ensuite nous analysons les communications entre les processus et le système de communication dans le système ; enfin une analyse de besoins du "scheduler" est présentée.

IV.3.2. La représentation des processus

le modèle défini dans le chapitre III est composé de trois types de processus principaux : OU-processus , ET-processus et R-processus. En cas de clauses récursives, les M-processus et C-processus sont introduits. En considérant les clauses récursives comme des cas particuliers, nous discutons seulement les trois types de processus principaux.

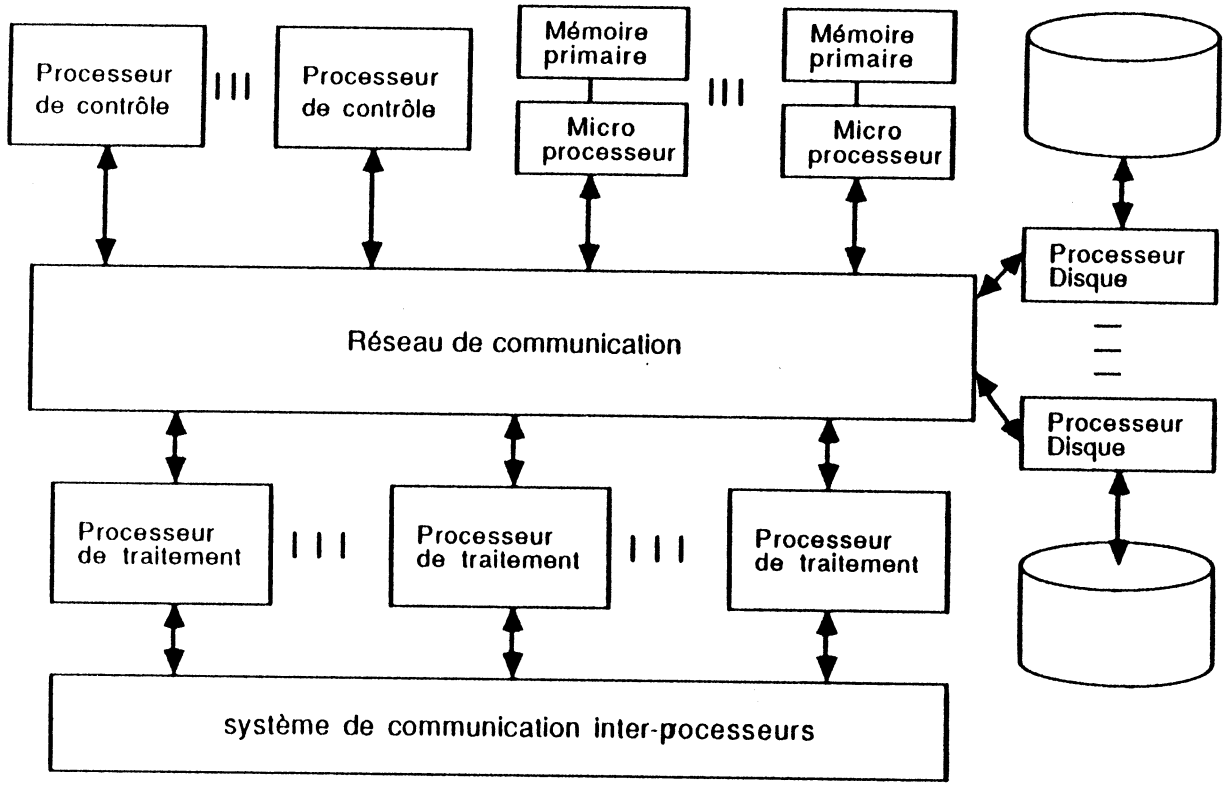

Fig. IV.5.

Une requête est évaluée par un réseau de processus communicant, mais la compléxité de chaque type de processus est différente. Un OU-processus n'a rien d'autre à faire, après la création des ET-processus fils, que d'attendre la "terminaison" de ses processus fils. Des R-processus ont une correspondance avec un ET-processus. Donc la représentation du réseau de processus peut être réduite à une structure dynamique arborescente que nous appelons l'arbre de processus. A chaque nœud correspond un ET-processus. L'arbre de processus reflète les relations génétiques des processus et les relations de communication. Le réseau de processus dans l'exemple III.3 peut être présenté dans la figure IV.6.(a). L'arbre de processus comprend les données minimales pour distribuer des processus, ces derniers peuvent être très nombreux, dans un ensemble de processeurs physiques disponibles. Les OU- processus ont pour effet d'ajouter des nœuds dans l'arbre au fur et à mesure de l'évaluation d'une requête. Les R-processus produisent les données circulant entre les nœuds.

IV.3.3. La localité de la communication et le réseau

La localité de la communication et du calcul existent naturellement dans le modèle défini au chapitre III. Les réseaux matériels de communication qui réalisent l'interconnection des processeurs ne permettent pas toutes les C_n^2 sortes de connexions (n est le nombre de processeurs désirant participer à la communication). Donc, la localité de la communication doit être prise en compte pour une transmission plus efficace.

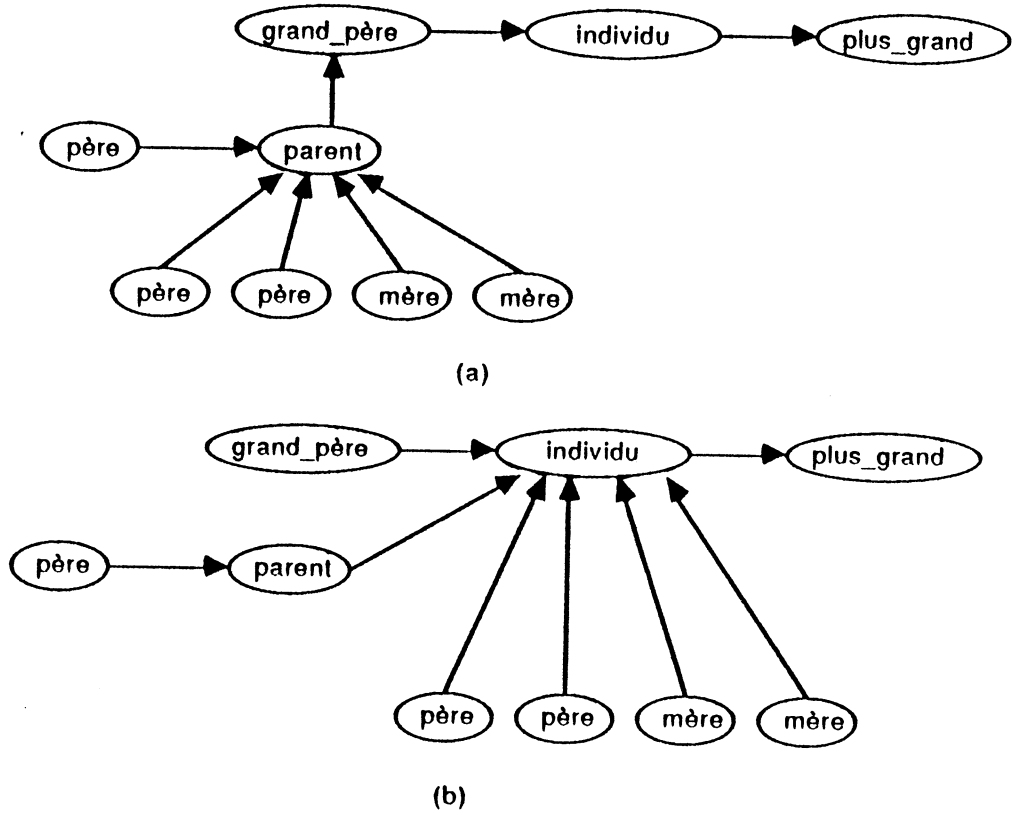

Fig. IV.6

Dans le modèle d'OPALE, on peut distinguer différentes communications entre les processeurs :

1) ET-processus => R-processus : les messages de communication doivent contenir les buts à vérifier, les environnements utiles à la suite d'évaluation et les informations de gestion. Le volume

de données à transférer est le plus grand par rapport aux autres types de communication. Pourtant, une partie des données, c'est à dire les environnements, peut être partagée entre les ET-processus et les R-processus correspondants si ces processus sont alloués dans un même processeur.

2) R-processus => OU-processus : la communication est relativement simple. Les informations transférées contiennent une clause, un environnement et les informations de gestion.

3) OU-processus => ET-processus : après avoir créé les ET-processus, un OU-processus transmet simplement les données de contrôle à ses ET-processus fils. Pour le ET-processus correspondant au premier prédicat de la queue de la clause, il faut transmettre en plus l'environnement initial pour qu'il puisse commencer l'évaluation.

4) R-processus => ET-processus : les environnements nécessaires à l'évaluation sont transmis au ET-processus suivant. Si le R-processus et le ET-processus correspondants résident dans un même processeur, on peut les représenter par R/ET -> R/ET.

En plus de ces quatre types de communication, la communication entre le système de stockage et les R-processus est très importante. Cette communication est aussi volumineuse. En résumé, la localité de communication est apparente entre les R-processus et les ET-processus. C'est à dire que les communications qui ont le plus grand volume correspondent aux données circulant entre les nœuds de l'arbre de processus (Fig.IV.6.(b)). Ceci veut dire que l'arbre de processus représente aussi la caractéristique de communication. En se basant sur cette analyse, l'architecture de la machine OPALE contient deux réseaux de communication (voir figure IV.5.), le réseau 1 pour les messages entre les systèmes de stockage et les processeurs, et le réseau 2 pour la communication inter-processeurs.

Le réseau 1 réalise toutes sortes de communications, sauf celles entre les processeurs de traitement. Il doit permettre la connexion entre les processeurs de contrôle, les modules de la mémoire primaire, les processeurs disques et les processeurs de traitement. Des études

supplémentaires doivent être menées sur ce réseau. Nous discuterons dans la suite du système de communication réalisant la communication inter-processeurs.

Comme on a dit auparavant, le modèle d'OPALE a une caractéristique arborescente au niveau de la communication. Une structure arborescente de processus peut être traitée par une machine à structure arborescente pour une distribution correspondante. L'expérience sur ce type de machine montre des difficultés sur deux aspects. L'un est que, lorsque la profondeur de l'arbre est plus grande que celle de la machine, il est difficile de profiter de l'avantage de ce type de machine [Pre81]. L'autre est que le goulot d'étranglement se présente toujours près du sommet de l'arbre de la machine, à cause de la communication. Pour éviter ces inconvénients, surtout le deuxième, car OPALE a une caractéristique de communication intensive, nous tenons à réaliser l'évaluation par pipeline à la place de l'évaluation arborescente.

La transformation de la structure arborescente en une structure linéaire peut être réalisée par un "scheduler" en temps d'exécution. Le principe de cette tranformation est basé sur les hypothèses que:

- les processeurs disponibles à un moment donné sont limités, donc la longueur du pipeline est déterminée par la limite du nombre de processeurs disponibles ;
- la multiplication de certains processus ne présente pas d'inconvénients (voir IV.3.4.)

Dans l'arbre de processus, chaque nœud correspond à un ET-processus. Par l'intermédiaire du dictionnaire du système, le "scheduler" peut savoir si la définition d'un littéral contient ou non des clauses non-unaires. L'organisation du pipeline commence par un nœud et regarde le nœud successeur, dans le cas où un prédicat est défini par des clauses non-unaires, le pipeline inclut ce nœud et termine , car l'évaluation du nœud demandera la création des processus fils et sera donc très longue à aboutir. Dans le cas contraire, on inclu ce nœud et continue à regarder les nœuds successifs jusqu'à un nœud qui demande la création des processus fils.

Exemple IV.4 : l'évaluation de l'arbre de processus présenté dans l'exemple III.3 (Fig.III.4)

peut être réalisée par trois pipelines :

père -> parent, père -> individu ->plusgrand et mère -> individu ->plusgrand.

L'évaluation en pipeline des processus entraîne un système de communication très simple. La connexion entre processeurs peut être réalisée par des liaisons une par une entre les processeurs de traitement, comme ce qui est présenté dans la figure IV.7 (a). En considérant le volume de communication, nous pensons que l'utilisation de mémoires locales intercalées est une solution alternative (figure IV.7 (b)).

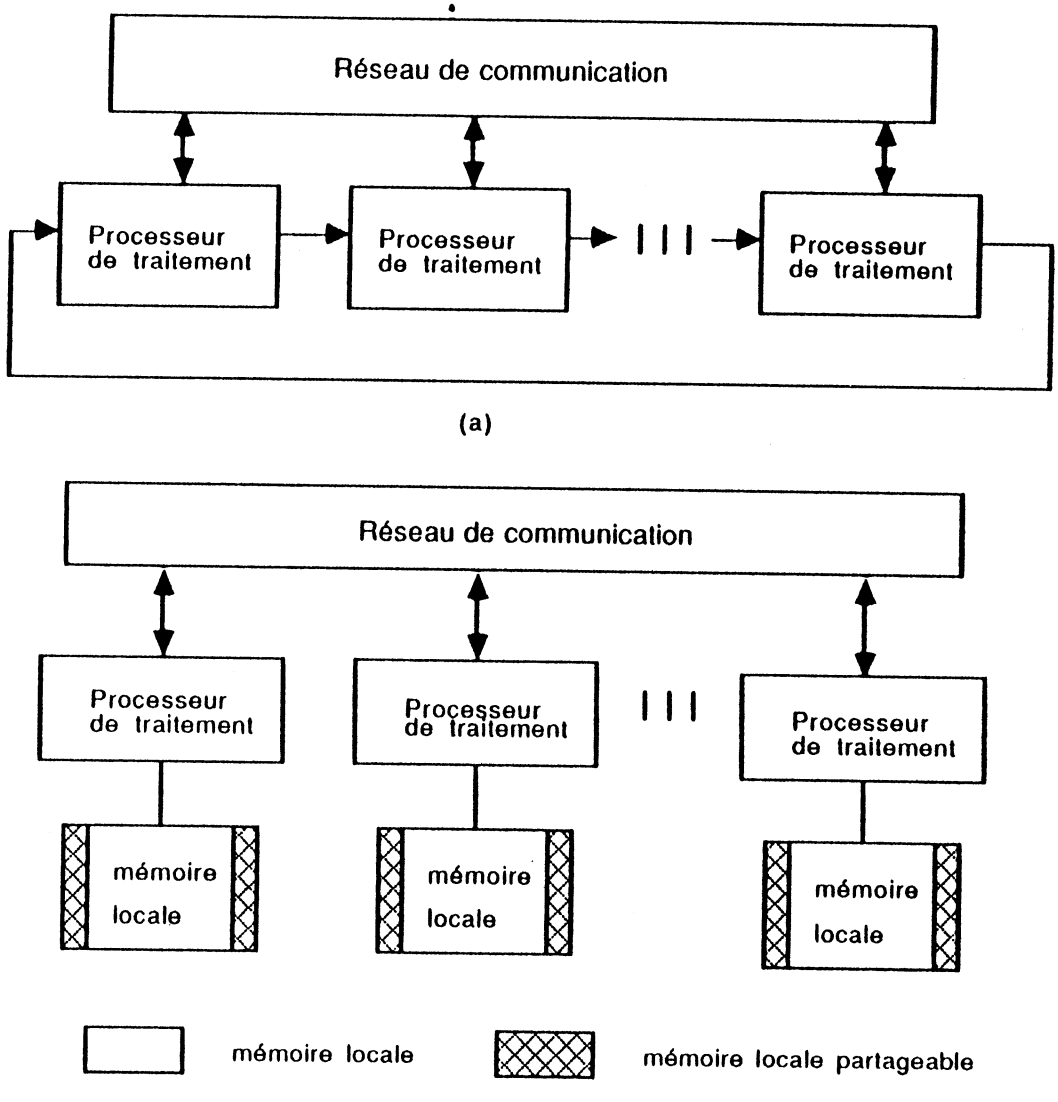

Fig. IV.7

IV.3.4. La localité du calcul et les processeurs

Dans la section précédente, nous avons discuté la localité de la communication dans le modèle et en avons déduit le système de communication pour la machine OPALE. Dans cette section, nous discuterons la localité du calcul dans chaque processus, et en déduirons l'exécution de ces différents processus dans les différents processeurs. Les natures des calculs des processus déterminent la conception de chaque processeur.

Le traitement effectué par le OU-processus est relativement simple. Le rôle des OU-processus est de créer des ET-processus fils. Ceci correspond à un ajout de nœuds sur l'arbre. L'arbre contient aussi les informations nécessaires à la distribution des processus. Un OU-processus doit partager les informations centralisées du système avec le "scheduler" qui réalise la distribution. Donc nous pensons installer les OU-processus et le "scheduler" dans les processeurs de contrôle. L'arbre de processus doit être stocké dans une mémoire partageable par des processeurs de contrôle.

Les ET-processus sont créés par un OU-processus. Une fois alloué dans un processeur, un ET-processus crée des R-processus. Il n'est pas nécessaire de noter la création des R-processus dans l'arbre de processus en raison de la correspondance entre ET-processus et R-processus. Les ET-processus peuvent être alloués dans les processeurs de traitement. Puisque la production des résultats intermédiaires par des R-processus dépend de l'accès aux disques, il est très difficile de résoudre le problème de synchronisation. Dans ce cas, la multi-programmation dans les processeurs de traitement est nécessaire. La mémoire locale attachée à chaque processeur de traitement doit avoir une taille importante car il y a plusieurs tampons à gérer pour les processus différents.

Les R-processus sont créés par un ET-processus. Les R-processus doivent accéder au système de stockage et leur durée d'éxécution dépend de la vitesse du disque. L'analyse de la communication entre un ET-processus et ses R-processus fils a prouvé que l'allocation d'un ET-processus et des R-processus fils dans un même processeur peut profiter du partage des

données et réduit la communication (voir IV.3.3). Ceci peut être modifié dans le temps d'exécution si les processeurs de traitement ont la capacité de répartir leur travail dans d'autres processeurs.

La multi-programmation dans les processeurs de traitement est faisable puisque les traitements de chaque processus sont "standard" dans le sens que le code de chaque processus de même type (ET ou R) est identique. La représentation d'un processus n'a besoin que de certaines informations de contrôle (état, nom... etc) et d'une file d'attente qui contient les données à traiter.

Actuellement, la gestion de processus n'a été prise en compte que par des processeurs relativement compliqués. Par exemple, dans la machine VAX 11/780, il existe des instructions "Store Process Context" et "Load Process Context" pour faciliter le changement de processus [Dig82]. Nous pensons que ce genre de support matériel est important dans l'architecture orientée processus, car les calculs parallèles à travers le modèle de processus ont besoin d'un mécanisme pour réduire l'"overhead" au niveau de la gestion de processus. Avec l'idée du "RISC" (Reduced Instruction Set Computer) [Das82], une nouvelle génération de microprocesseurs avec des supports matériels de communication, comme le Transputer [wil83], doit apparaître pour le calcul parallèle.

IV.3.5. Le contrôle du système

Le contrôle du système dans OPALE est réalisé à deux niveaux. L'un est le contrôle de stockage qui réalise l'organisation du dictionnaire du système, le contrôle d'accès concurrent etc. L'autre est le contrôle d'évaluation qui s'occupe de la recherche de l'information pour des requêtes d'utilisateurs. Nous discutons le deuxième type de contrôle, qui est essentiellement le problème du scheduler.

Le rôle du "scheduler" est de distribuer les processus, ces derniers peuvent être très nombreux, dans un ensemble de processeurs de traitement disponibles de nombre limité. La distribution des processus doit prendre en compte la régulation de charge du système, c'est à dire avoir une meilleure distribution, de sorte que le calcul puisse être réalisé le plus rapidement possible.

La distribution doit prendre en compte deux critères principaux : le coût de la communication et la charge de chaque processeur. En plus, le coût de contrôle, c'est à dire le coût dépensé pour avoir la meilleure distribution doit être minimisé. Le compromis recherché présente un même comportement que celui de la figure III.9. Le but du "scheduler" est de trouver une distribution de coût minimum pour la communication et de meilleure régularisation de la charge. La régularisation de la charge peut être définie par un Facteur d'Irrégularisation de la Charge (FIC) (Unbalanced load factor) comme suit :

$$\sum_{j=1}^{n} \frac{(N(P_j) - N_{mm})^2}{n}$$

où - n est le nombre de processeurs,

- $N(P_j)$ le nombre des tâches allouées dans le processeur P_j,

- N_{mm} la moyenne du nombre de tâches dans les processeurs.

L'objectif du "scheduler" peut être défini ci-dessous :

$$\text{Min } (\text{Coût}_{\text{comm.}} + C_{\text{coif}} * \text{FIC}) \qquad \text{(IV-1)}$$

Le coût de communication dépend de l'organisation du système de communication. Il peut être représenté par une fonction :

$$\sum_{h=1}^{m} \left(\sum_{k=1}^{L_h} \text{coût}_k(P_i, P_j) \right)$$

où coût (P_i, P_j) désigne le coût de communication entre processeurs P_i et P_j quand deux processus communicants sont alloués repectivement dans P_i et P_j; m désigne le nombre de tâches à allouer et L_h le nombre de processus dans la tâche h.

. En général, la minimisation du coût de la communication et la régulation de charge sont deux critères contradictoires [Lu86]. Dans le cas d'OPALE, nous considérons que la communication est

le premier caractère à prendre en compte. Donc on alloue toujours les processus communicants dans des processeurs voisins, le coût de la communication devient constant et peut être supprimé dans la formule IV-1.

Le problème de distribution de processus dans la machine OPALE peut être spécifié de la manière suivante :

Etant donné un groupe de tâches {T1, T2, ... Tm}, chacune comprend un ensemble de ET-processus, le nombre des processus dans chaque tâche est {L1, L2, ... Lm}; un ensemble de processeurs {P1, P2, ... Pn} et un vecteur représentant la charge initiale de chaque processeur {C1, C2, ... Cn}; on cherche une allocation des tâches dans les processeurs de sorte que le FIC soit minimal, avec la condition que les processus communicants dans une même tâche doivent être alloués dans des processeurs voisins.

IV.3.5.1 L'algorithme optimal

L'algorithme optimal consiste à parcourir tout l'arbre de recherche pour trouver la meilleure allocation. Par exemple, étant donné le nombre de processeurs n=8, le nombre de tâches m=3 avec {T1 : 5, T2 : 3, T3 : 1} et la charge initiale { 2, 1, 0, 4, 3, 1, 2, 2}, l'arbre de recherche est présenté dans la figure IV.8.

La complexité de l'algorithme optimal est $O(n^m)$, il est donc souhaitable de trouver un algorithme plus simple.

IV.3.5.2. Un algorithme heuristique

Nous avons développé un algorithme heuristique dans un but de simplicité et d'efficacité. Les heuristiques sont effectuées sur trois critères :

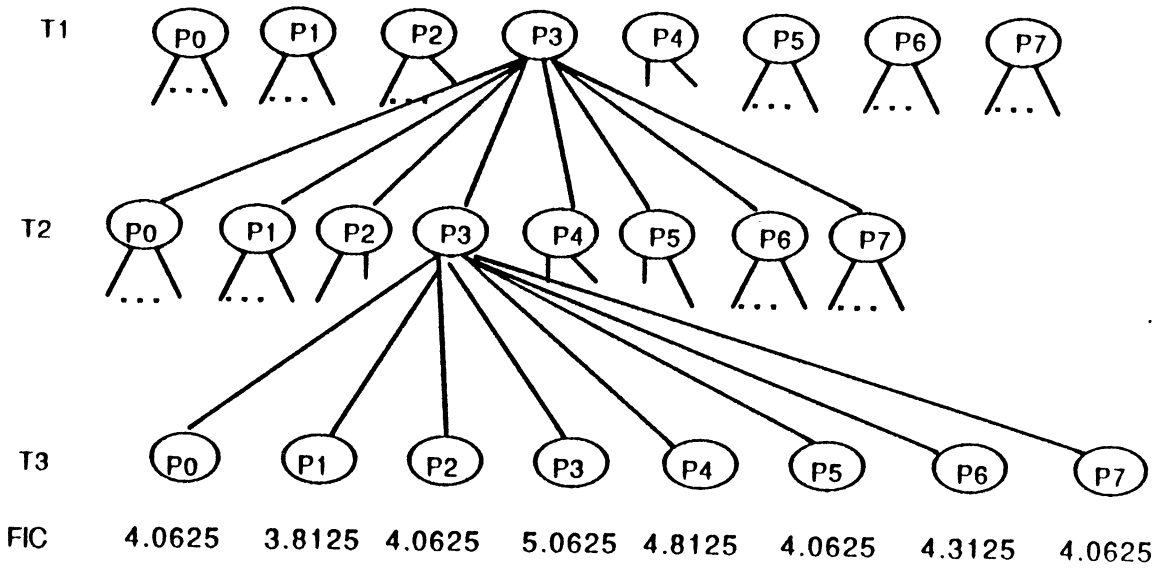

Fig. IV.8.

1) L'ordre d'allocation pour des tâches en attente : il est plus facile d'allouer une tâche qui a moins de processus. La tâche qui a le moins de processus va donc être considérée en premier.

2) La charge actuelle : L'allocation d'une tâche doit avoir un FIC minimum.

3) La charge potentielle : L'allocation d'une tâche doit considérer aussi les tâches suivantes en attente, pour avoir un FIC minimum après allocation de toutes les tâches.

Ces considérations peuvent être concrétisées par les règles suivantes:

- Pour le choix d'une tâche à allouer:

Règle 1: La tâche qui a le moins de processus sera servie en premier.

Règle 2: Si une tâche contient autant de processus que le nombre de processeurs, cette tâche sera allouée tout de suite sans considérer d'autres règles.

- Pour que la charge actuelle soit régulière:

Règle 3: Un processus doit être alloué au processeur le moins chargé.

Règle 4: En respectant la règle 3, essayer les différentes possibilités d'allocation. Choisir celle qui a le FIC minimum.

- Pour la charge potentielle:

Règle 5: S'il y a plusieurs processeurs qui ont la même charge et que cette charge est la charge minimum dans le système, on regarde les deux processeurs voisins de chaque processeur. Celui qui a un processeur voisin plus chargé sera choisi.

Exemple IV.5:

Soit un vecteur de charge {0,0,0,0,4,2}, le processeur P_4 sera choisi en premier. Dans le cas où on a deux tâches à allouer qui ont toutes les deux 2 processus, cette règle évite l'allocation de la premiere tâche aux processeurs P_2 et P_3, qui donne la charge finale {1,2,1,0,4,2} au lieu de l'allocation optimale {1,1,1,1,4,2}.

Règle 6: En respectant la règle 4, si plusieurs allocations donnent le même FIC et ce FIC représente le FIC minimum, on regarde les deux processeurs voisins de chaque allocation, celle qui a un processeur voisin le plus chargé sera choisie.

Exemple IV.6:

Soit un vecteur de charge {1,1,0,1,4,2} et une tâche avec 2 processus, on choisit le processeur P_3. Parmi les deux allocations possibles $\{P_2,P_3\}$ et $\{P_3,P_4\}$, on choisit la deuxième pour la même raison que celle de l'exemple IV.5.

L'algorithme IV.1 : Un algorithme heuristique pour allocation des processus :

Entrée :

Un vecteur de charge initiale $\{C_1, C_2, \ldots C_n\}$ pour les processeurs $\{P_1, P_2, \ldots P_n\}$;

un ensemble de tâches $\{T_1, T_2, \ldots T_m\}$;

le nombre de processus dans chaque tâche $\{L_1, L_2, \ldots L_m\}$;

les processus dans chaque tâche notés par $\{Proc_1, Proc_2, \ldots Proc_{Li}\}$.

Sortie :

Une allocation de tâches: chaque processus d'une tâche est alloué dans un des processeurs $\{P_1, P_2, \ldots P_n\}$, les processus dans une même tâche doivent être alloués dans les processeurs voisins.

Algorithme :

1) Choisir une tâche T_j qui a plus de processus, c'est à dire T_j avec

$$L_j = \min (L_1, L_2, \ldots L_j, \ldots L_m)$$

éliminer Tj dans les tâches en attente.

2) Choisir un processeur P_i avec min $(C_1, C_2, \ldots C_i \ldots C_n)$. S'il existe plusieurs processeurs qui ont la même charge, soit $(P_i, P_j, \ldots P_n)$, choisir le processeur qui correspond au

$$\max(\max(C_{i-1}, C_{i+1}), \max(C_{j-1}, C_{j+1}), \ldots \max(P_{n-1}, P_{n+1}))$$

3) Calculer un ensemble $\{FIC_1, \ldots FIC_{Lj}\}$ en supposant que $proc_1$ est alloué dans le processeur P_{i-Lj+1}, $P_{i-Lj+2} \cdots P_i$ respectivement et les autres processus de la tâche sont alloués dans les processeurs successifs.

4) Choisir l'ensemble $\min(FIC_1, FIC_2, \ldots FIC_{Lj})$, s'il n'existe qu'un élément dans l'ensemble, FIC_q, allouer le premier processus de la tâche $proc_1$ au processeur P_q et aller

en 1). Sinon aller en 5).

5) Soit l'ensemble $\{FIC_a, FIC_b, ...FIC_s\}$ où tous les éléments sont les mêmes et réprésentent le FIC minimum, cet ensemble correspond à l'allocation du premier processus de la tâche dans les processeurs $\{P_a, P_b, ...P_s\}$ respectivement. On regarde les deux processeurs voisins de chaque allocation proposée. Choisir celle qui correspond au

$$\max(\max(C_{a-1}, C_{a+Lj}), \max(C_{b-1}, C_{b+Lj}), \max(C_{s-1}, C_{s+Lj})).$$

Une expérimentation a été effectuée en C sur SM90 en vue de la mesure sur l'algorithme heuristique. La charge initiale et le nombre de processus dans chaque tâche sont générés aléatoirement. La table IV.1 donne les résultats de cette expérimentation. Dans la table on constate les améliorations de l'allocation par rapport à l'allocation optimale lorsqu'on applique de plus en plus de règle. Les résultats montrent que l'algorithme heuristique obtient des allocations très proche de celles de l'algorithme optimal.

Nombre de simulations	Nombre de processeurs	Nombre de tâches	charge initiale maximum	Nombre maximum de processus par tâche	pourcentage de l'allocation optimale			
					sans règle 5,6	sans règle 1	sans règle 6	avec toutes les règles
200	3	3	3	3	100%	100%	100%	100%
200	4	4	4	4	98%	99.5%	100%	100%
50	5	5	5	5	80%	88%	88%	92%
50	6	6	6	6	78%	86%	88%	88%

Table IV.1

La complexité de l'algorithme heuristique est O(m). Le temps de calcul est donné dans la Table IV.2 en comparaison avec l'algorithme optimal. Les chiffres sont obtenus par l'outil profile sur système UNIX.

Nombre de processeurs	Nombre de tâches	temps pris par l'algorithme optimal (milli-secondes)	temps pris par l'algorithme heuristique (milli-secondes)
3	3	3.80	3.90
4	4	30.01	8.90
5	5	398.07	11.20
6	6	5761.41	18.40

Table IV.2

CHAPITRE V

EXPERIMENTATION PAR SIMULATION

CHAPITRE V

EXPERIMENTATION PAR SIMULATION

Une expérimentation a été effectuée en vue de la validation du modèle défini dans le chapitre III. Nous décrirons dans ce chapitre l'expérimentation par simulation en langage parallèle OCCAM sur VAX/VMS. Le modèle de la simulation est légèrement simplifié. L'architecture et l'algorithme de dispatching sont simulés en faisant l'hyphothèse de multi-processeurs présentée au chapitre IV avec quelques modifications à cause des limitations du langage OCCAM. Le résultat de la simulation montre que les parallélismes exploités dans le modèle et l'architecture multi-processeurs accélèrent effectivement le traitement.

V.1 Généralités

Le but de la simulation est de valider le modèle défini dans le chapitre III. Pour simuler le fonctionnement d'une machine multi-processeurs, nous avons choisi un langage parallèle. Le langage OCCAM [Inm84] est issu de CSP [Hoa78] et développé par la société Inmos. OCCAM permet de spécifier un problème à l'aide de processus parallèles. La communication entre les processus parallèles est réalisée par envoi de messages à travers les canaux de communication déclarés pour ces processus.

Le langage OCCAM a été conçu comme l'assembleur du Transputer [Wil83]. Ce dernier est un micro-processeur supportant directement OCCAM et sert comme composant de base pour construire des machines hautement parallèles. L'utilisation du langage OCCAM pour la spécification de problèmes parallèles et la communication entre les processus parallèles est très commode. Mais de

par sa liaison directe avec le Transputer, OCCAM n'est pas un langage de très haut niveau. Il ne permet pas la récursivité, et peu de fonctions existent pour la manipulation des données structurées complexes. Ces caractéristiques d'OCCAM compliquent beaucoup la programmation et nous avons donc simplifié le modèle défini au chapitre III en ce qui concerne les aspects suivants:

- L'algorithme de traitement des clauses récursives n'est pas pris en compte.
- Les résultats produits par le R-processus ne sont pas envoyés directement au successeur de son ET-processus père. Ils sont retournés au ET-processus qui a créé le R-processus lui-même.
- Du fait que les résultats de recherche sont retournés au ET-processus, il n'est plus nécessaire de transférer l'environnement du ET-processus au R-processus. Ceci implique qu'un ET-processus correspondant à un prédicat défini par des clauses non-unaires, peut être désalloué en gardant un environnement contenant des variables libres pour attendre les résultats issus d'un processus fils, les arrivées des résultats des processus fils ne provoquent pas de nouvelle recherche mais une combinaison d'environnements.

Le système du langage OCCAM version 2 permet de manipuler simultanément huit fichiers à travers les canaux de communication spéciaux prédéfinis. Ceci est une limite dans un contexte base de connaissances parce que les connaissances sont stockées dans des fichiers et nous ne pouvons pas avoir plus de huit processus parallèles qui accédent aux connaissances.

En vue de la réalisation d'un premier prototype matériel sur la machine SM90 [Ber85] (cette dernière est une machine ayant au maximum huit processeurs de traitement), et compte tenu de la simulation de l'algorithme de dispatching au chapitre IV, l'architecture simulée est un peu modifiée. Le nombre des processeurs de traitement est limité au maximum à six et on suppose que la communication entre les processeurs de traitement est réalisée par un réseau qui permet toutes les connexions inter-processeurs possibles.

La figure V.1 présente un schéma conceptuel des processeurs dans la simulation. Un OU-processeur exécute tous les OU-processus. En effet, la création d'un OU-processus est

équivalente à un envoi de message au OU-processeur. L'arbre de processus est une structure dynamique partagée par le OU-processeur et le Scheduler-processeur. La création d'un ET-processus se fait à l'aide du OU-processeur en ajoutant un noeud dans l'arbre de processus.

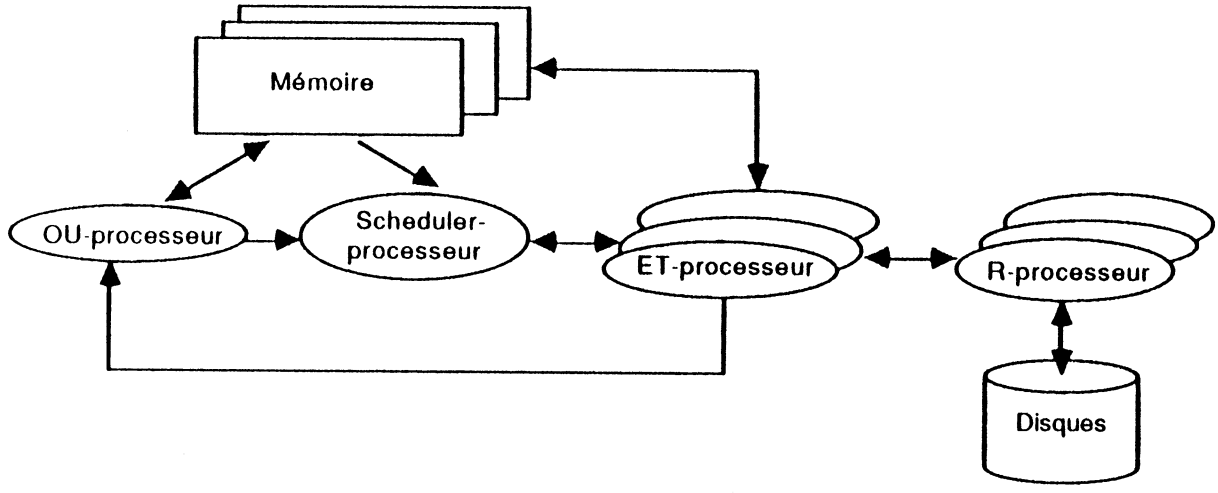

Fig. V.1

Les modules de mémoires ont pour rôle de stocker l'arbre de processus et les environnements à traiter par des ET-processus lorsque les ET-processeurs ne sont plus disponibles. L'évaluation d'un ET-processus est réalisée par un ET-processeur, il s'agit de transférer les paramètres et les environnements nécessaires au ET-processeur. Le ET-processeur construit des buts, crée des R-processus etc. On considère que les R-processeurs résident dans le même processeur physique que le ET-processeur. La création d'un R-processus correspond à un envoi d'un message au R-processeur. Le message contient les buts à vérifier. Les R-processus accèdent aux fichiers qui stockent les clauses sous forme interne.

Le Scheduler-processeur permet d'organiser le pipeline et de distribuer des ET-processus. Un ET-processus devient évaluable dans deux cas: soit il est le premier ET-processus qui correspond au premier prédicat de la queue d'une clause et il vient d'être créé par un OU-processus, soit il y a un message pour lui contenant des environnements à traiter. Le Scheduler reçoit un signal d'un OU-processus dans le premier cas, et d'un ET-processus dans le deuxième cas. Chaque fois que le Scheduler est informé qu'il y a un ET-processus qui devient évaluable, il analyse d'abord le

ET-processus évaluable, puis les ET-processus suivants. Tous les processus qui peuvent être évalués dans un pipeline sont écrits dans un tampon Pret_éval.

L'allocation de processus est faite en fonction de la longueur du pipeline et de la disponibilté des ET-processeurs. La multi-programmation des ET-processeurs n'est pas implémentée car le programme de simulation a déjà atteint la limite d'imbrication des procédures. Donc l'algorithme de dispatching adopté est une simple comparaisson de la longueur du pipeline et du nombre des ET-processeurs disponibles.

V.2 La représentation de l'arbre de processus et des données

Les fichiers sont utilisés pour la simulation des unités disques. Un fichier simule une piste du disque, ce qui correspond à une unité de filtrage. Les fichiers contiennent les clauses codées produites par un compilateur de clauses. Le compilateur transforme les clauses sous forme externe, textuelles en forme interne. Chaque symbole est codé par quatre octets dont le premier est le type (symbole, entier, caractère,...). Chaque fichier ne contient que des clauses ayant la même tête et le même nombre d'arguments. Les fichiers sont typés en trois catégories: Données, Règles et Mixte. Le premier type de fichiers ne contient que des clauses unaires. Le deuxième ne contient que des clauses non-unaires. Le troisième contient les deux types de clauses. Ces types seront utilisés pour la construction de pipeline et nous y reviendrons dans la section V.4 lorsque nous présenterons le Scheduler.

Comme nous avons indiqué au chapitre IV, le réseau de processus peut être réduit à un arbre nommé arbre de processus. Chaque noeud réprésente un ET-processus. Cet arbre est réprésenté par un tableau de processus et deux tableaux de données. Le tableau de processus contient les informations des noeuds, c'est à dire les ET-processus. Le nom du ET-processus correspond au numéro d'entrée dans le tableau de processus. La largeur du tableau de processus est de dix octets. Ce tableau contient les informations suivantes:

- Niveau dans l'arbre.
- Nombre de prédécesseurs.
- Nom du successeur.
- Etat actuel.
- Type.
- Pointeur sur le tableau de squelettes.
- Pointeur sur le tableau d'environnements.

On considére que l'arbre de processus est stocké dans une mémoire intelligente qui accepte la demande d'accès par contenu. Ceci est simulé par des fonctions qui retournent les résultats dont on a besoin. L'accès à l'arbre est effectué par le OU-processeur et le Scheduler-processeur, donc dans la simulation, un sémaphore est utilisé pour le maintien de la cohérence lorsqu'il y a des accès concurrents. Cet arbre peut être distribué dans plusieurs modules de mémoire si nécessaire. Ceci permet l'accès parallèle et l'augmentation du taux de parallélisme.

Deux tableaux de données sont utilisés. L'un est le tableau de squelettes pour le stockage du squelette du prédicat dans chaque ET-processus. Le début du squelette est pointé par un pointeur dans le tableau de processus. L'autre est le tableau des environnements à traiter par des ET-processus. Le dernier ET-processus dans un pipeline transfere les environnements dans le tableau d'environnements. Ces deux tableaux correspondent aux modules de mémoires primaires dans l'architecture orientée processus. Ils sont utilisés pour stocker les résultats intermédiaires.

La réprésentation d'environnements dans un système parallèle concerne essentiellement la réprésentation des variables et leurs substitutions. Dans la simulation, les substitutions sont copiées littéralement dans chaque processus.

Exemple V.1:

But: t(Y,machine,Z)

Clause: t(arbre,X,Y) -> t1(...) ...

Les substitutions générées pour le but sont [Y<-arbre, Z<-Y]; celle générée pour la clause est [X<-machine].

Dans le cas où une variable dans la clause est liée avec un terme contenant des variables libres, la technique de création des variables intermédiares [Mel82] est utilisée. Les variables intermédiaires sont créées dans l'environnement de clauses invoquées.

Exemple V.2:

But: t(m(X,Y))

Clause: t(X) -> t1(...) ...

La substitution générée pour la clause est [X<-m(A,B)] et pour le but [X<-A, Y<-B].

V.3 Les processus et les messages

Les codes des différents processus sont implémentés dans les processeurs correspondants. Le OU-processeur et le Scheduler-processeur sont relativement simples. Le ET-processeur doit arranger beaucoup de communications avec différents processeurs. Le R-processeur réalise l'unification entre un groupe de buts et un flux de clauses issu d'un fichier. L'algorithme du filtrage n'est pas implémenté dans cette expérimentation car ceci a fait l'objet d'un autre travail [Ian85] [Bel84]. La programmation de l'algorithme d'unification est compliquée car OCCAM ne permet pas la récursion.

La figure V.2 donne un schéma global de l'implémentation du système. Les tampons utilisés ont pour but d'envoyer les messages asynchrones entre les processus parallèles. Le multiplexage, et le contrôle d'accès concurrents aux tampons sont réalisés par le biais des sémaphores. Le

processeur aiguillage analyse les messages issus des ET-processeurs et les envoie au OU-processeur ou au Scheduler-processeur selon le contenu des messages.

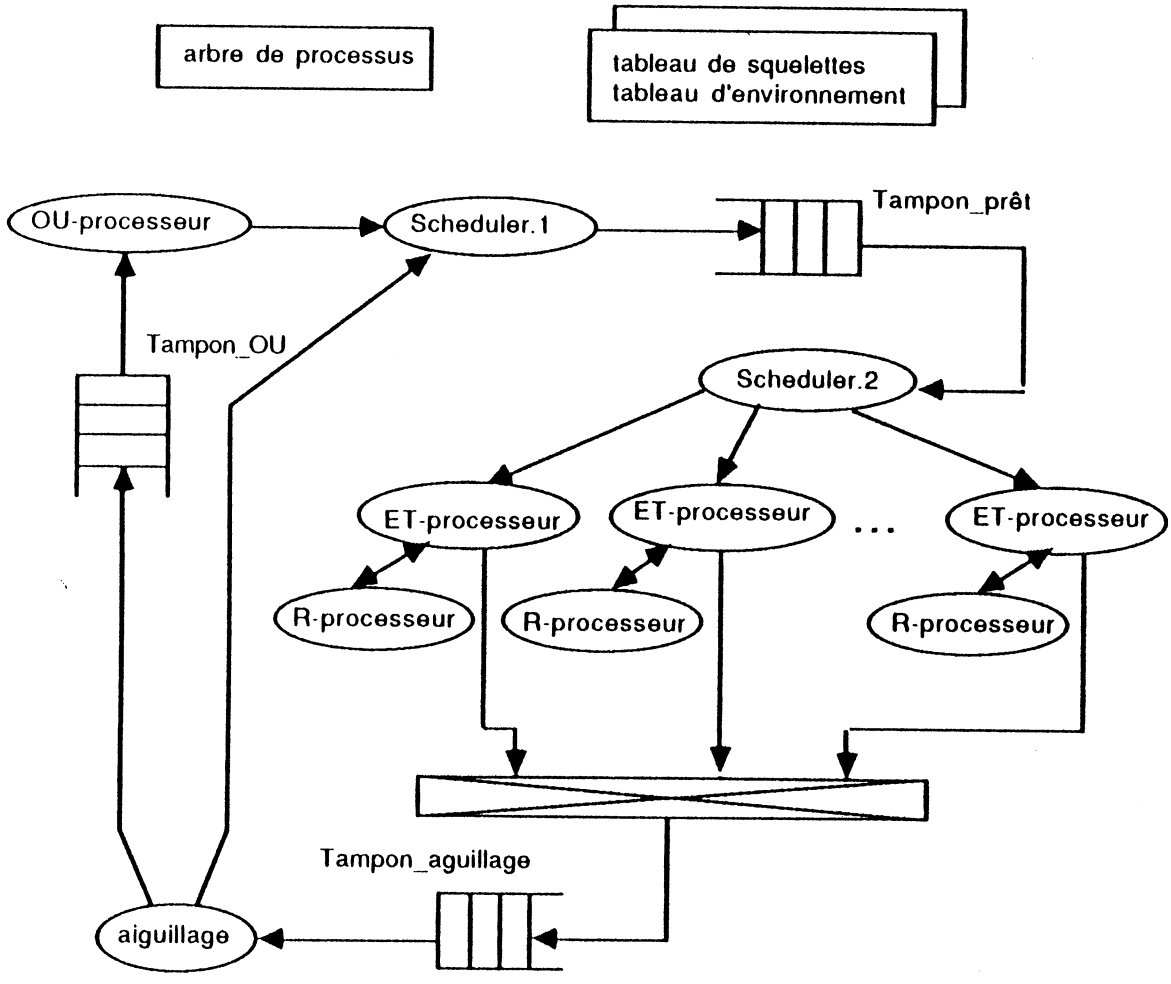

Fig. V.2

Les messages circulant dans le système ont différentes formes. Puisque les codes des différents processus sont implémentés dans différents processeurs, les messages entre ces processeurs réprésentent soit la création des processus (les messages au OU-processeur et au R-processeur), soit les environnements à traiter qui déclenchent l'évaluation d'un ET-processus (les messages du ET-processeur au ET-processeur ou au Scheduler-processeur). Nous décrivons ci-dessous ces messages entre différents processeurs.

Les messages du ET-processeur au R-processeur: Un message contenant des buts à vérifier correspond à la création d'un R-processus. Le deuxième message ci-dessous est un message de

contrôle nécessaire pour arrêter un R-processeur.

{No.but terme ::: No.but ::: FIN_PAQUET}

{No.ET-processus FIN}

Le message du R-processeur au ET-processeur: Ce sont les messages de réponse au ET-processeur. Si un but est vérifié, les substitutions générées sont retournées. Puisque l'unification se fait entre un groupe de buts et un ensemble de clauses, les substitutions pour un but sont précédées par le numéro du but. Si une clause non-unaire unifie un but, la clause entière est retournée avec les substitutions. Un message de contrôle est utilisé pour indiquer la fin de recherche pour un groupe de buts.

{No.but SUCCES No.but No.variable substitutions :::
:::
No.but ECHEC
:::
FIN_PAQUET
}

{FIN_RECH}

{No.but SUCCES
DELIMITEUR substitutions_pour_le_but
DELIMITEUR substitutions_pour_la_clause
DELIMITEUR la_clause
FIN_PAQUET
}

Le message du ET-processeur au OU-processeur: Un message du ET-processeur au

OU-processeur correspond à la création d'un OU-processus. Le OU-processeur traite les messages en écrivant les substitutions dans différents tableaux et en ajoutant les noeuds dans l'arbre de processus.

(No.Et-processus
DELIMITEUR substitutions_pour_le_but
DELIMITEUR substitutions_pour_la_clause
DELIMITEUR la_clause
FIN
)

Le message du ET-processeur au Scheduler ou au ET-processeur: Les substitutions retournées par le R-processeur sont combinées avec des environnements initiaux par le ET-processeur. Ces nouveaux environnements seront envoyés directement à un autre ET-processeur dans lequel le successeur du ET-processus est évalué en parallèle si c'est le cas. Sinon les environnements seront envoyés au Scheduler-processeur. Il est à noter que dans tous les cas, le ET-processeur envoie les messages concernant la création de sous processus au OU-processeur.

Le message SUCCES mais sans substitution signifie qu'il y a succès mais pas de substitution générée. Le message ECHEC informe que tous les buts dans le groupe ne sont pas vérifiés. Le message FIN indique la terminaison d'un ET-processus.

(No.ET-processus
environnement_1 DELIMITEUR
:::
environnement_n DELIMITEUR
FIN_PAQUET
)

(No.ET-processus SUCCES FIN_PAQUET)

(No.ET-processus ECHEC FIN)

(No.ET-processus FIN)

Le message du OU-processeur au Scheduler processeur: C'est un message pour signaler au Scheduler que le ET-processus en question devient évaluable.

(No.ET-processus FIN)

Le message du ET-processeur au Scheduler processeur: Le dernier ET-processeur dans une évaluation en pipeline écrit les environnements à traiter dans le tableau d'environnement. Un message est envoyé au Scheduler-processeur pour l'informer que le ET-processus successeur devient évaluable.

(No.ET-processus_producteur No.ET-processus_successeur FIN)

V.4 Le Scheduler

Le rôle du Scheduler est d'organiser le pipeline et de distribuer les ET-processus. Il est composé de deux modules: Scheduler.1 et Scheduler.2 qui réalisent ces deux fonctionnalités.

Le Scheduler.1 accepte les signaux envoyés par le OU-processeur ou un des ET-processeurs. Chaque fois que le Scheduler.1 est informé qu'un ET-processus devient évaluable, il analyse d'abord le ET-processus prêt. Si le message vient d'un niveau inférieur, ça veut dire que c'est un sous processus qui retourne les résultats et il s'agit de la combinaison d'environnements gardés et des substitutions reçues. Dans ce cas là, le successeur du ET-processus courant peut être évalué en pipeline avec le ET-processus prêt. Si le message est issu d'un ET-processus de même niveau, c'est un nouvel environnement et il s'agit d'une recherche dans la base de connaissances. Dans ce cas là, si le ET-processus est du type Données, c'est à dire que dans la base de connaissances, il n'existe

que des clauses unaires définissant le prédicat en question, le successeur peut être évalué en pipeline avec le ET-processus prêt. Si le ET-processus est d'un type Règle ou Mixte, c'est à dire qu'il existe des clauses non-unaires définissant le prédicat et on doit créer des sous processus, le pipeline s'arrête là et les successeurs ne seront pas évalués avec le ET-processus prêt.

Le même principe s'applique à chaque ET-processus successeur pour déterminer les ET-processus qui peuvent être amenés dans un pipeline. Le nombre de processus dans un pipeline doit être inférieur au nombre de ET-processeurs dans le système à cause de l'algorithme de dispatching qu'on emploie (voir ci-dessous). Les ET-processus constituant un pipeline constituent une tâche. Les tâches sont écrites dans un tampon nommé "tampon_prêt", qui est une file d'attente. Ce tampon est une interface entre le Scheduler.1 et le Scheduler.2.

Le Scheduler.2 alloue les ET-processus aux ET-processeurs. Le tampon_prêt est organisé d'une manière FIFO (First In First Out). L'algorithme de dispatching est une simple comparaison du nombre de ET-processus dans le pipeline et du nombre de ET-processeurs disponibles. Donc une tâche est allouée s'il y a assez de ET-processeurs. Dans le cas où le nombre de processus de chaque tâche dans le tampon_prêt est supérieur au nombre de ET-processeurs disponibles, le Scheduler.2 choisit aléatoirement une tâche et alloue les premiers processus dans le pipeline aux ET-processeurs disponibles. Les restes des processus dans la tâche sont abandonnés et seront reconsidérés par le Scheduler.1 lorsque ses prédécesseurs produiront les données.

V.5 Les résultats de la simulation

On effectue la simulation dans l'hypothèse suivante: le nombre de ET-processeurs varie de deux à six, il y a un OU-processeur et un Scheduler-processeur. Par manque d'outil exact de mesure du temps d'exécution dans chaque processus, nous supposons que le temps d'exécution d'un ET-processeus est d'une unité. Il est à noter que cette façon de calcul du temps d'exécution donne un résultat indicatif, car dans une machine séquentielle (VAX-780 pour notre cas), le système OCCAM donne le contrôle d'un processus à un autre processus parallèle suivant trois conditions:

- il y a une communication entre le processus en exécution et un autre processus parallèle;
- le processus en exécution termine;
- le processus en exécution atteint le "time out".

Ceci veut dire que dans le cas où il y a plusieurs ET-processus qui se déroulent, leur chance d'avoir le contrôle du processeur est inégale. Nous prenons le ET-processus qui dure le moins longtemps comme unité.

L'exemple ci-dessous présente des benchmarks pour la simulation.

Exemple V.3:

```
personne(nom_personne, age, sexe).    23 clauses.
père(nom_père, nom_enfent).           11 clauses.
mère(nom_mère, nom_enfent).           12 clauses.

parent(X,Y) :- père(X,Y).
parent(X,Y) :- mère(X,Y).

soeur(X,Y) :- parent(Z,X), parent(Z,Y), personne(Y,_,feminin).
frère(X,Y) :- parent(Z,X), parent(Z,Y), personne(Y,_,masculin).
couple(X,Y) :- père(X,Z), mère(Y,Z).
famille(X,Y) :- couple(X,Y).
famille(X,Y) :- soeur(X,Y).
famille(X,Y) :- frère(X,Y).
famille(X,Y) :- parent(X,Y).
famille(X,Y) :- parent(Y,X).

Requête1: soeur(X,Y), imprime(X,Y).
Requête2: famille(jean_pierre,Z), imprime(Z).
```

Lors de la simulation, pour la requête1, 52 ET-processus et 96 R-processus ont été créés, et 17 solutions ont été retournées. Pour la requête2, 72 ET-processus et 122 R-processus ont été créés, et 10 solutions ont été retournées. Le temps d'exécution et l'espace du travail (tableau d'environnements) sont donnés dans la table V.1.

Le résultat de la simulation montre l'amélioration du temps d'exécution par rapport au nombre de ET-processeurs (Fig.3). Le taux d'échec des ET-processus est grand. Cet inconvénient peut être compensé par l'économie de l'espace de travail, parce que l'évaluation en pipeline évite de temporiser des données, puis de les copier pour l'évaluation des ET-processus successeurs.

Nombre de ET-processeurs	Requête1		Requête2	
	Temps d'exécution	Espace de travail (K octets)	Temps d'exécution	Espace de travail (K octets)
2	133	5.67	183	5.65
3	96	4.46	143	4.67
4	87	3.60	117	3.98
5	74	3.66	95	3.68
6	62	2.98	90	3.28

Table V.1

V.6 Conclusion

L'expérimentation effectuée montre que le taux du parallèlisme exploité dans le modèle est grand. L'évaluation parallèle de programmes Prolog dans un contexte de base de connaissances à grand volume de données en machine multi-processeurs accélère effectivement le calcul. La vitesse du calcul est améliorée lorsqu'il y a de plus en plus de ET-processeurs dans le système. La décomposition de l'évaluation en processus et l'évaluation en pipeline des processus est une

méthode efficace dans le contexte base de connaissance.

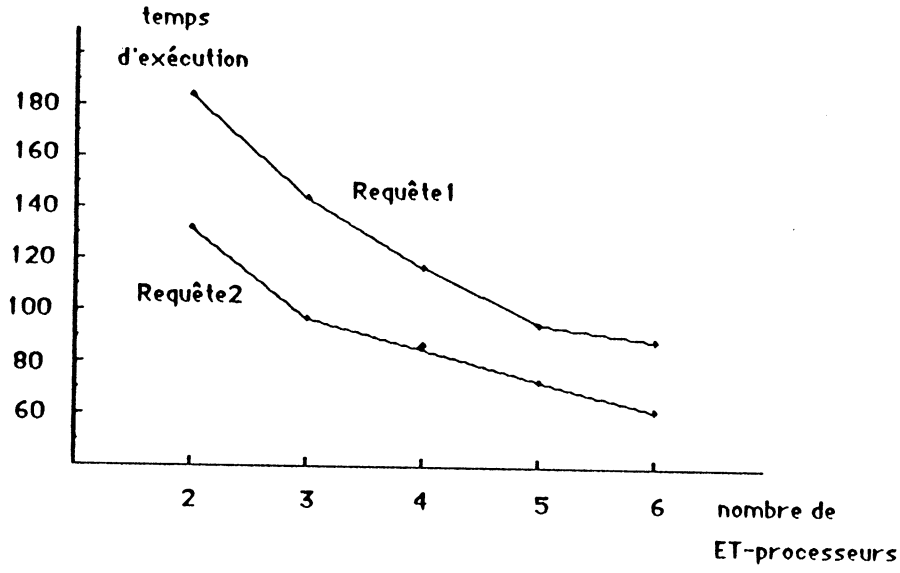

Fig.V.3

Le résultat de la simulation montre que dans un environnement parallèle, le compromis entre le nombre de processeurs et l'espace du travail est le problème principal de la conception de l'algorithme de dispatching.

CHAPITRE VI

CONTRIBUTION AU LANGAGE PROLOG

CHAPITRE VI

CONTRIBUTION AU LANGAGE PROLOG

Les idées principales d'OPALE, à savoir le filtrage des clauses et le parallélisme, influencent aussi le langage Prolog lui même. Dans ce chapitre, nous présentons un système Prolog enrichi par des notions d'OPALE qui permettent de manipuler les fichiers externes contenant les clauses unaires, de changer la stratégie de recherche pour avoir un accès efficace aux fichiers externes et d'utiliser le principe du filtrage en tant que mécanisme de base de l'accès. Ce système a été conçu et implémenté en langage C/UNIX sur SM90 en collaboration avec Gillles Berger Sabbatel et Jean Christophe Ianeselli.

VI.1 Introduction

Le langage Prolog a eu beaucoup de succès dans différentes branches de l'intelligence artificielle et d'autres domaines informatiques. L'importance croissante des applications en Prolog, surtout en système expert, implique des manipulations sur des volumes de données de plus en plus grands. Un interpréteur Prolog classique suppose que sa base de connaissances réside dans la mémoire centrale et cela limite sa puissance de résolution.

Trois approches sont envisagées pour résoudre ce problème:

- Interfacer Prolog avec une base de données relationnelle [Cha81].
- Le développement de la base de connaissances reposant sur Prolog. C'est l'approche d'OPALE.

- L'enrichissement de l'interpréteur Prolog en le munissant d'une capacité de manipulation des clauses à l'extérieur de la mémoire centrale.

La première approche se base sur l'idée de la coopération Prolog-base de données relationnelle. Or, les données manipulées dans la base de données relationnelle sont totalement sous une forme classique (type, structure etc), tandis que Prolog manipule essentiellement des listes et des arborescences. Donc une coopération Prolog-base de données relationnelle ne satisfait pas parfaitement les besoins de Prolog. La deuxième approche est la plus séduisante, mais de par la quantité du travail, on ne peut pas espérer un produit à court terme. De plus, le coût de la réalisation d'un tel système est très grand.

Les travaux présentés dans ce chapitre reposent sur la troisième approche qui peut être considérée comme une première étape vers la machine base de connaissances Prolog, et aussi un système Prolog autonome.

Actuellement, peu de systèmes Prolog ont fourni le moyen de manipulation des clauses extérieures. A notre connaissance, seul D-Prolog [Don84] comporte un mécanisme un peu plus sophistiqué qu'un simple ajout. Un système permettant la manipulation de clauses extérieures doit résoudre les problèmes de l'organisation du dictionnaire global commun à l'interpréteur et à des clauses extérieures, de la compilation et l'organisation des fichiers contenant les clauses, et du mécanisme d'accès efficace aux clauses extérieures. Nous expliquons ci-dessous notre solution et présentons l'implémentation.

VI.2 L'organisation du système

Le système est constitué de quatre parties principales: l'interpréteur YAAP, un compilateur de clauses extérieures, les fichiers contenant les clauses unaires et un dictionnaire global. La figure VI.1 donne une vue globale du système. L'idée principale de la conception et de la réalisation est de construire un système modulaire et souple, chaque module étant indépendant. Nous présentons

ci-dessous chaque partie.

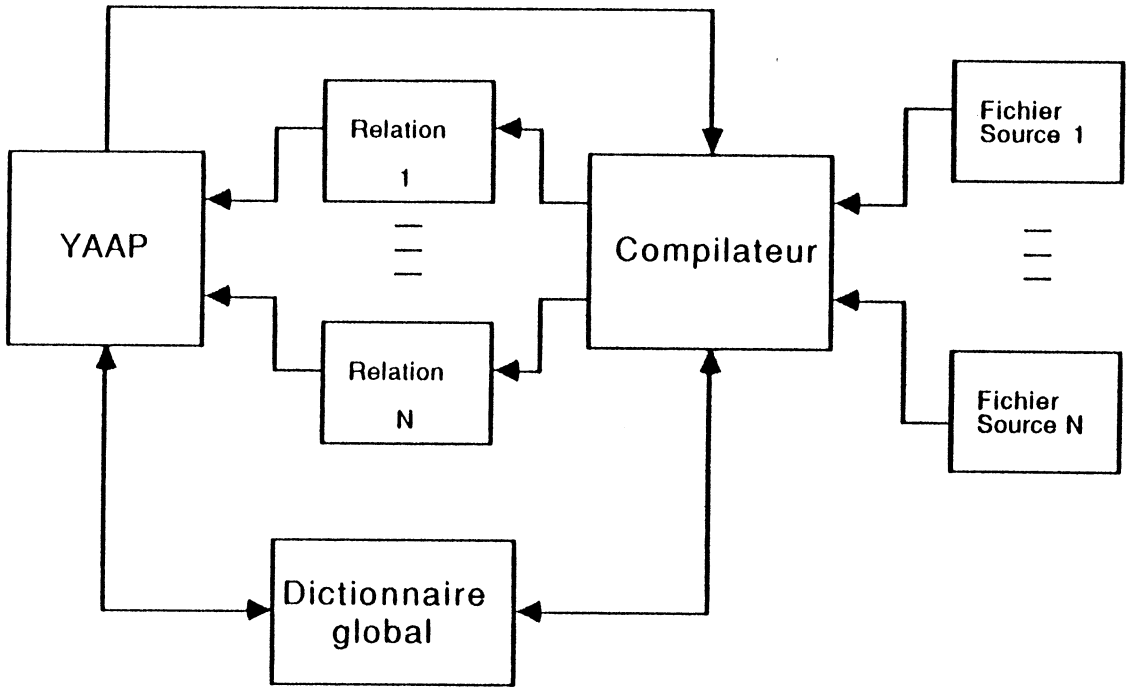

Fig.VI.1

VI.2.1 Le dictionnaire

Le dictionnaire global conserve les noms externes de tous les symboles du système. La compilation des clauses extérieures par le compilateur et la résolution des problèmes par l'interpréteur YAAP utilisent en commun ce dictionnaire. Ceci permet d'avoir un codage uniforme et unique pour tous les symboles apparaissant dans le système. Le dictionnaire est stocké dans un fichier et a une capacité de codage de 16K symboles différents. Cette taille est assez grande, de sorte qu'il ne peut pas être chargé entièrement ni dans YAAP ni dans le compilateur. Donc la technique de pagination est adoptée pour YAAP et aussi pour le compilateur. Le dictionnaire global est divisé en 256 pages, chacune pouvant contenir 64 symboles.

Le compilateur charge une page du dictionnaire global à la fois. YAAP a son propre dictionnaire nommé dictionnaire central qui charge seulement les symboles demandés, c'est le cas

pour l'impression d'un symbole: le nom externe du symbole doit être affiché. Le dictionnaire central contient aussi des symboles qui correspondent aux têtes des clauses dans la mémoire primaire (Table de clauses, voir VI.2.3). Un pointeur pour chaque symbole de ce type est conservé dans le dictionnaire central, ce pointeur pointe sur la première clause du paquet qui permet de trouver les clauses permettant de résoudre un but.

L'algorithme "charger à la demande" s'applique ici pour le chargement du dictionnaire central. Quand le nom externe d'un symbole est demandé et qu'il n'est pas dans le dictionnaire central, le numéro de la page du dictionnaire global contenant ce symbole peut être calculé à partir du code (nom interne) du symbole. La page demandée sera chargée à ce moment, et la page précédente sera réécrite dans le dictionnaire global. Une gestion du dictionnaire plus efficace pourra être implémentée ultérieurement (par exemple l'algorithme LRU). L'implémentation actuelle a été voulue simple pour en accélérer la réalisation.

VI.2.2 Le compilateur et les clauses extérieures

Les clauses unaires ayant la même tête et même nombre d'arguments peuvent être regroupées et stockées dans un fichier nommé fichier source. Les fichiers source sont compilés par le compilateur. Le compilateur produit des fichiers accessibles par l'interpréteur YAAP. La compilation des clauses est une simple traduction d'une forme externe, textuelle, à une forme interne. Les fichiers compilés sont appelés relations de base, nom issu des bases de données relationnelles. La compilation des clauses peut être invoquée à partir de YAAP. Mais les manipulations complexes, comme insertion, modification et suppression des clauses se font par éditeur de texte au niveau de source. L'implémentation de ces fonctions dans le système n'a pas été réalisée pour des raisons de temps. Nous nous concentrons donc plutôt sur le problème du mécanisme d'accès, au lieu de créer un système complet de gestion de base de données.

Les clauses non-unaires et les requêtes portant sur le système peuvent être regroupées dans un autre fichier et chargées lors de l'initialisation du système.

VI.2.3 L'interpréteur YAAP

YAAP (Yet Another Avatar of Prolog) est un interpréteur classique enrichi par certaines fonctionnalités nouvelles. La technique de partage de structure[BoM72] est utilisée dans YAAP. Comme tous les interpréteurs Prolog, il est composé essentiellement de deux zones: une statique et une dynamique.

La zone statique consiste en deux parties qui sont:

- La table des clauses qui contient les clauses sous forme interne. Les clauses ayant la même tête et le même nombre d'arguments sont chaînées par les pointeurs. La première est pointée par le dictionnaire central.

- Le dictionnaire central qui contient une page du dictionnaire global. En plus, il contient aussi des informations qui permettent de trouver la première clause lors de la résolution d'un problème.

La zone dynamique consiste en quatre parties nécessaires à la résolution de problèmes, les espaces sont gérés en pile:

- Pile de résolution: Cette pile mémorise le contexte de résolution. Elle contient cinq pointeurs qui sont:
  * But actuel, qui désigne le but qui est en train d'être résolu.
  * Alternative qui pointe sur la clause dont la tête est unifiée avec le but actuel.
  * Retour qui pointe sur la continuation de résolution, ceci permet de retrouver les buts laissés en attente lorsque les buts d'une clause sont tous vérifiés.
  * Instance qui pointe sur une entrée de la pile des instances où les substitutions générées au cours de la résolution du but actuel sont gardées.

* Trace qui pointe sur une entrée de la pile trace.

Chaque fois que l'unification est réussie, on empile le premier prédicat de la clause comme but actuel. Lorsque tous les buts dans une clause sont vérifiés, on dépile jusqu'à l'entrée pointée par le pointeur Retour. La résolution sera réussie si la pile de résolution devient vide, et elle aura échoué si le pointeur Alternative devient vide et l'unification n'est pas réussie.

- Pile des instances: Cette pile conserve toutes les substitutions générées au cours de résolution. Les places sont allouées avant l'unification entre un but et la tête d'une clause. A chaque variable de la clause invoquée est allouée une entrée. La technique de "structure partagée" est adoptée par YAAP et donc chaque entrée de la pile contient un double pointeur, l'un sur le squelette du terme unifié avec la variable et l'autre sur l'environnement correspondant.

- Pile de trace: Cette pile contient un ensemble de pointeurs sur la pile des instances. Les substitutions non-locales sont notées dans cette pile. Lors de retour arrière, les entrées de la pile des instances sont remises à nul.

- Pile de solutions: Cette pile sert comme un tampon qui conserve les solutions produites lors d'accès aux fichiers extérieurs.

VI.3 Les mécanismes d'accès aux clauses extérieures

Les mécanismes d'accès aux clauses extérieures implémentés dans les interpréteurs actuels sont souvent un chargement des clauses contenues dans un fichier dans la table des clauses de l'interpréteur. Cette approche est naturelle et simple, mais les clauses chargées sont limitées par la taille de la table des clauses. Notre approche consiste à amener seulement les informations utiles au lieu de charger toutes les clauses dans un fichier. Ceci est faisable par l'intermédiaire d'un filtre logiciel de données. Le principe du filtrage a été présenté dans III.1.2. Cette section présente son

utilisation à travers les prédicats évaluables, et son intégration dans l'interpréteur YAAP.

VI.3.1 Le prédicat interobase

Comme nous avons indiqué auparavant, YAAP est un interpréteur classique et il opère sur des données en mémoire centrale. Son utilisation avec des fichiers extérieurs pose des problèmes sous deux aspects:

(1) l'accès aux fichiers extérieurs qui sont en dehors de l'espace du travail.

(2) le traitement ensembliste des données retournées par chaque accès.

L'accès aux relations de base exige une stratégie de recherche "en largeur d'abord", mais YAAP explore l'arbre de recherche avec une stratégie de recherche "en profondeur d'abord". Une expression de l'accès implique un changement de stratégie. Deux méthodes sont envisageables, le changement implicite et le changement explicite. Avec la première méthode, un prédicat portant sur une relation de base est exprimé comme tous les autres prédicats dans le système, la reconnaissance du prédicat, le changement de la stratégie et le déclenchement de l'accès étant du ressort de l'interpréteur. Avec la deuxième méthode, un prédicat concernant une relation de base doit être explicitement spécifié dans le programme par l'utilisateur.

Nous avons adopté la deuxième méthode en raison de sa simplicité au niveau de l'implémentation. Un accès à une relation de base doit être spécifié par un prédicat évaluable interobase.

Forme:

interobase(V1.V2..., nom_relation(V1,V2,...))

Le premier argument est une liste des variables intéressées, le deuxième argument est le nom d'une relation de base stockée dans un fichier. Les variables du premier argument doivent apparaître dans le deuxième argument.

Cette forme de spécification demande très peu de modification de l'interpréteur YAAP. La spécification des variables intéressées nous permet d'amener seulement les informations utiles dans la mémoire primaire. On peut considérer que c'est une optimisation de l'accès, manipulée par les utilisateurs, similaire à une projection dans les bases de données relationnelles.

Les résultats de l'accès, c'est à dire les substitutions générées pour les variables spécifiées, sont empilés dans la pile de solutions. Chaque solution sera dépilée quand il y a retour arrière sur interobase. C'est à dire que le retour arrière sur interobase ne provoque jamais un nouvel accès à la relation de base mais seulement une copie du sommet de la pile. Cette méthode nous permet d'établir une interface entre le traitement ensembliste exigé par les relations de base et le traitement séquentiel de l'interpréteur YAAP.

VI.3.2 Le prédicat pipe

Une requête portant sur plusieurs relations de base peut être réalisée avec plusieurs prédicats interobase. Par exemple, une jointure sur la sélection de r1 et r2 dans la base de données relationnelle peut être exprimée comme suit:

interobase(X.Z,r1(a,X,Z)) interobase(X.Y,r2(X,Y,Z))

Dans ce cas là, interobase n'est pas très commode à utiliser lorsque il y a plusieurs relations de base concernées. D'abord on doit répéter l'écriture d'interobase pour chaque prédicat; deuxièmement, la capacité du filtre n'est pas pleinement utilisée. Dans l'exemple ci-dessus, supposons que le premier interobase retourne M solutions, la terminaison du premier interobase donne le contrôle à l'interpréteur YAAP en empilant M solutions, l'interpréteur YAAP copie seulement une solution dans son espace de travail, donc le deuxième interobase obtient chaque fois une solution à filtrer, et ceci nécessite M accès au fichier lié au deuxième interobase. On constate que la capacité de traitement de plusieurs solutions à la fois par le filtre n'est pas exploitée ici, ce qui

entraine que le nombre d'accès dans une suite de prédicats interobase successifs croît de façon exponentielle.

Pour résoudre ce problème, nous avons introduit le deuxième prédicat évaluable nommé pipe. La définition du prédicat évaluable pipe a pour effet de faciliter l'écriture et rendre plus efficace le traitement portant sur plusieurs relations de base.

Forme:

pipe(V1.V2...,nom_clause(V1,V2,...))

Le premier argument est de même forme que celui d'interobase, le deuxième spécifie une clause non-unaire de la forme:

```
nom_clause(V1,V2,...):-relation1(V1,...)
                        :::
                        relationk(Vn,...).
```

où les prédicats de la queue de la clause doivent être les relations de base stockées dans des fichiers.

L'évaluation du prédicat pipe provoque d'abord une analyse de la clause spécifiée par le deuxième argument, il s'agit d'établir un descripteur pour chaque prédicat. Puis l'interpréteur lance un ensemble de processus parallèle, chacun correspondant à un prédicat dans la queue de la clause. Ces processus se déroulent en pipe-line, les solutions produites par le premier prédicat seront transmises au prédicat qui suit. Le dernier prédicat retourne les solutions dans la pile de solutions et donne le contrôle à l'interpréteur YAAP.

VI.3.3 L'implémentation

L'accès aux relations de base est réalisé par le biais du module filtre. Le filtre peut être

décomposé en trois sous-modules: compilation, filtrage et association. Le module compilation reçoit un descripteur du prédicat et un ensemble des substitutions générées par le filtrage précédent. Il construit les buts à vérifier, et génère les codes internes du filtre. Le module filtrage déclenche la recherche dans la relation de base correspondante et réalise la pré-unification. Le module association complète ensuite l'unification et retourne les substitutions dans le tampon de solutions (Fig. VI.2).

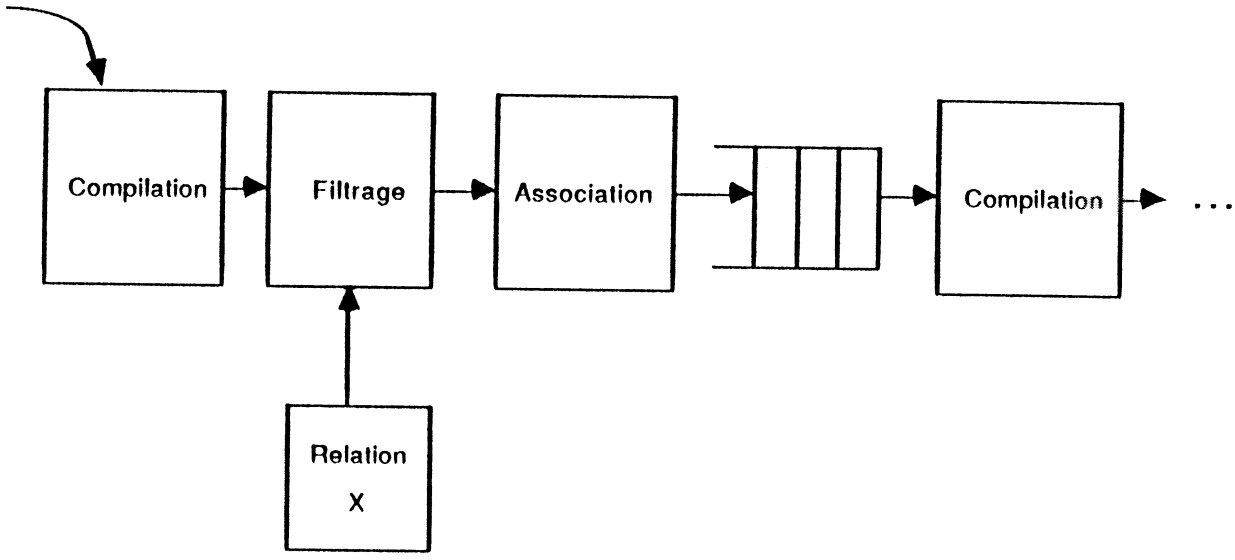

Fig.VI.2

L'algorithme du prédicat évaluable interobase est assez simple. Le déclenchement de la recherche dans le prédicat pipe est précédé par une analyse de la clause spécifiée. Il s'agit de typer chaque variable de la clause en suivant l'optimisation décrite dans III.2.4.. Les descriptions des variables dans la clause sont notées dans un descripteur pour chaque module du filtre. Nous donnons ici l'algorithme du prédicat pipe. L'algorithme de filtrage peut être trouvé dans [Ian85] [Bel84].

L'algorithme VI.1: L'algorithme du prédicat évaluable pipe.

BEGIN

noter les variables spécifiées dans le tampon.0;

```
SI (unifie(terme spécifié, tête de la clause) == SUCCES)
  (*unification entre le deuxième argument du pipe et la tête de la clause dans le programme *)
ALORS BEGIN pile.alternative := -100;          (* <0 *)
      copier la queue de la clause avec les substitutions des variables dans tampon.1;
      analyser le type de chaque variable;    (* V.P, V.S,...*)
      établir le descripteur pour chaque prédicat de la queue;
      créer un processus de filtrage pour chaque prédicat de la queue;
      SI dernier processus retourne ECHEC
      ALORS (* pas de solution *)
        BEGIN pile.alternative := 0 ;
              retourne(ECHEC);
        END
      SINON  (* recherche a trouvé des solutions *)
        BEGIN copier les solutions reçues dans le tampon de solutions;
              copier une solution dans pile.instance;
              retourne(SUCCES);
        END;
   END;
SINON retourne(ECHEC);
END.
```

VI.4 L'algorithme de résolution

L'algorithme de résolution a été modifié pour s'adapter aux deux prédicats évaluables "interobase" et "pipe". Nous donnons la partie de retour arrière de l'algorithme qui concerne cette modification. L'instruction "break" dans l'algorithme a pour effet de sortir la boucle TANTQUE.

L'algorithme VI.2: L'algorithme de retour arrière dans la résolution.

```
TANTQUE pile <> fin          (* existe encore des buts *)
 SI   pile.alternative == 0     (* alternative inexistant *)
 ALORS  dépiler la pile de résolution
 SINON  SI pile.alternative > 0
          ALORS   (* un pointeur, alternatives existent *)
            BEGIN traitement normal...;
                  break;
            END
         SINON    (* pile.alternative < 0, prédicat interobase ou pipe *)
            BEGIN copier une solution;
                  SI cette solution est la dernière
                  ALORS pile.alternative := 0;
                  break;
           END;
```

VI.5 Les mesures et les applications

Un générateur de clauses a été développé pour le test du système. Le générateur peut générer des relations de base aléatoirement en fonction des paramètres fournis par l'utilisateur. On peut contrôler la complexité et la sélectivité des clauses pour expérimentation et statistique.

Les mesures ont été effectuées sur le système implémenté. Les mesures sur le filtre ont été effectuées, et les détails peuvent être trouvés dans [Ian85] [Bel84]. L'interpréteur YAAP tourne à 1000 LIPS (Inférence Logique Par Seconde) sur MC 68000 (SM90) en utilisant les benchmarks de Warren [War77]. L'interpréteur YAAP tourne plus vite que C-Prolog sur la même machine, ce qui est dû au fait que C-Prolog fournit un environnement de programmation plus complet que YAAP ; que C-Prolog distingue les variables locales et globales, ce que YAAP ne fait pas, et que YAAP a été écrit et optimisé sur M68000 et que C-Prolog l'a été sur VAX. A titre indicatif, une comparaison de performance des différents systèmes Prolog pour la concaténation déterministe et

les benchmarks de Warren est donnée dans l'annexe.

Les mesures du prédicat interobase montrent que, avec la version logicielle du filtre, l'accès aux relations de base par le filtrage est de trois fois à cinq fois plus rapide qu'un transfert des clauses dans la mémoire centrale, plus l'unification classique par interpréteur.

Les mesures sur le prédicat évaluable pipe sont effectuées par des opérations correspondant à la jointure entre plusieurs relations dans la base de données relationnelle. La mesure en LIPS pour ce type d'opération n'est pas significative, parce que dans ce cas là, le but est de sélectionner les clauses et la plupart des tentatives d'unification ont échoué (plus de 90%). Nous utilisons le nombre d'unification Par Seconde (UPS) pour la comparaison entre différents systèmes (Table VI.1).

Un appel de pipe concernant l'accès aux trois relations de base donne une performance correspondant à 14,200 UPS, y compris le temps de transfert des résultats à l'interpréteur. Le même type d'opération est testé à l'intérieur des interpréteurs YAAP et C-Prolog sans accéder au fichier (résultats dans la table VI.2). Les résultats montrent que l'utilisation du prédicat pipe accélère beaucoup la résolution concernant plusieurs accès aux relations de base.

Comparaison de performance sur la jointure simple Sur la machine SM90	
Système	performance
C-Prolog	3,340 UPS
YAAP interpréteur	4,770 UPS
YAAP avec prédicat pipe	14,200 UPS *

* y compris le temps de l'accès aux fichiers et du transfert des résultats à l'interpréteur.

Table VI.1

L'inconvénient de notre système est la lenteur de compilation de programme, car la compilation doit accéder au dictionnaire global qui se situe dans un fichier. Ceci peut être amélioré par une stratégie de gestion du dictionnaire plus sophistiquée.

Un système de gestion de la bibliothèque de notre équipe a été installé. Ce système est structuré en six relations de base et une trentaine de requêtes spécifiques à l'interrogation de ces relations. Le système contient actuellement six cents ouvrages et articles de conférence. Les utilisateurs peuvent obtenir une bibliographie à partir d'un ensemble de mots clés.

L'utilisation du langage Prolog pour la recherche de l'information présente un avantage au niveau de l'apprentissage. Les notions de la logique sont plus simples et plus faciles à apprendre pour des utilisateurs que celles de l'algèbre relationnelle. Cette expérience nous a encouragé pendant le développement des applications d'OPALE.

VI.6 Conclusion

Le système Prolog que nous avons présenté montre qu'un interpréteur enrichi d'un filtre pour accéder aux fichiers présente une performance prometteuse. Les prédicats "interobase" et "pipe", comme moyen de changement de stratégie de recherche, fournissent une interface efficace pour le traitement ensembliste dans un environnement Prolog. Grâce à la modularité de la réalisation de ces mécanismes, ils peuvent être facilement intégrés dans un interpréteur Prolog classique, pour des applications orientées vers l'accès aux grands volumes de données.

Le prédicat "pipe" pourrait aussi être implémenté dans un environnement parallèle, par exemple dans un réseau local, pour accéder aux clauses stockées dans différents sites d'un système distribué. Le mécanisme de filtrage pourrait également être intégré à un interpréteur Prolog pour sélectionner les clauses en mémoire secondaire, puis les amener en mémoire primaire.

L'expérience de la conception et la réalisation du système présenté nous permettent d'envisager

la construction d'une version logicielle d'OPALE pour la gestion de base de connaissances intégrée à un système Prolog plus complet.

CONCLUSION

CONCLUSION

Cette thèse a proposé un modèle d'interprétation parallèle de Prolog dans le contexte base de connaissances. Le modèle développé montre que Prolog, non seulement peut être considéré comme un formalisme de résolution et le point de départ pour une machine à inférence, mais s'adapte aussi aux systèmes de gestion de connaissances.

L'utilisation de la notion de processus présente une approche prometteuse pour la formalisation du modèle des systèmes parallèles complexes, et pour présenter la correspondance entre un modèle et l'architecture de la machine multi-processeurs dans laquelle le modèle doit être traité. Cette approche facilite aussi l'organisation du traitement du modèle parallèle dans une machine pour avoir des performances satisfaisantes.

L'architecture supportant le modèle défini et le traitement dans un environnement multi-processeurs a été discuté. Au niveau de l'architecture, la notion d'architecture orientée processus a été introduite. La propriété dynamique de ce type d'architecture s'adapte essentiellement aux systèmes en intélligence artificielle. Ces derniers sont souvent complexes et difficiles à décomposer en opérations directement exécutables par les matériels.

L'évaluation des requêtes en pipeline dans la base de données répartie a montré l'efficacité de cette approche [Lu86] [SIS85]. Nous avons montré que l'évaluation du langage Prolog, en tant que mécanisme de manipulation pour extraire des connaissances, peut aussi s'organiser en pipeline avec un traitement performant.

Dans le contexte de la machine à architecture orientée processus, la distribution des processus

est un problème majeur. L'algorithme optimal, qui cherche la distribution optimale dans le sens de la régularisation de charge, ne peut pratiquement pas être utilisé à cause de sa complexité exponentielle. L'algorithme heuristique, qui alloue des processus en fonction de critères empiriques, peut avoir un overhead minimum et une distribution très proche de celle de l'algorithme optimal. L'allocation dynamique de processus doit essentiellement reposer sur cette approche.

Un interpréteur Prolog, enrichi d'un filtre avec les prédicats évaluables "interobase" et "pipe", permet d'avoir un système Prolog avec la puissance de la manipulation de clauses extérieures et donc d'augmenter la capacité de résolution. L'expérience sur un tel système nous permet d'envisager une version logicielle de la machine OPALE.

Dans le futur, une implémentation du modèle et des différents algorithmes présentés dans cette thèse sur une machine multi-processeurs nous permettra de tester plus précisement ses performances. L'algorithme de traitement des clauses récursives doit être expérimenté. Les expériences acquises dans cette thèse et les expériences accumulées au cours de prochaines expérimentations permettront de réaliser effectivement une machine base de connaissances reposant sur Prolog.

ANNEXE

Comparaison de performance de 3 interpréteurs Prolog (avec les benchmarks de warren [War77])		
Machine	intepréteur	performance
VAX-785/UNIX	YAAP	2000 LIPS
VAX-785/UNIX	C-Prolog	1630 LIPS
SM90/UNIX	YAAP	1060 LIPS
SM90/UNIX	D-Prolog	850 LIPS
SM90/UNIX	C-Prolog	520 LIPS

Comparaison de performance des systèmes Prolog (concaténation déterministe seulement)			
Machine	Système	Performance (LIPS)	Référence
Berkeley PLM	(TTL)/Compilateur	420,000	Simulation
Tick & Warren	VLSI	415,000	Tick & Warren
Symbolics3600	Microcode	110,000	::
DEC 2060	Compilateur Warren	43,000	Warren
PSI	Microcode	30,000	ICOT
IBM 3033	Waterloo	27,000	Warren
VAX-780	Macrocode	15,000	Tick & Warren
SUN-2	Quintus Comp.	14,000	Warren
LMI/Lambda	Uppsala	8,000	::
VAX-780	Prolog	2,000	::
VAX-780	M-Prolog	2,000	::
VAX-780	C-Prolog	1,500	::
Symbolics3600	Interpréteur	1,500	::
PDP 11/70	interpréteur	1,000	::
SM90	YAAP interpréteur	830	W.Dang
SM90	C-Prolog interpréteur	820	::
SM90	D-Prolog interpréteur	750	::
Z-80	Micro-Prolog	120	Warren
Apple-II	interpréteur	8	::

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Anc83] F.Anceau
CAPRI: A Design methology and a Silicon Compiler for VLSI circuits specified by algorithms.
Caltesh conf. on VLSI, 3/1983

[Bab79] E.Babb
Implementing A Relational Database By Means Of Specialized Hardware.
ACM tran. on database system, Vol.4 No.1 1979

[BaS80] F.Bancilhon, M.Scholl
Le Filtrage De Données Dans La Machine Base De Données VERSO.
Journées INRIA MBD, 9/1980

[Ban85] F.Bancilhon
Les Recherches En Bases De Données à MCC.
Congrès AFCET
Matériels et logiciels pour la 5e génération, mars 1985, Paris

[BDI84] G.Berger Sabbatel, W.dang, J.C.Ianeselli
Unification For A Prolog Database.
Second international logic programming conf. Uppsala,7/1984

[BDI85] G.Berger Sabbatel, W.dang, J.C.Ianeselli
Search Strategy And Unification For OPALE Machine.
Congrès AFCET.
Matériels et logiciels pour la 5ème génération.
Mars 1985, Paris

[BeI84] G.Berger Sabbatel, J.C.Ianeselli
Un Atomate d'Unification Pour la Machine OPALE.
R.R TIM3/IMAG No.465 oct.1984

[BeN82] G.Berger Sabbatel, NGUYEN Gia Toan
Projet OPALE: Motivations et Principes Pour Une Machine Base de Données Prolog.
R.R IMAG No.339 dec.1882

[Ber85] G.Berger Sabbatel
OPALE: vers un système pour le traitement des connaissances.
Actes des journées SM90, Versailles, 12.1985.

[BoK82] K.A.Bowen, R.A.Kowalski
Amalgamating Language And Meta-Language In Logic Programming.
Logic programming, 6/1982, academic press

[BoM72] R.S.Boyer, J.S.Moore
The sharing of structure in theorem-prouving programs.
machine intelligence 7, 1972.

[Bor84] P.Borgwardt
Parallel Prolog using stack segments on shared-memory multi-processors.
Proc. 1984 int. symp. Logic Programming. Feb.1984.

[BoD82] H.Boral, D.J.Dewitt
Applying Dataflow Techniques To Database Machines.
IEEE computer, 8/1982

[CaD85] M.J.Carey, D.J.Dewitt
Extensible database systems.
RR-585, Computer sciences Departement, University of Wisconsin.

[CiH83] A.Ciepielewski, S.Haridi
A Formal Model For Or-Parallel Execution Of Logic Programs.
IFIP 83, pp.299

[ClG81] K.L.Clark, S.Gregory
A Relational Language For Parallel Programming.
Proc. of ACM conf. on functional language and comp. archi. 10/1981

[ClG83] K.L.CLARK, S.GREGORY
PARLOG : A Parallel Logic Programming Language.
Research report DOC83/5, imperial college, 5/1983

[ClM82a] K.L.CLARK, F.G.McCabe
PROLOG : A Language For Implementing Expert System.
Machine intelligence 10, 1882

[ClT82b] K.L.Clark, S.A.Tarnnlud
Logic programming.
Academic press, 6/1982

[CMT 81] U.S.Charkrarthy, J.Minker, D.Tran
Interfacing Predicte Logic Language And Relational Database.
1st logic programming conf. 1981

[CNM85] J.Chomicki, N.H.Minsky
Towards a programming environment for large Prolog programs.
LCSR-TR-71, Labo. for Computer science Research, Rutgers university, 5.1985.

[Cod70] E.F.Codd
A relational model of data for large shared data banks.
Comm. ACM, Vol.13, No.6, 1970.

[Coi84] P.Cointe
Comprendre Smalltalk, Penser Smalltalk.
R.R de IJTP, univ. Paris-8, 1984

[CoK81] J.S.Conery, D,F.Kibler
Parallel Interpretation Of Logic Programs.
ACM conf.on functional prog. and comp.arch. oct/1981

[CoK85] J.S.Conery, D.F.Kibler
AND Parallelism And Non-determinism In Logic Programs.
New generation computing No.3 1985

[Col77] A.Colmerauer
Programmation en logique du premier ordre.
Actes journées la compréhension, IRIA (1977).

[Cra85] J. Crammond
A Comparative Study Of Unification Algorithms For OR-Parallel Execution Of Logic Languages.
IEEE Tran. on Computer Vol. C-34 No.10, Oct.1985.

[CrM84] J.A.Crammond, C.D.F.Miller
An Architecture For Parallel Logic Programs.
Second international conf. on logic programming, Uppsala,july/1984

[Dan85] W.Dang
Prolog concurrent sur SM90.
note inter, Lab.TIM3, 1.1985.

[Dan86] W.Dang
Programmation concurrente en langage orienté objet.
3ème journées langages orientés objet, Paris, 1.1986.

[DDP85] T.P.Dobry, A.M.Despain, Y.N.Patt
Performance studies of a Prolog machine architecture.
12th inter. symp. on comp. archi. Silver spring, 6.1985.

[DDS80] A.L.Davis, W.M.Denny, I.Sutherland
A characterization of parallel systems.
UUCS-80-108, Computer science departement, University of Utah, 8.1980.

[DeP85] A.M.Despain, Y.N.Patt
Aquarius - a high performance computing system for symbolic/numeric application.
Compcon85, 13th IEEE computer society conference, San Francisco, 2.1985.

[Dew79] D.J.Dewitt
Direct: A Multiprocessor Organization For Suporting Relational Database Management System.
IEEE tran. on computer, Vol.C-28, No.6, 1979

[Dew81] D.J.Dewitt
A Performance Evaluation Of Database Machine Architectures.
VLDB 81

[Dig82] Digital equipment corporation
Cours pour les ingénieurs du système VAX.
1882.

[DKM84] C.Dwork, P.C.Kanellakis, J.D.Mitchell
On the Sequential Nature of Unification.
J. Logic Programming, 1.1984.

[FKT84] K.Furrkamwa, S.Kunifuji, A.Takeuchi
The Conceptual Specification Of The Kernel Language Version 1.
TR-054, ICOT, 3/1984

[Got84] A.Goto, et al.
Highly Parallel Inference Engine PIE.
New generation computing, 2/1984

[Han85] J.Han
Pattern-Based and Knowledge Based Query compilation For Recursive Data Bases.
Thesis Ph.D, university of Wisconsin-Madison, 1985.

[Hew77] C.Hewitt
Viewing control structures as patterns of passing messages.
Artificial intelligence, 6.1977, p323-364.

[Hoa78] C.A.R.Hoare
Communicating Sequential Processes.
CACM Vol.21, 8/1978

[HoR73] J.J.Horning, B.Randell
Process Structuring.
Computing surveys, Vol.5, No.1, March 1973.

[Ico84] ICOT journal, No.3 1984

[Inm84] Inmos limited
OCCAM programming system.
1984 (VAX/VMS host manuel).

[Ian85] J.C.Ianeselli
Un opérateur d'unification pour une machine base de connaissances Prolog.
Thèse de doctorat 3ème cycle, INPG, 1985.

[JCD78] A.K.Jones, R.J.Chansler, Jr.Durham et al.
Programming issues raised by a multi-processor.
Proc. IEEE Vol.66, pp 229-237, Feb. 1978.

[Kac83] P.Kacsuk
Generalized Data Flow Model For Programming Multiple Microprocessor System.
Proc. of 3th symp. on microcomp. and microproc. Budapest, 1983

[Kac84] P.Kacsuk
A Highly Parallel Prolog Interpreter Based On The Generalized Dataflow Model.
2nd intern. conf. on logic programming, Uppsala, july/1984

[Kow74a] R.Kowalski
Predicate Logic As A Programming Language.
IFIP 74

[Kow74b] R.Kowalski
Logic For Problem Solving.
DEL Memo. Dept. of AI, imperial college, 1974

[Lau83] J.L.Laurière
Représentation et utilisation des connaissances - Premiere partie: les systèmes experts.
TSI. 83

[Lie86] H.Lieberman
Delegation and inheritence: two mecanisms for sharing knowledge in objet oriented systems.
3ème journées de langages orientés objet, Paris, 1.1986.

[Lin85] G.Lindstrom
OR-parallelism on Applicative Architecture.
Proc. second int. logic programming conf. july 1984.

[Lio77] J.Lions
UNIX Operating System Source Code Level Six.
University of new south wales, juin,1977.

[LuO84] Ewing Lusk, Ross A. Overbeek
Stalking the Gigalip.
Computer achitecture newsletter, sept.1984 pp.138

[Mac79] D.B.MacQueen
Models for distributed computing.
R.R.351, IRIA.

[Mag85] G.Mago
Making parallel computations simple: the FFP machine.
Compcon85, 13th IEEE computer society conference, San Francisco, 2.1985.

[Mai85] C.d. Maindreville
Evaluation of recursive predicats in deductive databases.
RR.467, INRIA, 1985.

[Maz78] G.Mazaré
Structure multi-microprocesseurs - problème de parallélisme, définition et évaluation d'un système particulier.
Thèse d'état, INPG, 1978.

[MKK83] T.Miyachi, S.Kunifuji, H.Kitakami, K.Furukava, A.Takeuchi, H.Yokota
A Knowledge Assimilation Methode for Logic Databases.
TR-025,ICOT, septembe 1983.

[Nil80] N.J.Nilsson
Principles of artificial intelligence
Tioga publishing company, Palo Alto, California, 1980.

[NKH85] R.Nakazaki, A.Konagayam, S.Habata et al.
Design of a high speed Prolog machine.
TM-0105, ICOT, 4.1985.

[OAS85] R.Onai, M.Aso, H.Shimizu, K.Masuda
Architecture of a Reduction-Based Parallel Inference Machine: PIM-R.
New generation computing No.3 1985

[Rob65] J.A.Robinson
A Machine-Oriented Logic Based On The Resolution Principe.
JACM Vol.12 1965

[RoL85] J.Rohmer, R.Lescoeur
La méthode d'Alexandre; une solution pour les axiomes récurisfs dans les bases de données déductives.
Congrès reconnaissance des formes et intelligence artificielle, Grenoble, 1985.

[Ros85] S.J.Rosenschein
Formal theories of knowledge in AI and robotics.
New generation computing, Vol.3, No.4, 1985.

[Rou75] P.Roussel
Prolog : Manuel De Reference Et D'utilisation.
Groupe d'intelligence artifitielle, Marseille, 1975

[San82] E.Santance et al.
Prolog Application In Hungary.
Logic programming, 1982, academic press

[San85] J.P.Sansonnet
La Machine Pour Les Applications En Intelligence Artifitielle : MAIA.
Congrès AFCET. Matériels et logiciels pour la 5ème génération.
Paris 3/1985

[Sha83a] Ehud Y. Shapiro
A Subset Of Concurrent Prolog And Its Interpreter.
ICOT Report TR-003 1983

[Sha83b] E.Y.Shapiro
System Programming In Concurrent Prolog.
ICOT TR-034, 10/1983

[ShT83] E.Y.Shapiro, A.Takenchi
Objet Oriented Programming In Concurrent Prolog.
Journal of new generation computing, Vol.1, 1/1983

[Shi84] S.Shibayama et al.
A Relational Database Machine With Large Semiconductor Disk And Hardware Relational Algebra Processor.
New generation computing, 2/1984

[SIS85] S.Shibayama, K,Iwata, H.Sakai
A knowledge base architecture and its experimental hardware.
TM-0117, ICOT, 6.1985.

[Sno83] R.Snodgrass
An objet-oriented Command language.
IEEE Tran. on software engineering, Vol.SE-9, No.1, 1.1983.

[Tan85] H.N.Tan
RSESS interconnection network.
1985 inter. conf. on parallel processing, 8.1985.

[TBH82] P.C.Treleaven, D.R.Brownbridge, R.P.Hopkins
Data-driven and demand-driven computer architecture.
Computing survey, Vol.14, No.1, 3.1882.

[TiW 84] E.Tick, D.H.D.Warren
Towards a Pipelined Prolog Processor.
New generation computing No.2 1984

[UmT83] S.Umeyama, K.Tamura
A Parallel Execution Model Of Logic Programs.
ACM 10th annual symbosum on comp. arch. 1983

[Veg84] S.R.Vegdahl
A survey of proposed architecture for execution of functional languages.
IEEE tran. on computer, c-33, No.12, 1984.

[WaD84] D.S.Warren, S.K.Debray et al.
Executing distributed Prolog programs on a broadcast network.
1984 inter. symp. on logic programming, Atlantic city, 2.1984.

[War77] D.H.D. Warren
Implementing Prolog - Compiling Predicate Logic Programs.
DAI R.R No.39-40, Edinburgh university, 1977.

[War81] D.H.D.Warren
Efficient processing of interactive relational database queries expressed in logic.
VLDB 81.

[War84] D.S.Warren
Efficient Prolog memory management of flexible control strategies.
New generation computing, No.2, 1984.

[Wil83] Wilson
Occam architecture base system design.
Computer design, 11.1983.

[YaN84] H.Yasuhara, K.Nitadori
ORBIT: A Parallel Computing Model Of Prolog.
New generation computing, 2/1984

[YMS86] A.Yonezawa, H.Masuta, E.Shibayama
An approach to objet oriented concurrent programming: a language ABCL.
3èmes journées langages orientés objet, Paris, 1.1986.

[Yok83a] T.Yokoi
A Perspective Of The Japanese FGCS Project.
ICOT TM-0026, sept/1983

[Yok83b] M.Yokpta et al.
The Design And Implementing Of a Personal Sequetial Inference Machine PSI.
New generation computing, 1/1983

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

. J. MOSSIERE, Professeur
. D. HERMAN, Professeur

Monsieur DANG Weidong

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique".

Fait à Grenoble, le 18 décembre 1986

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,