Un système Prolog parallèle pour machines à mémoire distribuée

Michel Favre

THESE

présentée par

FAVRE Michel

pour obtenir le titre de

DOCTEUR de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 23 Novembre 1988)

UN SYSTEME PROLOG PARALLELE POUR MACHINES A MEMOIRE DISTRIBUEE

Date de soutenance : 15 Avril 1992

Composition du jury :

Président	Brigitte	**PLATEAU**
Rapporteurs	Françoise	**ANDRE**
	Bernard	**TOURSEL**
Examinateurs	Jacques	**MOSSIERE**
	Jacques	**BRIAT**

Thèse préparée au sein du Laboratoire de Génie Informatique

Je tiens à remercier :

- Mme Brigitte PLATEAU, Professeur à l'Institut National Polytechnique de Grenoble et responsable du projet Calcul Massivement Parallèle de l'IMAG, qui me fait l'honneur de présider le jury de cette thèse,

- Mme Françoise ANDRE, Professeur à l'Université de Rennes, et Mr Bernard TOURSEL, Professeur à l'Université de Lille, qui ont accepté de rapporter ce travail,

- Monsieur Jacques MOSSIERE, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de l'Ecole Nationale Supérieure d'Informatique et de Mathématique Appliquée de Grenoble qui a accepté d'être mon directeur de thèse.

- Monsieur Jacques BRIAT, Maître de Conférence à l'Université Joseph Fourier, sans qui cette thèse n'aurait pas vu le jour. Je veux lui exprimer ma profonde gratitude pour les idées dont il m'a fait part et les encouragements qu'il m'a prodigués tout au long de mon ravail de recherche.

Je dois aussi beaucoup à mon collègue de travail Claudio GEYER pour son importante contribution à la réalisation de ce projet.

Je remercie enfin mes collègues des équipes "FLOP" et "SYMPA" qui m'ont aidé et encouragé.

à tous les miens.

UN SYSTEME PROLOG PARALLELE POUR MACHINES A MEMOIRE DISTRIBUEE

Résumé

Cette thèse est consacrée à l'étude de l'implantation du langage Prolog sur les architectures parallèles MIMD sans mémoire commune. Nous présentons le modèle OPERA qui exploite implicitement le parallélisme OU de Prolog pour répartir dynamiquement l'évaluation des programmes sur les différents nœuds du réseau de processeurs.

Le système OPERA est de type multiséquentiel : il n'y a parallélisation que lorsqu'un processeur est inoccupé. Ce système se décompose en une partie opérative chargée de l'évaluation du programme Prolog, et une partie contrôle chargée de l'allocation des travaux aux processeurs de la partie opérative. Les principaux problèmes de ce type de systèmes sont d'une part le choix de représentation en mémoire de l'arbre OU ainsi que la gestion des liaisons multiples, et d'autre part, le contrôle de l'allocation des différentes branches de l'arbre aux machines abstraites qui effectuent des évaluations séquentielles. La technique de régulation de charge utilisée est fondée sur des méthodes heuristiques. L'ordonnanceur d'OPERA travaille sur une image approchée de l'état global du système obtenu par échantillonnage des états locaux de chaques unités de travail.

Un prototype d'OPERA a été réalisé sur un réseau de Transputers reconfigurable dynamiquement : le Supernode. Cette propriété a été mise à profit dans notre implantation pour réduire les coûts de communication. Les communications sont effectuées en parallèle avec le calcul. Le prototype réalisé fournit des gains de performances importants et OPERA figure parmi les systèmes Prolog parallèles les plus efficaces à l'heure actuelle.

Mots-Clés : Prolog, Parallélisme, Multi-séquentiel, Régulation de charge, WAM, Multiprocesseur à Mémoire distribuée.

A PARALLEL PROLOG SYSTEM FOR DISTRIBUTED MEMORY MULTIPROCESSOR

Abstract

This thesis is dedicated to implementation of Prolog on distributed memory architectures. The OPERA model is presented. It uses implicit parallelism (OR parallelism) to distribute dynamically program evaluation on several nodes of a network of processing elements.

The OPERA system is multisequential: parallelization occurs only when a processor become idle. The system is composed of an operative part which computes Prolog program an a control part which allocates jobs to workers. Major problems of this system are the choice of a memory representation for the OR tree, the management of multiple binding and control on branch allocation to abtract machines (based on WAM). The load balancing strategy is based on heuristics and the scheduler works on an approximate global state obtained by sampling workers's local states.

An OPERA prototype has been implemented on Supernode which is dynamically reconfigurable Transputer network. Dynamic configuration is used to reduce communication overhead. In addition, communications are performed in parallel with computation. The prototype provides good speedups and OPERA is one of the most efficient existing parallel Prolog system.

Keywords : Prolog, Parallelism, Multisequential, Load Balancing, WAM, Distributed Memory Architecture.

Chapitre 1

INTRODUCTION

1.1 Motivations

1.1.1 Le parallélisme

Les applications informatiques croissent sans cesse en volume et en complexité. Cette évolution se traduit par une double demande :

- un accroissement de performance pour un coût constant ou décroissant,
- des outils de développement permettant une amélioration de la productivité de logiciel.

Le parallélisme en Prolog peut donc être justifié de deux façons. En effet, Prolog est considéré comme un langage propre à permettre le développement rapide d'applications ; par contre, son efficacité est souvent moindre que celle des autres langages. Le parallélisme constituerait une bonne solution à ce manque de performance.

Accroître la puissance de la machine devient de plus en plus difficile avec les architectures de type Von Neuman car les techniques utilisées pour construire celles-ci vont bientôt atteindre des limites physiques qui semblent incontournables dans l'état de nos connaissances actuelles. Les machines multiprocesseurs MIMD sans mémoire commune sont facilement extensibles et constituent une alternative prometteuse.

La programmation de ces machines est une activité très complexe, ce qui augmente notablement les coûts de développement de logiciel déjà élevés dans le domaine des applications sophistiquées. De plus, la recherche sur les langages permettant d'exprimer un problème sous une forme parallèle n'est encore que peu développée et en retard sur les possibilités offertes dès aujourd'hui par le matériel. Prolog, et plus généralement la programmation logique, peut offrir la possibilité

de développement d'applications parallèles à un coût moindre par rapport aux autres langages actuels. La raison principale en est la possibilité d'extraction automatique de parallélisme d'un programme Prolog. Ainsi, on devrait pouvoir améliorer les performances par un accroissement de la puissance de la machine sans remettre en cause le logiciel écrit.

Si l'accroissement de performances s'avère vérifié, l'implantation de Prolog sur les différentes sortes d'architectures parallèles permettra d'exploiter une large gamme de machines de manière uniforme. Le pari est de réussir à faire fonctionner le même programme (sans aucune modification) sur un seul ou sur un grand nombre de processeurs selon l'accroissement de performance souhaité et/ou l'accroissement de la taille du jeu de données traitées. C'est l'objectif du projet OPERA.

1.1.2 Les problèmes à résoudre

Les voies de recherche sur l'exploitation du parallélisme au niveau langage de programmation sont nombreuses. Deux approches sont possibles :

1. Utiliser un langage exprimant le parallélisme : il s'agit alors soit de porter des applications écrites pour des machines séquentielles sur des machines parallèles, soit de faciliter le codage d'un problème intrinsèquement parallèle. Le premier type d'objectif peut être atteint en étendant un langage de programmation classique, typiquement un langage impératif, par des constructeurs parallèles (introduction des notions de processus et de communications dans le langage ou son environnement d'exécution). Dans le deuxième cas, le parallélisme est une fin en soi. Les constructeurs parallèles de ce type de langages sont en général beaucoup plus nombreux, ont une sémantique beaucoup plus précise que dans le premier cas. Cette approche consiste à fournir au programmeur les outils de base pour qu'il gère explicitement le parallélisme.

2. Utiliser un langage dont les propriétés sont telles que l'on peut extraire automatiquement des parties exécutables en parallèle. Cette approche consiste à exploiter le parallélisme implicite d'une application tout en permettant au programmeur de s'abstraire de l'aspect gestion du parallélisme. Dans cette approche, les recherches portent essentiellement sur les techniques de compilation capables de détecter automatiquement le parallélisme potentiel et sur la définition de modèles d'exécution permettant d'exploiter ce potentiel en utilisant efficacement le parallélisme réel que procure une architecture cible. Notre thèse suit cette approche.

1.2 Le projet OPERA

1.2.1 Le contexte du projet OPERA

Depuis la réalisation du premier interprète du langage Prolog, à Marseille en 1972, par l'équipe d'A. COLMERAUER et les travaux sur la compilation de D.H.D. Warren (77) Prolog a été utilisé beaucoup plus comme langage de spécification que comme langage de programmation universel. Ceci est principalement du à ses performances à l'exécution relativement faibles devant celles obtenues par les langages impératifs classiques.

Avec le projet de machines dites "de cinquième génération" lancé au Japon par l'ICOT en 1985, les recherches en programmation logique se sont multipliées. La "contre-offensive occidentale" au projet FGCS est principalement constituée par le projet GIGALIPS (qui regroupent les laboratoires d'Argonne au USA, de SICS en Suède, de Manchester et Bristol en Grande-Bretagne), le centre ECRC crée á Munich par différents constructeurs européens (BULL, SIEMENS,ICL...). C'est dans ce contexte que le projet OPERA est né à l'IMAG.

Alors que la plupart des équipes choisissaient de travailler sur des machines parallèles à mémoire commune, le projet s'est orienté vers des machines à mémoire distribuée en participant à la conception d'une telle machine : le Supernode (projet ESPRIT 1085).

1.2.2 Objectif et aperçu du projet

L'objectif du projet OPERA est d'implémenter efficacement le langage Prolog sur une architecture parallèle à mémoire distribuée. Il s'agit de paralléliser automatiquement l'exécution du langage Prolog tel qu'il est, c'est-à-dire sans intervention du programmeur et sans modification du langage ou des programmes existants.

Le modèle de calcul d'OPERA exploite le parallélisme OU selon un modèle multiséquentiel.

Chaque processeur consiste en une machine Prolog séquentielle. Ces machines coopèrent à l'exécution du même programme. La communication entre processeurs se réduit à l'échange d'une tâche entre une machine surchargée et une machine inactive. On n'installe une tache que si son exécution parallèle produit un accroissement d'efficacité (régulation de charge).

L'implémentation de la machine Prolog est basée sur la machine abstraite de D.H.D. Warren (WAM), considérée comme la technique la plus efficace, à l'heure actuelle, pour la compilation de Prolog. Chaque processeur de la machine parallèle execécute une TWAM. Lorsqu'un processeur oisif demande un travail à un processeur actif, l'état de la machine abstraite active est copiée sur l'unité inactive.

La répartition de charge est effectuée par une hiérarchie de processeurs spécialisés dans cette tâche et opérant en parallèle avec l'évaluation du programme. Comme il est impossible de maintenir un état global exact, les répartiteurs de charge doivent décider à partir d'informations approchées.

Un prototype d'OPERA a été implémenté sur une machine Supernode composée de 16 unités de travail. Ce prototype, aujourd'hui opérationnel, fournit d'ores et déjà une accélération effective par rapport aux plus efficaces des systèmes Prolog séquentiels commerciaux. Sur des programmes présentant un parallélisme potentiel suffisant, le prototype actuel délivre des accélérations quasi-linéaires du nombre de processeurs. Il est encore améliorable car de nombreuses optimisations sont encore possibles et constituent une base expérimentale importante pour orienter les études à venir.

1.2.3 La contribution de l'auteur

Le projet OPERA est le résultat d'un travail d'équipe. J'ai participé activement à la définition du Projet OPERA dès son début. J'ai contribué à la définition, réalisation, validation de l'ensemble des nombreux outils nécessaires au projet. J'ai, en particulier, assuré la coordination et l'intégration des différents composants du système OPERA. Le projet OPERA a nécessité un travail technique considérable en particulier nous avons réalisé une chaîne de développement complète (compilateurs Prolog, C//) et un micro-noyau de système parallèle.

1.3 Structure de la thèse

Nous avons choisi de ne présenter dans ce document que les principes généraux liés à la parallélisation automatique de Prolog sur machine à mémoire distribuée.

Le chapitre 2 rappelle le mécanisme d'évaluation du langage Prolog et sa compilation. Suit une présentation des différentes approches visant à l'exploitation du parallélisme en programmation logique et les différentes modèles de parallélisme.

Le chapitre 3 traite d'une classe de systèmes Prolog parallèle à laquelle notre système OPERA appartient : les système multi-séquentiels.

Les hypothèses de travail et les principes généraux du modèle OPERA sont présentés au chapitre 4. La partie opérative (moteur d'évaluation Prolog) y est également décrite.

Le chapitre suivant (5) est entièrement consacré à la partie contrôle du parallélisme. Nous y examinons le problème de l'obtention de gain de performance dans un cadre général, puis nous y décrivons les choix retenus dans le modèle OPERA. L'aspect dynamique des tâches parallèles extraites de l'évaluation de

programme Prolog sur les architectures multiprocesseurs à mémoire distribuée nous amène à examiner le problème de la régulation de charge.

La description de l'implantation du prototype OPERA sur Supernode et de ses premiers résultats fait l'objet du chapitre 6.

Une annexe décrit l'environnement de développement C parallèle et noyau de système support d'OPERA. Suivent quelques exemples de programmes de tests utilisés pour les mesures.

Chapitre 2

PROLOG : COMPILATION - TYPE DE PARALLELISME

2.1 Structure du chapitre

Ce chapitre introductif présente le langage Prolog,sa compilation et quelque unes des différentes formes de parallélisme que procurent les langages logiques. Les notions qui y sont présentées sont assez générales et constituent le prérequis nécessaire à une bonne compréhension des chapitres suivants.

La section 2.2 rappele brièvement le langage Prolog et les mécanismes de son évaluation. La section 2.3 porte sur la technique de compilation de ce langage pour une machine séquentielle. En particulier, la structure de la machine abstraite cible de la compilation (WAM) y est présentée. Une version modifiée de cette machine abstraite constitue la brique élémentaire de notre système Prolog parallèle. Enfin la section 2.4 présente différentes approches pour exploiter le parallélisme en programmation Logique en situant notre travail dans ce cadre.

2.2 Le langage Prolog

2.2.1 Concepts et notations

Cette section présente succintement la terminologie généralement usitée qui fait référence à la logique dans le but de préciser les notions utilisées dans le langage Prolog.

formule atomique : Une formule atomique ou atome se présente syntaxiquement sous la forme d'un terme. Exemples : $p(X), q, t(X, f(Y))$ sont trois atomes. La logique du premier ordre interdit l'usage de formule atomique se réduisant à

une variable. Le foncteur principal muni de son arité est appelé prédicat (p, q et t dans les exemples ci-dessus).

Formule , conjonction, disjonction, négation, implication: Les formules logiques qui nous intéressent sont construites à partir de

- conjonction de 2 formules logiques (notée F1 , F2)
- ou de disjonction de 2 formules logiques (notée F1 ; F2)
- ou de négation d'une formule logique (notée not F)
- ou d'implication de 2 formules logiques (notée F1 :- F2 et qui se lit F2 implique F1)
- les formules atomiques sont des formules logiques

clause de Horn - tête - corps - fait - prédicat: Une clause de horn est soit un fait (exemple: $P(X)$.) soit une clause définie.

Une clause définie est une formule logique de la forme $A : -B$ où A désigne une formule atomique et B une formule quelconque ne contenant pas d'implication. L'atome A est appelé tête de clause, la formule logique B qui est une conjonction est appelée corps de clause.

En pratique, on autorise l'utilisation des disjonctions (factorisation de l'écriture de la tête de clause) et de la négation (négation simulée par l'échec) dans le corps de clause.

Un fait dénote une clause définie dont le corps est vide (égal à vrai).

Un programme Prolog est un ensemble de clauses de Horn (syntaxiquement, chacune d'entre elles termine par un point). Un programme Prolog peut être vu comme une disjonction de clauses et un but à prouver.

Exemples:

```
pere( albert , paul ).
pere( albert , jacques ).
pere( pierre , albert ).

grandpere( X , Y ) :- pere( X , Z ) , pere( Z , Y ).
frere( X , Y ) :- pere( Z , X ) , pere( Z , Y ).

:- grandpere( pierre , X )  % but a prouver
```

Stratégie d'évaluation séquentielle

Nous utiliserons l'arbre d'évaluation ET/OU comme représentation de l'arbre de preuve. L'évaluation d'un programme Prolog consiste en un parcours exhaustif de cet arbre. Plusieurs parcours sont possibles. Un effort important a été consacré à la définition de stratégies d'exécution (séquentielle ou parallèle) pour langages logiques. L'arbre pouvant contenir des branches infinies, la stratégie de parcours est essentielle pour garantir que toutes les solutions (preuves possibles) soient trouvées (propriété de complétude).

L'arc entre un nœud père et un nœud fils de l'arbre ET/OU matérialise une relation de précédence: un nœud père doit être évalué avant ses nœuds fils. C'est la seule contrainte qui découle de la méthode de preuve par réfutation. On définit ainsi un ordre partiel sur les nœuds. Tout ordre total compatible avec cet ordre partiel est une stratégie d'évaluation, il en existe donc un très grand nombre (autant que de permutations de nœuds non comparables selon l'ordre partiel). Il en existe plusieurs permettant de calculer solution par solution (tous les ordres totaux compatibles avec l'ordre partiel défini par l'arbre ET d'une solution). En outre, il est possible d'évaluer simultanément plusieurs solutions (là encore toutes les façons de parcourir l'arbre OU sont des stratégies possibles d'évaluation). Les stratégies de parcours de l'arbre ET/OU sont en fait toutes les combinaisons des stratégies de parcours de l'arbre ET et de celles de l'arbre OU. Les stratégies parallèles sont dérivées des stratégies séquentielles, elles prennent en compte des critères architecturaux. Cet aspect est examiné dans le chapitre 2 qui traite du parallélisme en Prolog.

On peut déterminer deux grandes classes de stratégies de parcours séquentiel d'un arbre ET/OU :

les stratégies dépendantes du contexte où le choix du parcours d'un arbre de preuve (ET) comme celui des différentes alternatives (OU) dépend d'heuristiques portant sur les propriétés structurelles du parcours (conditions globales) ou sur l'état des variables (conditions locales) .

Ces approches ont conduit à des variantes de langage proposant soit une extension du langage Prolog pour exprimer ces fonctions de sélection (ex : [Dinc84]), soit essayant d'intégrer des algorithmes de recherche heuristique comme par exemple des algorithmes de type A*, soit offrant un canevas permettant d'exprimer des systèmes de contraintes à satisfaire sur un parcours [Dinc88][Colm90].

les stratégies "aveugles" : Ces stratégies sont indépendantes du contexte car elles fixent un ordre de parcours de l'arbre "a priori". Elles diffèrent par les propriétés garanties par cet ordre. Les stratégies classiques pour Prolog sont :

- la stratégie "*en largeur d'abord*" qui a pour avantage d'être complète (elle permet de trouver toutes les solutions même s'il existe des branches infinies). Par contre, elle nécessite une quantité de mémoire qui peut être de l'ordre du nombre de nœuds de l'arbre ET/OU.

- la stratégie "*en profondeur d'abord*", bien qu'incomplète est la plus utilisée dans les systèmes actuels car elle ne nécessite qu'une taille mémoire de l'ordre du plus long chemin parcouru (entre la racine et une feuille donnée) de l'arbre. Elle consiste à sélectionner en premier le sous-but le plus à gauche d'une conjonction (règle du nœud ET) et choisir d'abord la première clause du texte source unifiable avec le sous-but (règle du nœud OU). En cas d'échec de la recherche de solution on effectue un *retour-arrière* qui permet de revenir à l'état d'évaluation au moment de ce choix puis on essaye l'alternative suivante. Cette stratégie de parcours de l'arbre ET/OU correspond à l'ordre de lecture du texte source ce qui facilite la maîtrise des évaluations à effets de bord des prédicats prédéfinis qui font de Prolog un langage de programmation. Le contrôle de l'évitement de branches infinies est à la charge du programmeur auquel sont fournis des opérateurs méta-logiques (ex: le coupe-choix).

- la stratégie mixte "*deep iterative deepenning*" qui peut être vue comme un mélange des deux stratégies précédentes. Le but de cette stratégie est de profiter des avantages de la stratégie "en profondeur d'abord" tout en conservant la complétude: pour ce faire on limite le parcours d'une portion chemin à une profondeur bornée avant de commuter (en largeur) sur la portion d'un autre chemin.

Il y a un rapport exponentiel entre le nombre de ressources nécessaires à la stratégie en largeur d'abord et celui requis par la stratégie en profondeur d'abord. Pour un arbre "ou" n-aire équilibré de hauteur h, la quantité de ressources est de l'ordre de h pour la stratégie "*en profondeur*" alors qu'elle est de l'ordre de n^h pour stratégie "*en largeur*".

L'avantage principal de ces stratégies résident dans la possibilité de compilation très poussée (puisqu'elles sont fixes). Cette compilation produit un programme "localement" efficace (cf. section 2.3).

2.3 Compilation du langage Prolog

Cette section décrit une façon de compiler Prolog séquentiel pour une machine à architecture séquentielle. Le but étant de faire une synthèse des problèmes de compilation et de faire comprendre le mécanisme d'évaluation séquentiel de Prolog sans pour autant noyer le lecteur dans une profusion de détails techniques. Aussi nous ne tracerons que les grandes lignes sans donner le jeu d'instructions complet de la machine virtuelle cible du compilateur.

Un des intérêts de Prolog est de pouvoir coder un ensemble de données sous forme de termes en faisant abstraction de la représentation interne de ces termes sur une architecture particulière. L'unification est alors l'outil universel de construction, décomposition, et d'identification de termes. L'interprétation logique ne joue un rôle qu'au niveau du contrôle du flot d'exécution ; son intérêt principal est l'aspect déclaratif qu'elle confère au langage car cela laisse un degré de liberté très élevé (au moins en théorie) dans le séquencement des opérations.

La compilation de Prolog est basée sur l'idée d'interprétation procédurale de Prolog. Nous décrirons donc cet aspect avant de donner les principes de la compilation du langage.

2.3.1 Interprétation procédurale

L'idée d'interprétation procédurale de Prolog n'est pas récente. Due à Kolwalski [Kowa74][Kowa79], elle date des travaux qui ont montré l'utilité de Prolog en temps que langage de programmation. D.H.D Warren a proposé la première implémentation par compilation vraiment efficace qui a permis d'envisager l'écriture de gros programmes d'application en Prolog.

Selon l'interprétation procédurale, la disjonction de clauses est considérée comme un ensemble de procédures.

Une procédure est constituée par l'ensemble de toutes les clauses dont la tête admet le même foncteur principal (nom de prédicat) et la même arité. Les clauses d'une même procédure sont ordonnées selon leur position relative dans le texte source.

Le corps de clause correspond au corps de procédure: les différents littéraux apparaissant dans la conjonction qui constitue le corps de clause sont autant d'appels de procédures. La conjonction est, dans cette interprétation, considérée comme l'opérateur de composition séquentielle.

Les sous-termes des littéraux du corps de clause sont les arguments réels alors que les sous termes du littéral de tête sont les "arguments formels" (ces derniers servant à "filtrer" les arguments réels).

Un programme est constitué d'un but initial (littéral à prouver) c'est-à-dire de l'appel d'une procédure. Les données initiales sont les sous-termes du littéral.

Dans Prolog, contrairement aux langages procéduraux classiques (type C ou Pascal), le non déterminisme fait partie du langage: la multiplicité des clauses dans une même procédure offre des alternatives dans l'exécution pouvant toutes produire un résultat.

Exemple:

```
pere( albert , paul ).      %|
pere( albert , jacques ).   %| procedure
pere( pierre , albert ).    %|

%     arg. formel    appel de procedure
%          ---           -------------
grandpere( X , Y ):- pere( X , Z ) , pere( Z , Y ).

:- grandpere( pierre , X )  | but initial
%       ------
%            arg. reel
```

2.3.2 L'origine de la WAM

La WAM (Warren Abstract Machine) est la machine virtuelle qui a été proposée par D.H.D Warren en 1983 [Warr83]. Elle est l'aboutissement de plusieures années de recherche sur les techniques d'implémentation par compilation de Prolog depuis le pas décisif du premier compilateur Prolog sur machine DEC10 ([Warr77]). Sa grande efficacité par rapport aux techniques d'interprétation a considérablement marqué les développements ultérieurs. Cette machine abstraite est devenue le "standard de fait" de la compilation de Prolog. Il en existe de nombreuses variétés plus ou moins optimisées mais toujours construites sur le même principe.

La description qui suit ne remplace en rien la lecture des articles originaux sur la WAM mais elle doit être vue comme un complément synthétique. Nous éviterons autant que possible de faire une présentation exhaustive de la WAM en omettant les détails trop techniques (car il sont très nombreux). Nous conseillons au lecteur intéressé de se referer à l'article original de D.H.D Warren 1983 (un peu succinct) et surtout au récent rapport (beaucoup plus pédagogique) de Hassan Ait Kaci [AitK90].

Le but de cette machine abstraite est d'offrir une architecture mémoire et un jeu d'instructions bien adaptés à Prolog et qui soient de niveau suffisamment bas pour être efficaces et rapidement portables sur n'importe quelle machine réelle.

Nous allons aborder progressivement le jeu d'instructions et préciser au fur et à mesure les structures de données correspondantes. La démarche suivie suppose le lecteur familier des techniques de compilation des langages procéduraux classiques (ex C, Pascal) avec lesquelles nous effectuerons quelques parallèles.

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

2.3.3 Les instructions de contrôle

Il s'agit d'une part du contrôle procédural classique (déterministe), et d'autre-part du contrôle du non déterminisme (retour arrière). Les instructions de contrôle sont intimement liées à la stratégie d'évaluation séquentielle choisie pour Prolog. Le contrôle dit "classique" correspond à l'ordonnancement de "gauche à droite" pour évaluer les conjonctions, et le contrôle du non-déterminisme correspond aux tentatives d'évaluation des différentes clauses d'une procédure dans l'ordre où elles apparaissent dans le texte source.

contrôle déterministe classique

La premiere catégorie correspond à la gestion des environnements et des appels de procédure (prédicat). Ce sont des notions classiques que l'on retrouve dans les langages de programmation impératifs. Une pile dite `Locale` stocke les environnements des procédures appelées (technique classique de réalisation du renommage de variables).

Un environnement Prolog est constitué d'une adresse de retour et de variables locales. Les arguments de la procédure sont passés par registres (dont on suppose le

nombre suffisant).

L'instruction **allocate** alloue un environnement sur la pile locale lors de l'entrée dans la procédure et **deallocate** libère cet environnement en fin de procédure. L'appel de procédure est effectué par l'instruction

```
call <procedure> , <taille_environ_appelant>
```

et le retour normal par **proceed**

Une généralisation de l'optimisation classique de l'appel terminal consiste à récupérer une partie de l'environnement dès que possible. Pour ce faire, un paramètre supplémentaire de l'instruction **call** désigne la taille restante de l'environ appelant à protéger lors du prochain **allocate** exécuté par la procédure appelée. De même, on transforme un appel terminal en un simple *goto*. Cette optimisation est également prise en compte dans la WAM par l'instruction.

```
execute <procedure>
```

Exemple: la compilation du corps de la clause

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

donnera la séquence d'instruction suivante [1].

```
allocate            % alloue l'environnment courant
...                 % passer X par registre
call    b/1 , 2     % X et Y sont preservees pour l'appel c( X , Y )
...                 % passer X et Y par registres
call    c/2 , 1     % Y doit etre preservee pour l'appel d( Y )
...                 % passer Y par registre
deallocate          % l'environnment courant n'est plus utile
execute d/1         % simple branchement
```

Le passage d'argument par registre (ou par toute autre ressource distincte de la pile locale) rend trivial l'optimisation de l'appel terminal (analogue à celle que l'on peut trouver en lisp). Cette optimisation améliore notablement l'efficacité des programmes déterministes. Un exemple complet illustrant cette optimisation peut être trouvé dans la section 2.3.4.

[1] La notation **nom_predicat/arite** spécifie le nombre d'arguments ce qui permet de distinguer les predicats homonymes

gestion du non-déterminisme

La seconde catégorie d'instructions de contrôle correspond à la gestion du non-déterminisme de Prolog du à la multiplicité des clauses définissant une même procédure.

a) Indexation des clauses : Le premier point est de réduire dès que possible ce non-déterminisme pour éviter des évaluations inutiles (celles qui finissent par un échec). Les instructions d'indexation des clauses répondent à ce besoin : elles permettent une sélection rapide d'un groupe de clauses de la procédure en fonction d'un argument. Le regroupement des clauses est fait à la compilation (de façon statique) en fonction des propriétés des arguments des têtes de clauses. Une clause peut appartenir à plusieurs groupes. Généralement, les compilateurs actuels indexent le groupe de clauses relativement au premier argument seulement, la tâche d'ordonner les arguments de manière ad'hoc étant laissée au programmeur.

Le premier niveau d'indexation porte sur le type de terme. A cette occasion signalons que tous les termes sont étiquetés selon leur type: Variable, Constante, Structure ou Liste (la liste étant une structure distinguée car d'occurence très fréquente). Ce marquage a pour but d'accélérer les tests de type qui sont très souvent nécessaires (cf. notre interprète abstrait). L'instruction

`switch_on_term <on_var><on_const><on_list><on_struct>`

a donc pour fonction d'aiguiller l'exécution vers l'une des étiquettes selon le type du premier argument de la procédure.

Le deuxième niveau d'indexation peut ne pas être présent s'il n'y a pas de clause qui corresponde à ce type de terme (auquel cas il y a échec de l'unification) ou se réduire à une seule clause (cas déterministe). Toutefois, dans le cas d'une procédure comportant un grand nombre de clauses, un deuxième niveau d'indexation sur la valeur du premier argument permet un gain important en temps d'exécution. Les programmes gérant des "bases de clauses" (petite base de données) sont ceux pour lesquels le deuxième niveau d'indexation offre le plus grand intérêt. Ainsi les instructions

`switch_on_constant <Hash_Table_Size> <Hash_Table_Addr>`

et `switch_on_struct <Hash_Table_Size> <Hash_Table_Addr>`

permettent d'indexer les clauses après recherche de la valeur du premier argument dans la table de "hash-code" spécifiée.

b) le point de choix : Le second point est d'assurer l'enchaînement des différentes tentatives d'évaluation qui peuvent encore subsister après indexation. Pour cela on introduit la notion de point de choix: c'est une structure contenant l'état actuel du calcul. Cette structure a pour but de sauvegarder cet état de calcul (les registres de la machine virtuelle) qui va évoluer ensuite au cours d'une tentative. Après l'obtention d'une solution ou après un échec de l'unification, on restaure l'état sauvegardé dans le

point de choix afin de passer à la tentative suivante. La stratégie classique de Prolog (en profondeur d'abord) consiste à choisir le point de choix le plus récemment créé pour continuer l'évaluation (gestion en pile).

Prolog comporte, implicitement, un mécanisme de retour arrière complet : c'est-à-dire qu'après un échec ou en succès , la mémoire doit être restituée dans l'état exact où elle était immédiatement au moment de la création du point de choix.

Cette capacité de retour arrière est l'une des originalité de Prolog. Elle est essentiellement basée sur le fait que les variables logiques sont à assignation unique. Par rapport aux langages impératifs Prolog est pénalisé par l'utilisation de l'assignation unique. Par contre, celle-ci facilite la restauration de l'état des variables avant leur liaison : l'état d'une variable logique étant initialement "`libre`" puis éventuellement "`liée`" à une valeur, il est extrêmement simple de restituer l'état initial de toutes les variables dont la liaison (affectation) a été effectuée postérieurement à la création du point de choix à restaurer. Pour cela il suffit de représenter l'ensemble des variables dont il faut défaire la liaison : lors de la liaison d'une variable on enregistre l'adresse de la structure représentant la variable dans une pile dite "trainée" ("`trail`" `en anglais`) ; ceci permet de remettre cette structure à la valeur "`libre`" lors du retour arrière provoqué par l'échec d'une unification ou la réussite d'une tentative.

Après avoir effectué une présélection des clauses d'une procédure à l'aide des instructions d'indexation, il peut subsister plusieurs alternatives candidates à évaluation. On code cette ensemble de clauses sous forme d'une liste chainée qui respecte l'ordre d'apparition dans le texte source. Pour améliorer les performances, on "compile" ce chaînage (puisqu'il est statique) en une série d'instructions. Les instructions qui permettent de gérer les point de choix sont :

`try <etiquette> et try_me_else <on_bactrack>` : Ces deux instructions correspondent à la création d'un nouveau point de choix. Elles apparaissent donc avant l'évaluation de la première alternative afin de préserver l'état courant pour une éventuelle reprise sur un retour arrière. Cette instruction correspond à la compilation du premier élément de chainage des clauses.

L'intruction `try` est utilisée lorsque le code de l'alternative ne peut être accédé que par une rupture de séquence ; en cas de retour arrière, la machine virtuelle revient à l'instruction qui suit le `try`.

Au contraire, l'instruction qui suit le `try_me_else` correspond à la continuation normale alors que l'étiquette fournie en argument permet de désigner l'échappement en cas d'échec. Généralement cette instruction précède le code de la première clause d'une procédure, alors que le `try` est la continuation étiquetée par les instructions d'indexation.

`trust <etiquette> et trust_me_else` Ces deux instructions correspondent à la restauration/destruction d'un point de choix. Elles apparaissent donc avant l'évaluation de la dernière alternative afin de restaurer l'état sauvegardé par `try` (resp. `try_me_else`) et détruisent aussitôt le point de choix courant pour que la reprise sur un retour arrière redevienne telle qu'elle était avant le parcours de la liste de clauses.

L'intruction **trust** correspond à la compilation du dernier élément de chaînage des clauses qui commence par **try** alors que l'instruction **trust_me_else** précède le code de la dernière alternative d'une chaîne qui débute par un **try_me_else**.

retry <etiquette> et retry_me_else <on_bactrack> Ces deux instructions correspondent à la restauration/modification d'un point de choix. Elles apparaissent dans le cas où il existe plus de deux alternatives dans l'ensemble des clauses candidates. Dans cette situation intermédiaire, il est nécessaire de restaurer l'état sauvegardé dans le point de choix mais, contrairement au **trust** (resp. **trust_me_else**), on ne détruit pas le point de choix puisque que l'on doit pouvoir restituer de nouveau cet état avant l'évaluation de l'alternative suivante. Plutôt que détruire et recréer le point de choix, ces instructions ne modifient qu'une petite partie du point de choix existant : le champ correspondant à la continuation en cas de retour arrière.

L'intruction **retry** est analogue à l'instruction **try** du point de vue des deux types de continuation, alors que, symétriquement, l'instruction **retry_me_else** est analogue au **try_me_else**.

L'exemple suivant illustre la compilation d'une procédure en listes de clauses :

```
a( 1 ) :- ...
a( 2 ) :- ...
a( X ) :- ...
a( [a,b] ) :- ...

se compile en

A:  switch_on_term SIVAR,SICONST,SILIST,C3
%-----------------------------------------
SIVAR:
    try_me_else V1
C1:    <code_clause1>
%-----------------------------------------
V1: retry_me_else V2
C2:    <code_clause2>
%-----------------------------------------
V2: retry_me_else V3
C3:    <code_clause3>
%-----------------------------------------
V3: trust_me_else
C4:    <code_clause4>
%-----------------------------------------
SICONST:
    try C1
    retry C2
```

```
    trust C3
%----------------------------------------
SILIST:
    try C3
    trust C4
%----------------------------------------
```

Bien que la compilation des chaînages de clause améliore l'efficacité de l'exécution des programmes Prolog, elle a un inconvénient important: elle est basée sur l'aspect statique de ce chaînage ce qui sous entend que le code produit est difficilement modifiable dans le cas d'ajout et/ou retrait de clauses au cours de l'exécution du programme comme le permettent les prédicats prédéfinis du type **assert** et **retract** que l'on trouve dans la plupart des systèmes Prolog. Une autre conséquence du même type est que l'ordre des clauses est établi une fois pour toutes à la compilation fixant une stratégie "déterministe", ce qui peut être en contradiction avec le degré de liberté offert par le parallélisme.

2.3.4 Les instructions d'unification

Avant de décrire la structure des instructions d'unification (celles qui gèrent la création et l'accès aux données) nous allons donner une idée de la façon dont sont représentés les termes dans la WAM ainsi que les différentes notions afférentes à la gestion des variables logiques.

La représentation des termes

Rappelons que, selon l'interprétation procédurale, les termes présents dans les clauses sont interprétés (après compilation) comme étant des valeurs immédiates et sont les arguments des instructions d'unification. Ces dernières sont la spécialisation de l'algorithme général d'unification pour le cas où un des termes est celui présent dans la clause. Nous ne décrirons pas le codage des instructions mais les paragraphes qui suivent donnent la représentation de l'autre argument de la procédure d'unification : c'est-à-dire le terme qui correspond à la donnée passée à et/ou produite par la procédure.

constantes : Les constantes élémentaires sont représentées sous forme d'une seule cellule étiquetée (en général 32 bits). Lorsqu'il s'agit de constantes particulières (ex: entier) on utilise une étiquette ("tag") distinctif. Dans le cas de constantes à format variable (symbole représentant un atome, chaîne de caractère, etc...) la cellule ne contient qu'une référence univoque (ex: un index dans une table) vers une zone où l'on archive les données de même type.

variables : Compte-tenu des opérations sur les variables (assignation d'une valeur unique, accès par indirection), on tend à confondre la représentation d'une référence

à un terme et la représentation d'une variables logique: toutes deux correspondent à une cellule d'une taille suffisante pour pouvoir stocker une adresse et un tag, ou éventuellement une constante. De manière générale, le format d'une cellule correspond à celui du mot mémoire du processeur sur lequel est implanté la machine virtuelle.

Une variable libre est représentée par une cellule pointant sur elle même. Cela permet de la différencier d'une variable liée à un terme quelconque T qui contient l'adresse de ce terme et le tag décrivant le type de l'objet pointé (variable ou terme structuré). Si la valeur de liaison T est une constante, on écrit directement la valeur de cette constante dans la cellule de la variable à lier.

termes structurés: Les termes structurés étant de tailles différentes, leurs représentations possèdent la même caractéristique. La WAM utilise la technique de recopie de structure [Warr83] Les termes structurés sont représentés par un enregistrement contenant le foncteur et l'arité suivi du vecteur contenant les références (taggées) correspondant aux différents arguments de cette forme fonctionnelle.

Allocation des variables logiques

Ayant décrit les différentes structures de données de la WAM, nous rappelons ci-dessous les principales notions liées à l'allocation des variables logiques et des termes.

passage de paramètres: Les constantes ou les références aux termes arguments de procédure sont passés en paramètres par registres. Le compilateur Prolog suppose donc que la WAM comporte un nombre suffisant de registres notés de manière générique A_i. Dans le cas de termes structurés les registres A_i contiennent la référence au terme.

Les registres ne sont pas sauvegardés au cours d'un appel, le compilateur doit donc déterminer la durée de vie de chaque variable logique apparaissant dans la clause pour savoir s'il peut les placer en registre ou non. Il distingue les:

variables temporaires: Les variables logiques temporaires sont telles qu'aucun appel de procédure n'est effectué durant leur "vie". Elle sont utilisées au cours d'opérations élémentaires (opérations d'unification, arithmétique, etc...). C'est, par exemple, le cas des variables qui n'apparaissent qu'en tête de clause avant le premier sous but ainsi que certaines variables intermédiaires introduites par le compilateur pour accéder au termes structurés. La portée des variables temporaires et leurs fréquences d'accès sont telles qu'on peut les placer dans les registres. Ces registres sont notés X_i.

variables permanentes: Au contraire, les variables permanentes notées Y_i sont celles dont la vie comprend au moins un appel de procédure. Ces variables seront donc allouées en mémoire de façon à les conserver.

L'allocation en mémoire dépend de la durée de vie des variables. relativement à à celle de la procédure qui les a créees. On distingue deux catégories:

les variables locales : Ce sont celles pour lesquelles le compilateur peut déterminer l'inclusion de leur durée de vie dans celle de l'environ de la procédure. Elle sont stockées dans la pile locale, leur allocation est statique. Elles font partie de l'environ de la procédure et sont accédées de manière analogue à celles utilisée dans les langages impératifs classiques (base d'environ+déplacement calculé à la compilation).

Signalons au passage une particularité de Prolog : en compilation, il est d'usage d'effectuer l'optimisation de l'appel terminal qui consiste à remplacer le dernier appel par un branchement : cela permet de remplacer l'environ appelant par celui appelé. Une autre optimisation, plus générale, appellée "*trimming*" consiste a récupérer une partie de l'environ dès que cela devient possible. Pour ce faire, le compilateur alloue les variables dans l'environnement par durée de vie décroissante. Ainsi, l'espace alloué par la variable dont la durée de vie est la plus faible se situe sur le sommet de la pile **locale** ; il est récupéré lors de la prochaine allocation d'environ : l'instruction **call** comporte un argument indiquant à la procédure appelée le nombre de variables de l'environ appelant restant en vie.

L'exemple ci dessous illustre la récupération continue de l'environnment (les instructions get et put sont décrites dans la section 2.3.4). L'allocation des variables de la clause

```
p(X,Y,Z) :- q(U,V,W) , r(Y,Z,U) , s(U,W) , t(X,V).
```

réalisée à la compilation est telle que :

```
allocate              % p(
get_variable Y1,A1    %    X , 1ere occurence de X => variable
get_variable Y5,A2    %    Y ,
get_variable Y6,A3    %    Z )
put_variable Y3,A1    % q( U ,
put_variable Y2,A2    %    V ,
put_variable Y4,A3    %    W
call q/3 , 6          % ) ,
put_value Y5,A1       % r( Y ,
put_value Y6,A2       %    Z ,
put_value Y3,A3       %    U ,
call r/3 , 4          % ) ,
put_value Y3,A1       % s( U ,
put_value Y4,A2       %    W ,
call s/2 , 2          % ) ,
put_value Y1,A1       % t( X , 2eme occurence de X => value
put_value Y2,A2       %    V
deallocate            % )
execute t/2           % .
```

les variables globales : Au contraire des variables locales, les variables globales sont celles dont la durée de vie peut dépasser celle de la procédure (clause) qui les a créées. Il s'agit essentiellement des variables qui sont contenues dans les instances de termes "résultats". A la différence des variables locales, l'allocation des variables globales est dynamique car la nécessité de persistance ne peut pas être détectée par les compilateurs de la génération actuelle qui ne font pas d'optimisations globales (entres procédures). Ces variables sont stockés une pile dite **globale** (qui est, a peu près, l'équivalent du tas des langages classiques avec une gestion en pile). L'espace occupé par les variables gobales n'est récupéré que lors du retour arrière (ou par le ramasse-miettes si le système en possède un).

allocation des termes structurés : Leur taille étant variable, on ne peut pas les stocker dans des registres donc il est nécessaire de leur allouer de la mémoire. On ne peut les stocker dans la pile `locale` puisqu'il n'est pas possible de faire une allocation statique. En outre, les procédures peuvent construire des termes qui doivent persister après la destruction de l'environ de la procédure qui les a crées. L'allocation est effectuée dynamiquement dans la pile `globale` : l'allocation d'un terme sur ce tas correspond à empiler ce terme, la libération d'espace (retour du sommet de pile à son ancienne valeur) n'étant effectuée que lors du retour arrière.

Les instructions d'unification

On peut considérer que, de manière abstraite, les instructions d'unification correspondent à l'évaluation partielle de l'algorithme d'unification avec un des deux termes arguments connu (le terme contenu dans la clause). Chaque instruction est donc une spécialisation de cet algorithme selon le type d'argument connu (variable, constante ou terme structuré).

Le codage de l'algorithme d'unification à un terme structuré est donc une composition séquentielle de ces instructions suivant le parcours en ordre préfixé de l'arbre représentant le terme apparaissant en argument de la tête de clause. Les termes "arguments formels" sont unifés aux termes "arguments réels" (référencés par les registres arguments A_i) tour à tour dans l'ordre textuel d'apparition.

Les occurences de variables sont distinguées : s'il s'agit de la première occurence, il ne peut pas y avoir de problème de cohérence : l'algorithme d'unification se réduit à une simple affectation, par contre, lors des occurences suivantes, il est nécessaire d'effectuer une unification générale car le premier accès à la cellule représentant la variable peut avoir lié la variable à n'importe quel type de terme (le type ne peut pas être déterminé statiquement).

Nous ne détaillerons pas la grande variété d'instructions d'unification (le détail peut être trouvé dans [AitK90]). Cela serait beaucoup trop fastidieux, d'autant plus que, pratiquement, le nombre de combinaisons augmente très rapidement lorsque l'on multiplie le nombre de types de base. De même, les techniques d'inférences de modes et de types peuvent générer beaucoup d'autres combinaisons rendant le jeu d'instruction

complexe. Dans ce qui suit la notation _xxx dénote les cas "variable, value, constant, list, structure" à partir desquels on peut obtenir les instructions WAM.

get_xxx les instructions **get** apparaissent uniquement en tête de clause elle servent à unifier les termes arguments aux termes apparaissant en tête de clause.

put_xxx les instructions **put** ne sont pas à proprement parler des instructions d'unification ; elles sont utilisées lors d'un appel de procédure pour charger les registres arguments.

unify_xxx Les instuctions **unify** correspondent au traitement des champs des structures. Elle fonctionnent selon deux modes : lecture ou écriture. Le mode écriture correspond à la construction d'une structure donnée. C'est toujours le mode des instructions **unify_xxx** suivant une instruction **put_xxx**. Le mode lecture correspond à la vérification de conformité d'une variable paramètre liée à une structure. Si cette variable est libre, le mode est alors écriture. Le mode est donc déterminé dynamiquement par l'instruction **get_xxx** précédant les instructions **unify_xxx**.

Dans les cas les plus généraux les instructions **get** et **unify** correspondent à l'opération d'unification.

2.4 Types et modèles de parallélisme en Prolog

Le parallélisme en Prolog peut prendre plusieurs formes: soit par l'exploitation du parallélisme intrinsèque du langage Prolog, soit par l'extension du langage par des constructeurs destinés à exprimer explicitement le parallélisme d'un algorithme. Dans ce premier cas, il s'agit en fait de proposer une stratégie d'exécution parallèle de Prolog indépendante du programme. Dans le second cas, il s'agit de proposer des extensions permettant au programmeur d'exprimer le parallélisme de son programme.

Ces deux approches ont pour origine un même objectif : l'accroissement de l'efficacité de Prolog par l'exploitation du parallélisme. Elles ont peu à peu divergé lorsque les chercheurs ont été confrontés aux problèmes pratiques posés par l'exploitation des machines multiprocesseurs.

2.4.1 Parallélisme explicite

Cette approche est celle des langages gardés ("Commited Choice Language"). Les langages logiques gardés [Shap89] sont de nouveaux langages pour la programmation concurrente. Dans ces langages, il s'agit d'exprimer un algorithme comme une ensemble de processus communiquants dans une optique similaire à celle formalisée dans le modèle CSP ou implanté dans le langage Occam. Ces langages ont été utilisés pour écrire les systèmes d'exploitations des machines dites de "$5^{ième}$ génération" étudiées par l'ICOT. L'approche des langages gardés a pour inconvénient de perdre une partie de l'aspect logique et surtout de reporter sur le programmeur, le problème de l'expression et de

la gestion du parallélisme. Parmi les problèmes liés à l'implantation de tels langages on peut remarquer que l'exécution de programmes écrits dans ces langages produit un graphe de processus évoluant dynamiquement et de taille non bornée a priori. De ce fait, l'implantation d'un tel langage sur une machine du type réseau de processeurs sans mémoire commune, de dimension finie par construction, pose un problème de placement très difficile à résoudre efficacement (il doit être résolu dynamiquement).

Enfin, citons un autre aspect du parallélisme explicite : on peut utiliser Prolog comme support d'évaluation séquentielle et introduire dans le langage ou sous forme de bibliothèque (prédicats prédéfinis), des mécanismes spécifiques du parallélisme comme par exemple les canaux de communication que l'on trouve dans un langage impératif parallèle comme Occam. CS-Prolog [Kacs90] , Shared-Prolog [Ambr90], Delta-Prolog [Pere84] sont exemplaires de ce type d'approche. La plupart des extensions proposées dans ce cadre permettent au programmeur de rendre complètement explicite le parallélisme. Elles correspondent à une volonté d'introduire dans Prolog les différents concepts connus dans les langages classiques. Ces langages s'éloignent de la programmation logique et nous semblent plus adaptés à la conception modulaire ou bien à une utilisation dans un cadre de système logique nécessairement distribué (exemple : ensemble de systèmes Prolog sur des stations de travail connectées par un réseau local).

2.4.2 Parallélisme implicite

La programmation logique "classique" se différencie des autres langages séquentiels par la séparation de la logique (la sémantique du programme) du contrôle de l'exécution de ce programme. Le contenu logique des programmes est à la charge des programmeurs tandis que la stratégie d'évaluation dépend du système.

Cette approche, très différente de celle que l'on vient de présenter, vise l'exploitation la puissance des machines parallèles en masquant autant que possible la complexité des problèmes de gestion du parallélisme. Dans cette optique le parallélisme n'est plus une fin en soi pour le programmeur, mais un moyen utilisé par le système pour augmenter l'efficacité à l'exécution en exploitant les ressources des architectures multiprocesseurs. Cette approche défendue par D.H.D. Warren et le groupe GIGALIPS, est également la notre.

On divise les différentes sources de parallélisme de la programmation logique en trois grandes classes : le parallélisme de l'unification (parallélisation de l'algorithme d'unification), le parallélisme OU qui permet l'évaluation en parallèle des clauses d'un même prédicat et le parallélisme ET qui consiste à évaluer en parallèle la conjonction des sous-buts d'une clause. Les stratégies d'évaluation parallèle permettent de suivre simultanément plusieurs chemins de l'arbre ET/OU. Idéalement, il s'agit de parcourir en parallèle tous les arcs de l'arbre ET/OU d'où les distinctions entre parallélisme ET, parallélisme OU et parallélisme ET/OU. Les difficultés techniques rencontrées ont conduit à un certain nombre de solutions partielles. On trouvera dans [Chas89b] une présentation de ces propositions et nous ne reprendrons ici que ce qui est nécessaire à l'explication de nos objectifs.

Nous présentons brièvement ces 3 classes de parallélisme puis nous examinons deux

familles de modèles d'exécution parallèle.

Parallélisme de l'unification

L'unification pourrait être une source de parallélisme dans la mesure où l'on pourrait envisager d'effectuer les unifications des sous-termes de structure en parallèle. Après ces unifications il faudrait alors vérifier la cohérence des substitutions produites par chacune des unifications de sous termes ; cette opération pourrait devenir coûteuse Les avis concernant la viabilité de l'exploitation de ce parallélisme sont partagés: certains considèrent que cette opération est intrinsèquement séquentielle [Dwor84], d'autres ont proposés une forme parallèle de l'algorithme [Mart82]. Il y a également eu quelques études de faisabilité pour des architectures de processeurs spécialisés réalisant l'unification.

Les études "théoriques" concernant l'unification parallèle sont éloignées du contexte pratique actuel dans lequel nous nous plaçons. En réalité, les programmes Prolog courants sont écrits de telle manière que la hauteur des arbres (termes) à unifier est souvent très faible et lorsque ce n'est pas le cas il s'agit, dans la plupart des cas, de structures linéaires (ex: listes) donc non parallélisables. De ce fait, la finesse du grain correspondant à ces "micro-unifications" rendent l'unification parallèle inexploitable de manière efficace sur les machines actuelles. De plus la faible fréquence d'occurence de configurations "intéressantes" peut difficilement justifier l'adjonction de matériel spécialisé.

Parallélisme OU

Dans l'interprètation procédurale, le parallélisme OU consiste en l'évaluation simultanée de plusieurs résolvantes qui peuvent être, elles mêmes, évaluées parallèlement ou séquentiellement (en utilisant le retour arrière).

Exemple:

```
grandpere( X,Z ) :- pere( X,Y ) , pere( Y,Z ).

pere( john   , philippe ).
pere( pierre , jean     ).
pere( jean   , marc     ).
```

Le but à "résoudre": quelles sont les personnes qui sont grand-père de marc:

```
:- grandpere( X , marc ).
```

donne lieu à l'évaluation séquentielle suivante:

```
resolvante 1 :  grandpere( X , marc )
resolvante 2 :  pere( X , Y ) , pere( Y , marc )
resolvante 3 :  pere( john , philippe ) , pere( philippe , marc )
```

Aucune des têtes de clause du programme n'étant unifiable au sous but de la résolvante 3, il y a retour arrière et le dernier point de choix, celui de la deuxième résolvante, est restauré.

```
resolvante 4 :  pere( jean , marc )
resolvante 5 :  vide --> SUCCES: X = pierre
```

Dans l'exemple précédent, le calcul peut être divisé en trois branches OU pour résoudre la deuxième résolvante en utilisant les trois clauses de la procédure pere. Les résolvantes suivantes des trois branches sont alors :

```
pere( philippe , marc )
pere( jean     , marc )
pere( marc     , marc )
```

•

Bien sur, seul la seconde branche aboutit à un succès (au pas suivant).

Si la division due au parallélisme OU intervient suffisamment tôt dans l'évaluation, les résolvantes restant à évaluer pour chacune des branches peuvent être importantes, produisant un gros grain de parallélisme.

Parallélisme ET

Le parallélisme ET consiste en l'évaluation simultanée de plusieurs sous-buts d'une clause. Il peut prendre plusieurs formes selon les règles de synchronisation des sous-buts lors de l'accès à une ressource partagée (variable logique).

En cas de variables partagées entre deux branches d'évaluation parallèle il faut donner un mécanisme permettant de gèrer la cohérence des accès. Dans le cas général on peut être amené à calculer la jointure d'ensembles de solutions. Cet algorithme complexe est difficile à mettre en oeuvre, toutes fois plusieurs solutions simplificatrices sont envisageables,

Le parallélisme ET indépendant consiste à restreindre l'évaluation parallèle au cas où les sous buts sont indépendants c'est-à-dire aux sous-buts n'ayant pas de variables en commun. Le principe du parallélisme ET indépendant est simple mais la détection de cette indépendance n'est pas facile comme le montre Conery [Cone83].

Le parallélisme ET orienté consiste à "orienter" l'opérateur de conjonction en définissant un prédicat producteur et des prédicats consommateurs relativement à une variable partagée.

On peut également restreindre le parallélisme ET au cas d'évaluation déterministe. Ce type de parallélisme peut coexister avec le parallélisme OU (exemple [AURORA]): du fait du déterminisme, l'absence de parallélisme OU est garantie dans les tâches ET parallèles; par contre les tâches OU parallèles peuvent créer des tâches ET parallèles.

Si les principaux problèmes du parallélisme ET sont relativement bien cernés à l'heure actuelle, l'obtention d'une solution efficace à l'exécution reste un problème ouvert. Dans notre système OPERA nous ne prenons en compte que le OU parallélisme.

2.4.3 Modèle de parallélisme massif

Conery [Cone85] présente le modèle théorique ET/OU pour l'exécution de programmes logiques purs. Dans son modèle, Conery propose de créer autant de processus OU qu'il y a de sous-but à évaluer dans une clause et un processus ET par clause pour gèrer la conjonction. Le réseau de processus engendré par ce modèle peut être représenté par l'arbre ET/OU si l'on considère que chaque nœud est un processus. Le principe de fonctionnement est le suivant: lorsqu'une clauses est évaluée avec succès, les variables instanciées sont transmises au processus père qui les transmet alors au processus OU chargé de collecter l'ensemble des solutions. Le parallélisme OU provient de l'exécution en parallèle des clauses définissant un prédicat; l'exécution pipelinée des sous-buts d'une clause définit le parallélisme de flot. Des algorithmes complexes doivent être exécutés pour déterminer l'ordre d'évaluation des sous-buts pour éliminer les problèmes liés au partage de variables. Une simulation de ce modèle montre des performances très faibles avec une courbe d'accélération qui s'aplatit à partir d'une demi-douzaine de processeurs.

L'inconvénient majeur des modèles à parallélisme massif est leur inadéquation aux machines actuelles. Cela provient du fait qu'ils ont étés étudiés de manière théorique avec l'objectif de faire ressortir le maximum de parallélisme d'un programme sans se préoccuper vraiment de la manière de "plier" efficacement ce graphe de processus résultant sur des machines réelles.

Une des raisons de l'inefficacité de ces modèles est l'exploitation d'un grain de parallélisme beaucoup trop fin par rapport aux machines multiprocesseurs actuelles. Ces modèles engendrent une explosion combinatoire du nombre de processus car ils correspondent à un parcours en largeur de l'arbre ET/OU. On pourrait penser que des machines comportant plusieurs milliers de processeurs conviendraient mieux à ce type de modèles. Cependant une forte proportion de ces nombreux processus ne sont pas activables (par exemple les nœuds pères en sont attente du résultat calculé par les nœuds fils) et il est probable que, compte tenu de leurs coûts d'installation et de gestion (ordonnancement, synchronisation...) devant leur durée d'exécution, le résultat ne soit pas meilleur. Chaque fois qu'il y a plus de processus que de processeurs, il est préférable de privilégier la stratégie séquentielle sur la stratégie parallèle pour limiter le recours à la multiprogrammation car le coût d'un changement de contexte est généralement moins efficace que l'exécution en séquence. De plus la gestion mémoire liée au développement

désordonné des branches de l'arbre ne peut être que dynamique (allocation en tas classique) donc très coûteuse.

Un autre inconvénient de ces modèles est de nécessiter un très grand nombre de messages de taille variable non bornée (les processus n'ont pas d'espace d'adressage commun) car les termes sont copiés d'un processus à l'autre. La gestion de l'ensemble de ces copies nécessite une capacité mémoire importante (parcours en largeur) et bien supérieure à celle dont on dispose actuellement. La gestion des communications entre processeurs étant plus coûteuse (de plusieurs ordre de grandeur) que l'accès unitaire à une mémoire, la copie de termes entre processeur ne peut être rentable que pour des transfert impliquant de gros calculs.

En résumé les modèles à parallélisme massif ont pour principal intérêt de mettre en évidence les problèmes théoriques et proposer des éléments de solution. Leur implantation efficace sur des machines réelles (même à moyen terme) est encore du domaine de la recherche prospective. Il peuvent fournir une métrique de référence aux systèmes réels en donnant le degré maximal de parallélisme associé à un programme.

2.4.4 Modèle multi-séquentiel OU

Le modèle multi-séquentiel a été développé pour tenir compte des caractéristiques des architectures existantes. La stratégie multi-séquentielle a pour but de contrôler la création de processus en fonction des ressources disponibles (processeurs, mémoires). Le principe du contrôle est simple: le degré de parallélisme maximum, c'est à dire le nombre de processus actifs à un instant, est borné. Dans ce modèle, un programme Prolog est exécuté avec une stratégie parallèle (en largeur d'abord) tant qu'il existe des ressources CPU (et mémoire) libres et suivant la stratégie séquentielle classique lorsque tous les processeurs sont déjà occupés. Ainsi la l'évaluation d'un programme Prolog se fait par mixage d'exécutions séquentielles et parallèles selon la disponibilité des ressources.

Le système est alors vu comme un nombre fini de processus ou machine Prolog (géneralement une machine Prolog par processeur matériel) tels que:

1. chaque processus poursuit une résolution séquentielle indépendante des autres processus actifs,

2. chaque processus produit des "points de choix" correspondant aux résolvantes suspendues des différents noeuds "ou" rencontrés,

3. un processus inactif acquiert du travail auprès d'un processus actif possédant un ou plusieurs points de choix en attente,

4. la fourniture d'un travail, c'est-à-dire le transfert d'une ou plusieurs résolvantes, est la seule communication entre deux processus,

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

L'évaluation d'un programme correspond à un parcours d'arbre. Les stratégies de parcours possibles sont nombreuses mais le parcours "en profondeur d'abord" reste le plus efficace car il minimise les besoins en mémoire et reste, en ce domaine, le plus proche de la gestion en pile pour laquelle les processeurs actuels sont efficaces.

Sa transposition dans le cadre parallèle s'appelle multi-séquentiel pour rappeler que chaque unité effectue un parcours "en profondeur d'abord" du sous-arbre qu'il évalue. La figure ci-après montre une manière de partager l'arbre OU du programme à évaluer en plusieurs sous-arbres; elle est utilisée dans le premier prototype de notre système OPERA.

Chaque processeur se charge d'une ou plusieurs parties de l'arbre et effectue un parcours conforme à la stratégie classique.

Ce modèle est intéressant pour plusieurs raisons:

- On peut utiliser, pour la résolution séquentielle, les mises en oeuvres les plus efficaces développées pour les machines séquentielles classiques et en particulier la WAM.
- Le parallélisme n'est utilisé que s'il est susceptible d'apporter un gain de performance. Cet objectif est beaucoup plus facile à atteindre que dans le cas de création massive de processus puisque l'on tient compte des ressources disponibles.
- Le coût de gestion du parallélisme OU est celui des échanges de travaux: il n'y a plus de communication après l'installation d'une tâche. Il est possible d'implanter très efficacement un système multiséquentiel exploitant le parallélisme OU sur des machines ne disposant pas de mémoire commune en raison du faible nombre des synchronisations nécessaires.
- La recherche de travail peut être à la charge des processus inactifs de sorte que cela ne ralentisse pas les processus actifs.

Il présente néanmoins l'inconvénient de ne laisser au programmeur aucune possibilité de contrôle du parallélisme. L'utilisateur ne peut pas prévoir simplement la manière dont va se développer l'exécution de son programme ce qui peut être gênant lors d'une mise au point de bas niveau.

La gestion du parallélisme ET nécessite toujours de nombreuses synchronisations et seul le parallélisme ET restreint déterministe semble pouvoir coexister efficacement avec le parallélisme OU (car le grain peut rester suffisamment élevé).

Notre projet OPERA se place donc dans la classes des systèmes multi-séquentiels à parallélisme OU implicite pour machine à mémoire distribuée. Nous avons choisi d'implémenter un premier prototype sur une architecture à base de Transputer en mettant l'accent sur l'aspect distribution (communication par messages) et l'aspect asynchronisme lié au couplage faible des processeurs.

ARBRE ET/OU

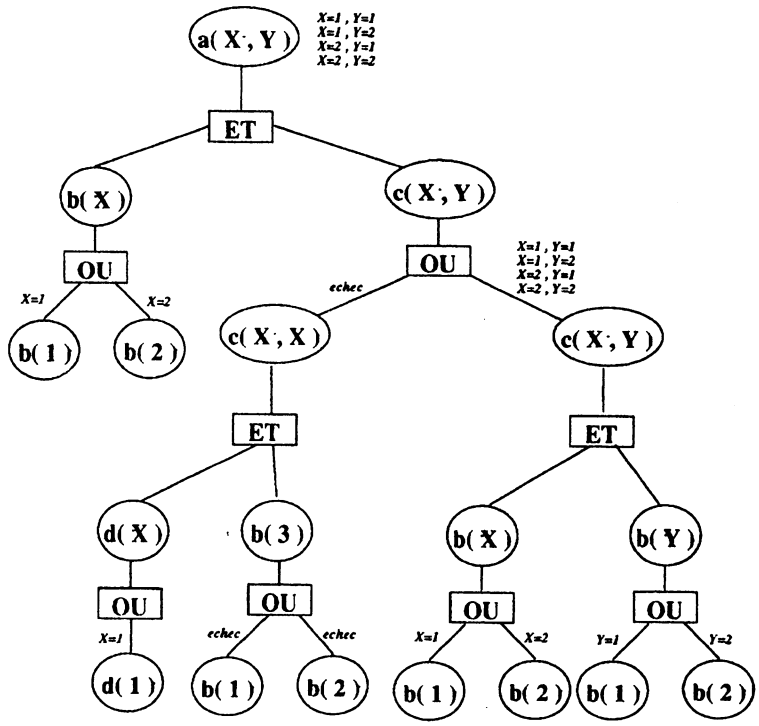


Figure 2.1 : Arbre ET/OU :

Chaque nœud ET représente une conjonction de sous buts, chaque nœud OU représente un ensemble d'alternatives possibles. Les arcs représentent les dérivations à effectuer. L'arbre ET/OU représente l'ensemble des dérivations possibles: il peut y avoir des branches de longueur infinie.

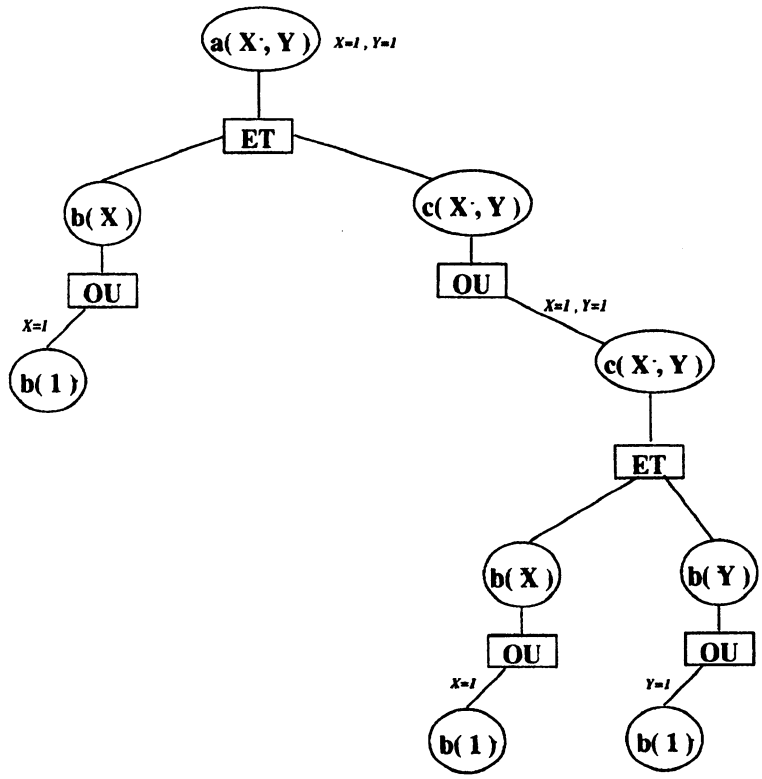


Figure 2.2 : Sous-Arbre solution

Cette figure représente le sous-arbre de l'arbre ET/OU (figure 2.1) dont le parcours fournit la subtitution solution $(X, 1), (Y, 1)$.

STRATEGIE DE PARCOURS DE L'ARBRE ET/OU

UTILISEE PAR PROLOG SEQUENTIEL

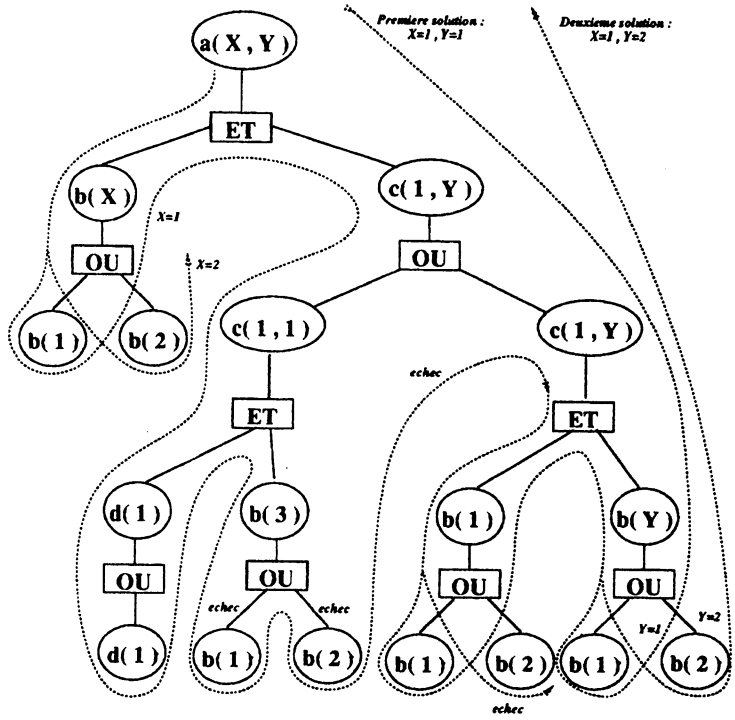
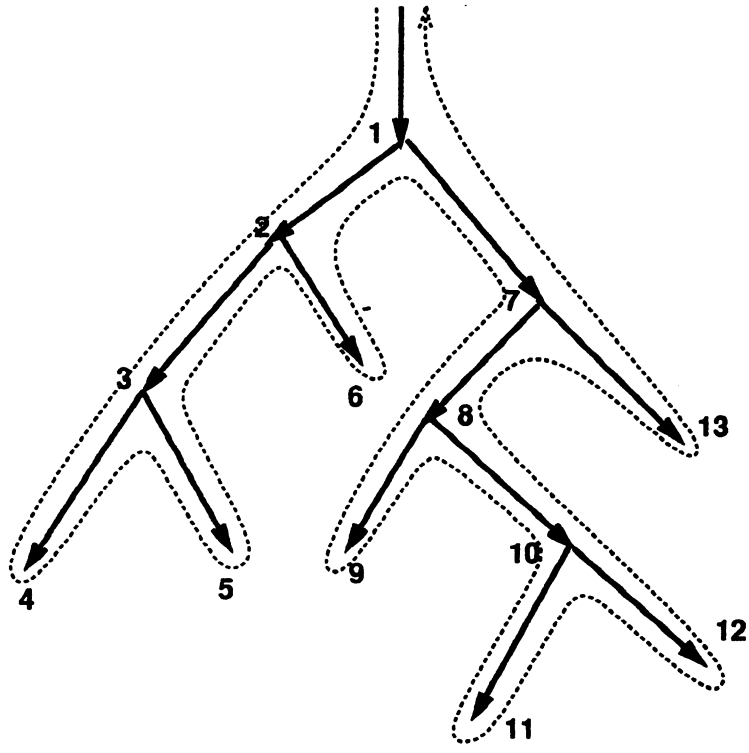


Figure 2.3 : Stratégie "*en profondeur d'abord*"

L'arbre en trait pointillé correspond à l'arbre de recherche des solutions selon la stratégie "en profondeur d'abord" utilisée par la plupart des systèmes Prolog séquentiel. Notons la différence entre la relation de filiation des nœuds OU de l'arbre ET/OU et celle des points de choix de l'arbre de recherche (qui dépend de la stratégie utilisée)




```
1 processeur: 1,2,3 ... 12,13

3 processeurs A,B,C:

        A:  1, 2, 3, 4, 5,
        B:     7, 8, 9,10,11
        C:        6,13,  ,12
```

Figure 2.4: Stratégie multi-séquentielle

Le parcours de l'arbre par un seul processeur selon la stratégie séquentielle correspond à 1,2,3,4,5,6,7,8,9,10,11,12,13. Nous donnons également un exemple du même parcours effectué par trois processeurs A, B, et C selon la stratégie multiséquentielle.

Chapitre 3

MODELES MULTI-SEQUENTIELS "OU"

3.1 Structure du chapitre

Ce chapitre est consacré exlusivement aux systèmes Prolog "multi-séquentiels OU". Il précise la place de notre système et justifie les choix que nous avons effectués dans ce projet. Le lecteur notera la dualité constante dans notre approche, entre l'aspect langage, selon lequel l'évaluation se résume en un parcours d'arbre, et l'aspect architectural du système, selon lequel l'exécution d'un programme est un problème d'allocation de ressources (CPU et mémoire). Cette prise en compte des propriétés du matériel cible à chaque niveau de conception du système logique parallèle est caractéristique des modèles multi-séquentiels. L'aspect architectural est souvent ignoré dans les autres modèles d'exécutions (mis à part les projets de l'ICOT) ce qui explique la faiblesse de leurs performances face aux systèmes multi-séquentiels.

La conception d'un système multi-séquentiel consiste à fournir un moyen de projeter ou parcourir l'arbre de recherche sur l'architecture physique. Dans la section 3.2, nous examinerons deux cas de figures selon que l'on dispose d'une machine cible à mémoire commune ou d'une architecture n'en ayant pas. Ayant choisi un mode de représentation des variables logiques pour une architecture donnée, il faut fournir un moyen de gérer les liaisons multiples avec deux objectifs contradictoires: profiter de la technique séquentielle (liaison unique) pour obtenir des performances élevées en phase séquentielle et minimiser le coût du parallélisme (qui, dans le cas du parallélisme OU, se résume au coût de création d'un nouveau processus). Diverses méthodes de représentation mémoire sont exposées et comparées dans la section 3.3.

3.2 Projection de l'arbre de recherche sur la mémoire

Cette section présente les différentes méthodes de projection du parcours de l'arbre de recherche sur la mémoire. La représentation de l'arbre est liée à l'architecture de la mémoire utilisée ; la stratégie d'allocation de cette mémoire dépend de la stratégie de parcours de l'arbre de recherche.

Nous rappelons les principes de la méthode séquentielle sur une machine mono-processeur puis nous présentons son extension aux machines multi-processeurs à mémoire commune. Enfin nous distinguons sur les machines multi-processeurs à mémoire distribuée les méthodes de "copie après calcul" et de "recalcul".

3.2.1 Méthode séquentielle

La méthode de projection de l'arbre de recherche sur la mémoire utilisée en Prolog séquentiel est directement liée à l'interprétation procédurale qui fixe le parcours de l'arbre "en profondeur d'abord". Pour une machine mono-processeur à espace d'adressage unique, il suffit donc de projeter un chemin de l'arbre, à un instant donné (i.e. chaque état de calcul), sur l'espace d'adressage. La technique d'allocation en pile, présentée au chapitre compilation, se révèle très efficace car il suffit de disposer d'un simple registre de base par pile dont l'incrémentation (resp. décrementation) suffit pour allouer (resp. libérer) de la mémoire, c'est-à-dire descendre ou remonter dans l'arbre.

3.2.2 Méthodes "naïves" pour multi-processeurs à mémoire commune

Lorsque l'on dispose d'un espace d'adressage unique accessible par plusieurs processeurs (cas des machines à mémoire commune), on peut chercher à représenter l'arbre d'évaluation en évitant les redondances : chaque chemin menant d'une feuille à la racine de l'arbre a une portion en commun avec les autres chemins (éventuellement réduite à la racine). Ce chemin représente l'ensemble des données potentiellement nécessaires à la poursuite de l'évaluation séquentielle (ensemble des piles de la WAM), il est tout naturel de tirer parti de l'adressage global à tous les processeurs pour que ceux-ci puissent partager les zones mémoires contenant la portion de chemin commune.

a) allocation en tas

Une première solution consiste à utiliser un mécanisme de gestion mémoire totalement dynamique (en tas) pour l'allocation des environs de procédure. L'inconvénient majeur de ce type d'allocation est son coût elevé (relativement à sa fréquence d'utilisation) et les problèmes de morcellement de la mémoire qu'il implique. Un inconvénient de la gestion en tas pour Prolog est le suivant : l'ancienneté des occurences de variables ne peut plus être déterminée à partir de l'ordre des adresses de ces occurences, il devient

nécessaire de dater les variables et d'utiliser des mécanismes beaucoup plus coûteux qu'une simple comparaison d'adresses.

L'allocation en tas est donc très coûteuse en temps, en mémoire (fragmentation) et elle est incompatible avec l'exécution séquentielle de la WAM. Elle n'est donc pas utilisée en multi-séquentiel. Cependant on peut remarquer qu'elle offre un degré de liberté supplémentaire dans le choix de la stratégie d'évaluation ; c'est pourquoi cette méthode très simple est utilisée dans bon nombre de prototypes de modèles "théoriques".

b) pré-allocation

Une autre solution est d'effectuer une préallocation en effectuant une projection "horizontale" de l'arbre de recherche sur la mémoire, c'est à dire en effectuant une allocation en "piles creuses" dès la compilation.

Cela n'est possible que lorsque l'arbre des processus (ou plus exactement l'arbre des appels) est fini et connu statiquement. C'est le cas dans un langage comme OCCAM2. La figure 3.1 illustre ce type d'allocation.

En Prolog, dès qu'il existe des clauses récursives dans le programme, l'arbre n'est plus fini. Dans ce cas, le compilateur Prolog n'est plus en mesure de déterminer statiquement la taille des ressources mémoires nécessaire à l'évaluation d'un but et les ressources allouées à un processus père peuvent ne pas être suffisante pour l'évaluation de tous ces processus fils.

Pour résoudre ce problème de récursivité, on peut allouer dynamiquement sur une pile tous les environs de clause dès l'entrée du prédicat (voir Figure 3.2). Statique ou dynamique, cette solution correspond à un parcours de l'arbre OU en largeur ; elle nécessite donc beaucoup de mémoire. De plus, on ne peut libérer les environs de clauses que dans l'ordre où ils ont été alloués. Dans le plus mauvais des cas, toutes les branches OU d'un nœud doivent avoir été évaluées avant de pouvoir libérer ces environs (elles doivent se synchroniser pour que la dernière à terminer libère l'espace).

Cette technique d'allocation est efficace du point de vue temps d'exécution car il n'y a pas de recherche de bloc libre mais elle est coûteuse en espace mémoire. Ceci est contradictoire avec l'objectif du modèle multi-séquentiel d'une bonne gestion des ressources.

3.2.3 Méthode des piles cactus pour Prolog

C'est un compromis des deux méthodes précédentes qui est retenu dans la plupart des systèmes multi-séquentiels OU pour machines à mémoire commune : la méthode des "piles cactus" [1]. L'idée sous-jacente à cette méthode de représentation de l'arbre de

[1] La notion de "pile cactus" est ancienne : historiquement le problème des piles multiples interdépendantes provient des langages à coroutines / processus (algol68 - simula67 ...). La "pile cactus" est la structure de données représentant les relations entres les environs des procédures / coroutines / processus. La méthode décrite ici, est une technique particulière de réalisation dans le contexte de Prolog sur les machines multiprocesseurs à mémoire partagée.

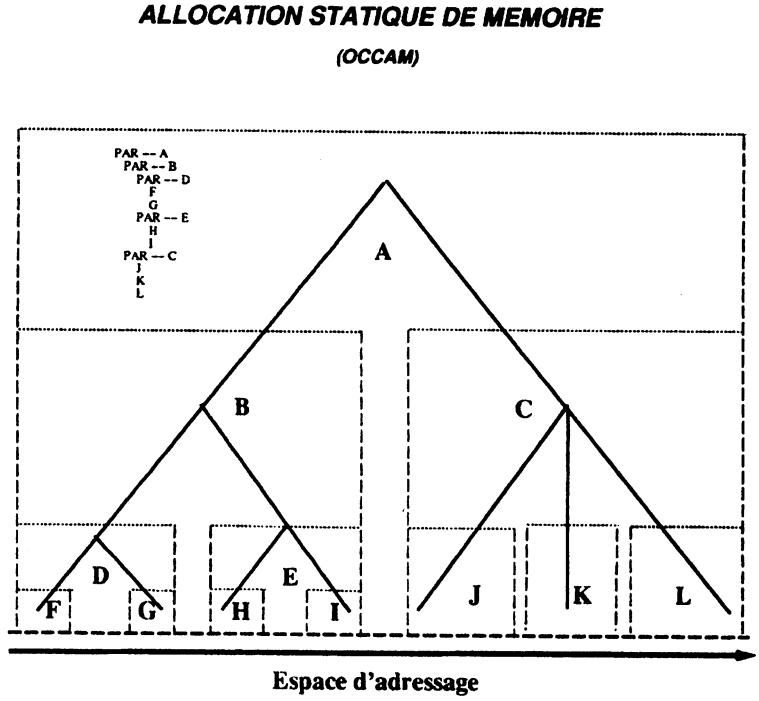


Figure 3.1 : pré-allocation pour OCCAM

En OCCAM l'arbre des processus défini par des compositions parallèles imbriquées est connu dès la compilation. Il est fini car il n'y a pas de récursivité. Le compilateur OCCAM est donc capable de synthétiser la taille des ressources mémoire nécessaires à l'évaluation d'un arbre à partir des tailles calculées pour les sous-arbres. Les ressources mémoire d'un processus fils (sa pile d'évaluation) sont donc allouées dans l'espace alloué au processus père. L'ensemble des piles d'évaluation correspond à la "projection" de l'arbre des processus OCCAM.

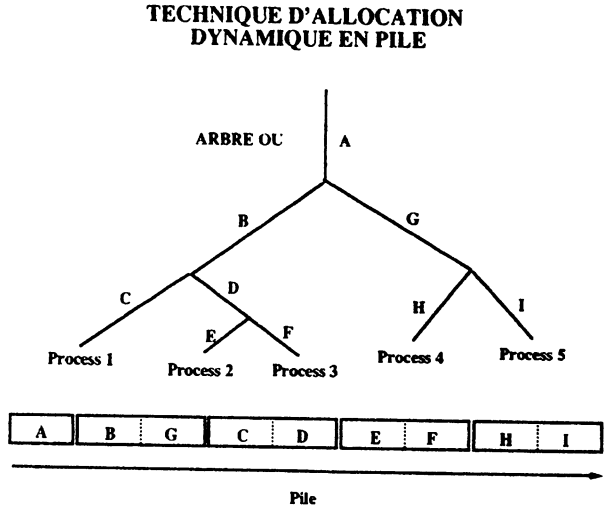


Figure 3.2: Allocation dynamique

Dès le début de l'exécution de la clause A l'espace nécessaire aus sous-buts B et G, de la clause A est alloué sur la pile. La stratégie d'évaluation séquentielle "en profondeur d'abord" implique que le sous-arbre issu de B soit totalement évalué avant de passer à la branche G. On alloue donc l'espace pour (C,D) et (E,F) avant (H,I). Dans le cas d'une stratégie multi-séquentielle les sous-arbres issus de B et G évoluent en parallèle. La terminaison de (C,D,E,F) ne permet pas de récupérer la mémoire immédiatement: il faut attendre que (H,I) termine puisque seul l'espace qui est sur le sommet de pile peut être récupéré lorsqu'il devient inutile.

recherche est basée sur la constatation suivante :

une fois en charge d'une branche, un processeur la parcourt jusqu'à sa fin puis effectue un retour arrière avant d'emprunter une voie alternative. On considère donc que le fonctionnement "séquentiel" standard est plus fréquent que le comportement parallèle qui provoque une "fourche de pile".

On partitionne (statiquement) l'espace d'adressage global de taille EG en P zones de taille EG/P s'il y a P processeurs. Chaque processeur possède alors sa zone d'allocation privée qui reste accessible en lecture/écriture à n'importe quel autre processeur. Du point de vue fonctionnel, une fois qu'une donnée a été allouée dans une zone privée par le processeur auquel elle est assignée selon la technique de la WAM (gestion en pile), elle peut être référencée par les processeurs qui prennent en charge l'évaluation des branches issues de la branche allocatrice.

Pour les exemples, on peut simplifier notre propos en considérant que chaque unité de travail (WAM) ne comporte qu'une pile représentée par une suite de segments (portion de chemin dans l'arbre d'évaluation).

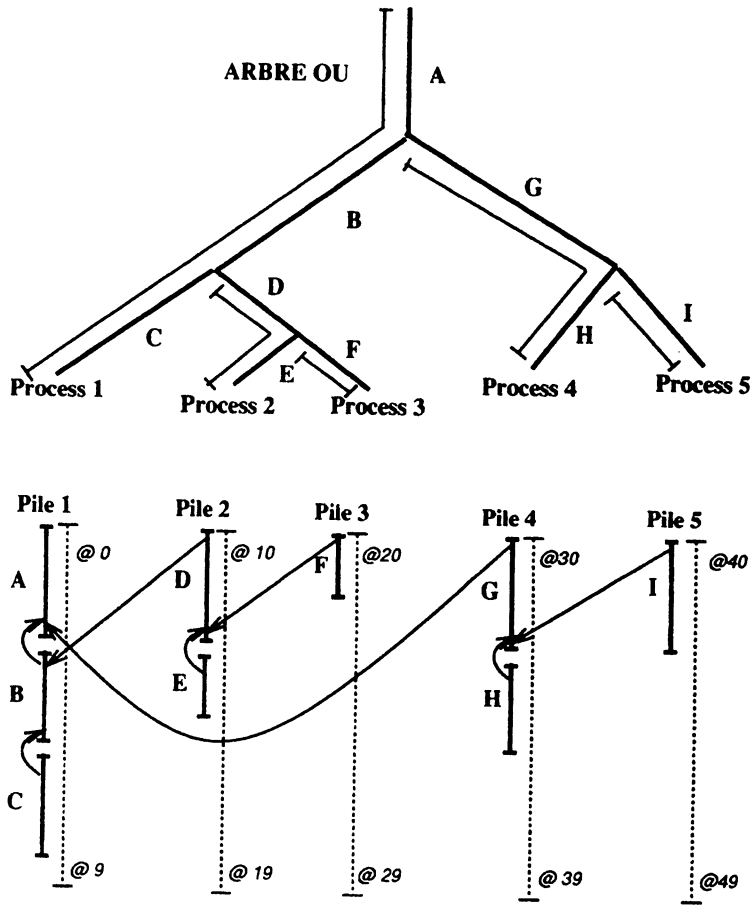
Ce partage de pile est économique du point de vue de la taille mémoire allouée aux données utiles et il ne pose pas de problème pour les accès en mode lecture. Par contre, pour les accès aux zones partagées en mode écriture, le problème de la cœxistence de plusieurs valeurs simultanément pour une même variable nécessite un mécanisme d'accès particulier. Différentes solutions à ce problème sont exposées dans la section 3.3.

Un autre inconvénient de cette technique de partage mérite d'être cité : lorsque un processeur termine l'évaluation d'une branche, il cherche un autre nœud (point de choix) dont sont issues des branches non encore évaluées, puis effectue l'évaluation correspondante. Ce choix relève de l'ordonnancement des tâches (parcours des sous-arbres).

Selon les opérations de l'ordonnancement, on peut se retrouver dans le cas suivant : une donnée peut ne plus être nécessaire au processus qui l'a allouée sans pour autant que l'on puisse désallouer la zone qu'elle occupe, comme le voudrait la gestion locale en pile (segment D sur la figure 3.4). En effet, de par le mécanisme de partage de pile décrit au paragraphe précédent, d'autres processus peuvent encore avoir besoin de cette donnée qui doit donc être conservée. En d'autres termes, la durée de vie d'une donnée, déterminée par l'ensemble de ses références, peut dépasser la durée de vie du "processus" qui est responsable de leur allocation. La solution consiste à ne pas libérer ce sommet de pile et à considérer que la nouvelle évaluation se fait sur une pile dont le fond correspond à cette limite. L'inconvénient de cette solution est que ces zones finiront par ne plus être utiles à d'autres processus sans pouvoir être réallouées car elles ne sont plus sur le sommet de pile. Le phénomène de "**trou noir**", est un inconvénient important de cette méthode car la dégradation progressive des ressources mémoire peut provoquer une saturation "artificielle" de l'espace disponible pour les programmes dont l'arbre d'évaluation a un facteur de branchement important (ceux là même qui, justement, ont le degré de parallélisme OU le plus intéressant à exploiter du point de vue du temps d'exécution).

Enfin, notons que le partitionnement de l'espace d'adressage global ne permet plus

TECHNIQUE DES PILES CACTUS (ESPACE D'ADRESSAGE GLOBAL)



Avec 5 processeurs

Figure 3.3: Partage de mémoire

Chaque processeur peut accèder à n'importe quelle adresse de la mémoire globale. Celle-ci est divisée en autant de zones qu'il y a de processeurs. Un processeur n'effectue d'allocation de mémoire et de modification (accès en écriture) que dans la zone qui lui est affectée ; cependant il peut accèder (en lecture) à d'autres zones (les arcs flèchés illustrent ce type de références).

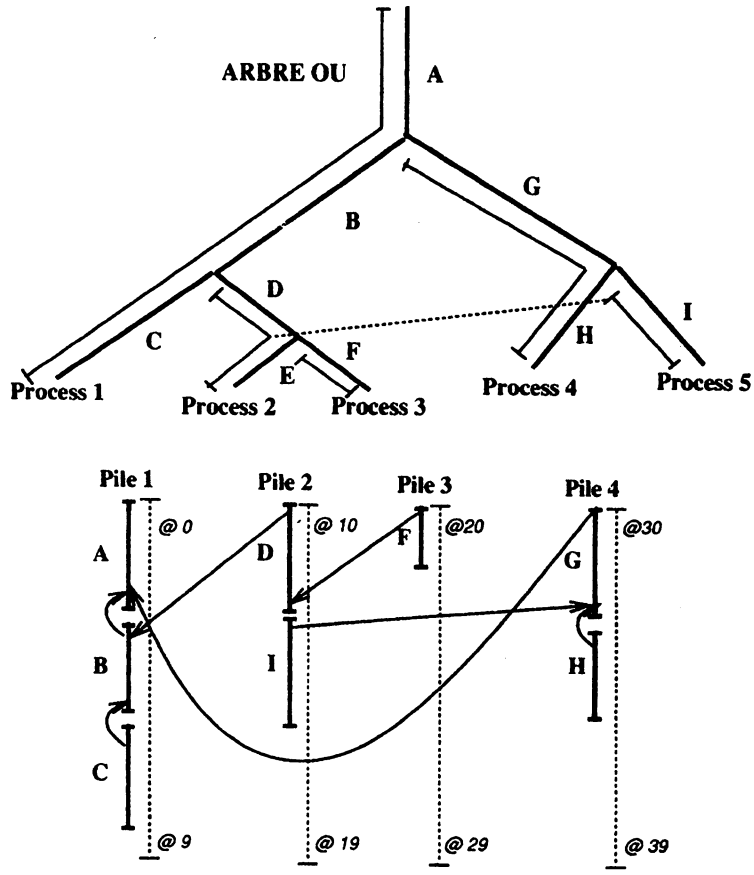


Figure 3.4 : Les trous noirs

Lorsqu'il y a moins de processeurs que de branches dans l'arbre il peut y avoir formation de trou noir : la figure ci-dessus illustre le cas où le processeur 2 termine le processus 2 et prend en charge l'évaluation du processus 5. Bien que le segment D ne soit plus utile au processeur 2 il ne peut en récupérer l'espace (alors que c'est possible pour E) car D est encore référencé par F. Le processeur 2 empile donc I "au dessus" de D. Un trou noir se formera lorsque le processus 3 terminera en libérant F et D.

de conserver l'équivalence de la relation d'ancienneté et de la relation d'ordre sur les adresses lorsqu'il s'agit de références entre deux piles différentes. Sur la figure 3.4 on a représenté un cas d'ordonnancement pour lequel on référence un objet crée antérieurement et qui se trouve à une adresse supérieure (référence I → G). La solution à ce problème est exposée dans la section traitant de la gestion des liaisons multiples.

En résumé, cette méthode permet un gain de temps en privilégiant l'allocation en pile aux prix du :

- risque de "trous noirs"
- risque de blocage par saturation d'un processeur

3.2.4 Méthode par duplication

Cette méthode est celle utilisée dans les machines parallèles faiblement couplées. Elle suppose que l'on dispose de P unités de travail identiques, possédant chacune un processeur et sa propre mémoire. Contrairement à la technique des piles cactus qui factorise des données au prix d'une gestion plus complexe de l'espace d'adressage global, la technique de duplication des piles factorise la fonction d'adressage (elle est la même pour tous les espaces d'adressage locaux) au prix de la duplication des données.

Chaque mémoire locale est donc adressée uniquement par le processeur correspondant. Chaque unités de travail est une WAM classique, Chaque mémoire locale contient donc un chemin de l'arbre de recherche. L'exécution de parties séquentielles est donc très efficace car très proche de ce que l'on sait faire de mieux en Prolog séquentiel. Lorsqu'un processeur devient oisif, il faut lui trouver un autre sous arbre à évaluer : cela nécessite de le remettre dans l'état par lequel est passé un autre processeur (partie commune de l'arbre). Deux solutions sont possibles :

1. Soit le processeur oisif constitue son nouvel état "initial" à partir des piles du processeur actif qu'il décharge d'un groupe de tâches (nœuds OU). Cette technique, basée sur la copie d'environnement, est bien adaptée aux systèmes communiquant par messages.

2. Soit le processeur oisif calcule son nouvel état "initial" : il effectue le même calcul qu'un autre processeur jusquá se retrouver dans le même état que cet autre processeur. Cette technique a pour intérêt de ne nécessiter aucune communication directe entre les unités de travail. Seule une unité de contrôle spécialisée dans l'allocation des chemins de l'arbre d'évaluation aux unités de travail doit maintenir une vision globale du système afin d'affecter les tâches aux processeurs.

Nous allons examiner plus précisement les avantages et les inconvénients respectifs de ces deux techniques.

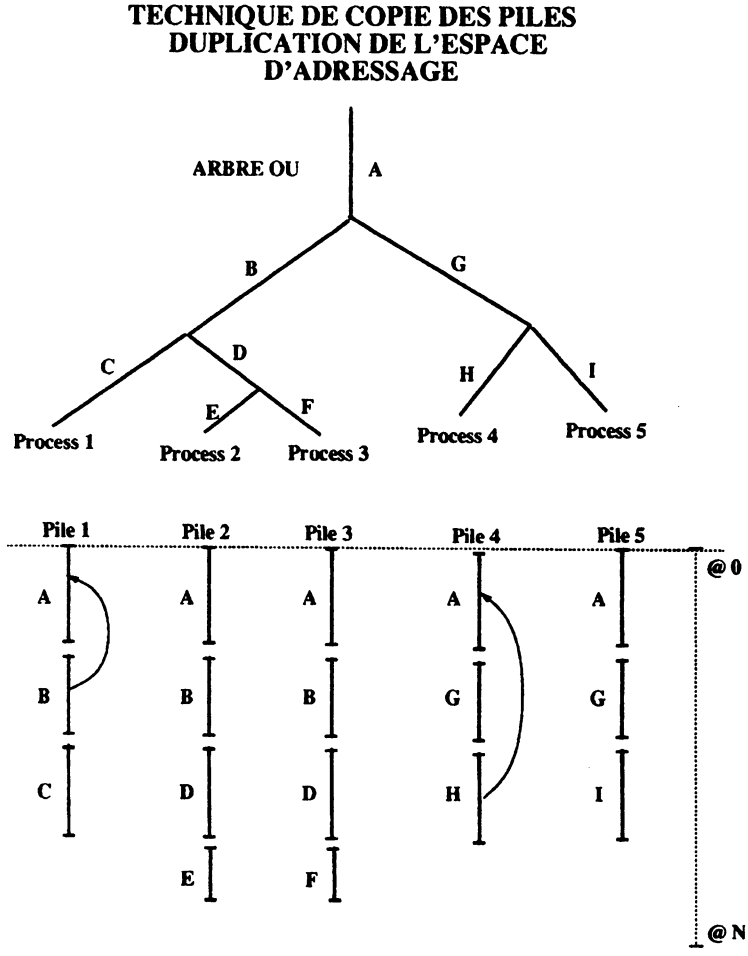


Figure 3.5 : copie de mémoire

Chaque processeur possède sa propre mémoire locale et est le seul a pouvoir l'adresser. Chaque mémoire locale contient un seul chemin de l'arbre OU (celui qui est en cours d'évaluation). Chaque mémoire locale est gérée de la même façon que dans la WAM séquentielle. Les portions communes à plusieurs chemins en cours d'évaluation correspondent à autant d'instances dans les mémoires locales. Exemple : le segment B est dupliqué dans 3 mémoires locales.

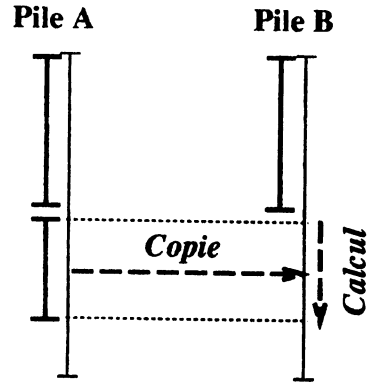


Figure 3.6 : techniques de duplication : copie de portion de piles ou recalcul

Pour obtenir dans la pile B le même état que la pile A, deux solutions sont possibles : recopier les portions de piles manquantes sur B (transfert par messages) ou bien effectuer les calculs amenant à ce nouvel état de pile à partir de l'état courant (pas d'intervention de A).

a) Technique de copie (duplication de données)

Pour cette première solution, la fonction d'adressage est la même pour tous les processeurs afin de faciliter la copie (pas de relogement explicite de données). Le coût du parallélisme est payé au moment de cette copie car, par la suite, les deux processus OU n'ont plus de communication.

Le coût principal du parallélisme réside dans la copie elle même ; il est fortement dépendant de l'architecture matérielle et, en particulier, du médium de communication employé. La copie naïve correspond à une sur-évaluation de l'ensemble des données nécessaires : on copie l'ensemble des piles du processus père (opération similaire au fork d'unix). On peut optimiser la technique de la copie naïve en profitant du partage logique, ce qui permet d'éviter de recopier les parties identiques déjà copiées antérieurement (cette technique dite de "copie incrémentale" est utilisée dans le modèle MUSE [Ali90]). On peut également restreindre la copie aux termes potentiellement accessibles c'est à dire aux termes arguments. L'inconvénient de cette dernière méthode est de nécessiter une unité de transfert qui sache décoder la représentation des termes ce qui est plus complexe et plus coûteux (du point de vue réalisation dans le matériel) qu'une simple unité de DMA qui effectue une copie de bloc sans en interpréter le contenu. Cette dépendance entre la technique de copie et le codage des termes est un handicap sérieux pour une implantation matérielle car elle fige la nature des objets manipulés et rend caduque une architecture matérielle intégrant une unité de copie dès que cette représentation change.

Nombreuses sont les possibilités d'optimisations de la technique de copie, quelques unes sont détaillées tout au long des chapitres consacrés à OPERA car la copie est la technique retenue dans notre modèle.

b) Technique de recalcul (duplication de calcul)

Les unités de travail sont des WAM à peine modifiées car aucune interface entre les unités de travail n'est nécessaire. C'est la solution la plus proche des systèmes séquentiels donc la plus efficace sur les portions d'évaluation séquentielle. Cependant, le choix de la branche à parcourir pose un problème sérieux. On doit recourir à un mécanisme global pour connaître les chemins non encore évalués. Dans le modèle DelPhi [Cloc88], ce mécanisme est appelé **oracle** en référence à l'oracle de Delphe : un contrôleur global donne à chaque unité de travail le chemin à suivre dans l'arbre sous forme d'une chaîne de bits de longueur variable (ce qui implique une mise du programme sous forme d'arbre binaire OU). L'idée de séparer totalement le contrôle du parcours et le mécanisme d'évaluation déterministe est séduisante car elle permet de simplifier considérablement celui-ci (pas de gestion du non déterminisme : pas de retour arrière donc pas de pile trainée). De plus, elle permet de faire varier la stratégie de contrôle sans modifier l'unité de travail. Cependant cette simplification apparente masque la grande complexité du système de contrôle qui doit alors effectuer des opérations analogues avec beaucoup moins d'informations donc essentiellement en appliquant des heuristiques. Cette complexité peut accroître l'effet de congestion lorsque l'on utilise un grand nombre de processeurs dans la mesure où le système de contrôle est centralisé. De plus, les opérations "non logiques" comme la coupure, la négation, les entrées/sorties et les effets de bord en général, ne sont pas pris en compte dans ce modèle et leur introduction peut remettre en cause les bases de celui-ci. En conclusion, le bien fondé de cette idée reste à démontrer car le modèle DelPhi n'a pas encore été implémenté sur une machine parallèle.

c) Comparaison des deux techniques de duplication

Comparons les deux familles de solutions. La technique de copie implique une exécution séquentielle plus coûteuse que la technique de recalcul car elle doit maintenir les structures de données permettant d'effectuer le retour arrière. Du point de vue du nombre de duplications de données (méthode naïve de copie complète) et du nombre de recalculs effectués, les deux méthodes se valent. Cependant la copie peut être optimisée (copie incrémentale et copie paresseuse) alors que le recalcul doit être complet. De manière générale, pour construire l'état de mémoire "initial" d'un nouveau processus, le nombre d'accès mémoire effectués par le recalcul est supérieur à celui que l'on obtient par la technique de copie de bloc (ce dernier est de l'ordre de la taille du bloc) car les accès dus aux calculs temporaires et les accès multiples à une même donnée sont éliminés dans la technique de copie après calcul. Pour cette raison la technique de duplication de calcul nous semble être beaucoup moins efficace que la technique de copie après calcul.

La duplication de calcul est plus efficace que la copie lorsque deux processeurs sont inactifs et qu'un seul travail est disponible (dans tout le système) à cet instant : plutôt que d'attendre que l'un d'eux amorce le travail et produise un point de choix puis que l'on copie cet état sur l'autre processeur, il est préférable de calculer cet état simultanément sur les deux processeurs puisqu'aucune autre tâche n'est disponible à cet instant. De manière générale, la duplication de calcul ne nous parait intéressante que lorsqu'elle n'implique pas une concurrence d'accès à une ressource (en particulier à un processeur).

En d'autres termes, la duplication de calcul n'est avantageuse que lorsque le système est globalement sous chargé, c'est-à-dire lorsqu'il existe des processeurs inoccupés : il vaut mieux dupliquer un calcul que ne rien faire. Cette méthode peut améliorer grandement la vitesse de diffusion du travail dans un réseau de processeurs mais elle nécessite un mécanisme de contrôle global complexe car l'apparition inopinée d'un travail plus utile (gestion de priorité) avant la fin d'un calcul dupliqué devrait permettre une réquisition du processeur qui effectue ce calcul redondant (voire l'abandon de ce calcul). Nous avons renoncé à utiliser cette possibilité dans notre première version de système car le processeur de la machine cible (Transputer) n'était pas très bien adapté au traitement des exceptions.

La méthode "par copie de données" est celle que nous avons choisie dans notre modèle car nous pensons que les machines MIMD à communication par messages ont un bel avenir.

3.2.5 Comparaison des méthodes de projection de l'arbre de recherche

Nous n'allons pas revenir sur les considérations architecturales qui différencient les machines à mémoire commune des machines à mémoire distribuée du point de vue de l'extensibilité (du nombre de processeurs). Il est encore trop tôt pour fournir des mesures précises et comparatives des deux types de machine. A ce stade de notre présentation, nous pouvons faire les remarques suivantes :

a) Du point de vue du temps d'exécution

L'adressage global permet le partage de mémoire au prix d'une exécution séquentielle ralentie par le coût de gestion de ce partage. Ce coût est du au problème des liaisons multiples (voir section 3.3) et à la surcharge du bus. Certains auteurs estiment que le surcoût total de l'exécution de leur système parallèle sur une seule unité de travail par rapport à l'exécution séquentielle normale (la WAM) varie entre 5% et 30%. Ce chiffre porte uniquement sur les coûts du modèle (les effets de caches sont ignorés). Par contre, la création d'une nouvelle tâche est très efficace car une partie du prix à payer pour fournir son contexte au nouveau processus d'évaluation est répartie sur toutes les phases de l'exécution séquentielle.

L'adressage local implique la copie de données ou le recalcul. L'exécution séquentielle est voisine des techniques classiques donc très efficace, par contre le coût de la création d'une nouvelle tâche d'évaluation est relativement élevé. Il est concentré dans le temps au moment de la copie des piles. Si le réseau de communication entre processeurs est sous dimensionné, si l'acheminement des messages utilise les mêmes ressources que le calcul (exemple : technique de "store and forward" des Transputers de la génération courante) alors la copie entre deux processeurs peut dégrader indirectement l'efficacité des autres proceseurs. L'avènement de circuits de routage autonome diminuera sans doute cet inconvénient.

En conclusion on peut considérer que, dans la mesure où les systèmes multiséquentiels ont été conçus pour favoriser l'exécution séquentielle, la technique de copie est l'approche la plus intéressante. Ces systèmes sont bien adaptés aux architectures actuelles qui comportent de quelques dizaines à quelques centaines de processeurs. En effet, le nombre d'occurences de compositions parallèles (création de tâche) est très inférieur au nombre de compositions séquentielles (poursuite en séquence). De ce point de vue, l'inconvénient de la technique de partage réside dans le fait que le surcoût de gestion du parallélisme potentiel est uniformément réparti sur la durée d'exécution du programme et que, par conséquent, on en paye le prix même s'il n'y a pas d'exécution parallèle (cas très fréquent: durant l'exécution d'un programme donné le nombre de créations de processus est très inférieur au nombre d'appels de procédure).

b) Du point de vue de la taille de mémoire requise

Si l'on considère uniquement la taille mémoire effectivement allouée par la machine virtuelle Prolog, c'est à dire le nombre maximal de cellules mémoire occupées par les piles d'évaluation à un instant donné, les modèles à partage de mémoire sont plus avantageux que les modèles qui comportent une duplication de données car le surcoût mémoire du partage est faible devant le taux de duplication lié à la copie.

Dans le cas où la branche la plus longue de l'arbre est entièrement allouée dans la même pile (pire des cas) la taille de pile est maximale pour un programme donné. Dans le cas des systèmes multi-séquentiels à partage de mémoire, l'environnement de machine virtuelle Prolog (le système d'exploitation) doit réserver le même espace (maximal) pour toutes les piles car il n'est pas possible de prévoir à la compilation quelle sera la pile qui de taille maximale, On peut remarquer que la taille la taille mémoire allouable (taille nécessaire dans le pire des cas) est la même pour les modèles à partage de mémoire et les modèles sans mémoire commune. Dans les deux cas la quantité de mémoire totale disponible est divisée en autant de parties (égales) qu'il y a d'unités de travail. La taille de ces parties borne la profondeur de l'arbre de recherche évaluable.

c) Du point de vue évolution des architectures

La facilité d'extension joue clairement en faveur des machines à mémoire distribuée bien que certaines machines à mémoire commune construites aujourd'hui ont des possibilités d'extension non négligables: c'est le cas des machines pour lesquelles le médium de communication entre la mémoire globale et les processeurs n'est pas un simple bus mais un réseau plus complexe limitant les effets de congestion (exemple: machine **BBN**, machine RP3 d'IBM).

Le coût de la copie, qui est le principal handicap des modèles conçus pour les architectures à mémoire distribuée (communication par message), est fortement lié à la technologie des médiums de communication qui en est à ses débuts; cependant son évolution est rapide.

3.3 Gestion des liaisons multiples

Dans la section précédente nous avons fait abstraction des problèmes liés au partage logique de portions de pile. Cette section est dédiée à la gestion des liaisons multiples. C'est une caractéristique majeure des systèmes multi-séquentiels exploitant le parallélisme OU. Après avoir rappelé les principales micro-opérations mises en œuvre dans le modèle séquentiel pour la gestion des liaisons et les différents types de liaisons, nous passerons en revue les principaux modèles de gestion des liaisons multiples qui ont été présentés dans la littérature (il s'agit essentiellement des modèles développés dans le cadre du projet Gigalips pour les machines à mémoire partagée). Seuls les mécanismes de base sont présentés; nous avons volontairement omis de discuter les nombreuses variantes obtenues par mixage de ces techniques. Nous détaillons plus particulièrement le modèle SRI [Warr87b]: bien que poursuivant une approche différente de la notre (le modèle SRI a été conçu en vue de l'utilisation de mémoire commune), on peut faire quelques analogies (les deux modèles ont en commun une même type de structure: le tableau de liaison, pile de vraiable). Cette section se termine par une comparaison des différents modèles de gestion des liaisons en les triant suivant deux grandes classes: les modèles basés sur la duplication des variables dans une zone d'adressage partagée (partage de l'adresse de la cellule allouée) et les modèles qui utilisent le rassemblement des liaisons dans la zone de création de la liaison. Les représentants de cette dernière classe sont principalement les modèles utilisant des techniques de recherche associative pour l'accès aux liaisons.

3.3.1 le modèle séquentiel

Nous rappelons ci-après les opérations élémentaires sur les variables dans le cas du modèle séquentiel classique (WAM) comme base de référence pour les surcoûts induits par le parallélisme. La réalisation de ces opérations intervient d'une manière très importante dans l'efficacité globale du système Prolog car leurs fréquences d'utilisation sont très élevées.

Les opérations élémentaires sur les variables sont au nombre de 4 (pour le modèle séquentiel):

a) L'allocation et l'initialisation d'une variable: Comme nous l'avons vu dans la section relative à la WAM, les variables peuvent être allouées soit dans la pile globale, soit dans la pile locale, selon leur durée de vie relativement à celle du contexte de la procédure qui les a créées.

Indépendamment de son allocation, il est nécessaire d'initialiser une variable ou plutôt de définir son état de liaison (Libre ou lié à une valeur unique).

Dans le cas d'une variable dans la pile globale, l'initialisation suit immédiatement l'allocation ; dans le cas d'une variable dans la pile locale, l'allocation est effectuée par l'instruction `allocate` et l'on initialise la variable lors de sa première occurence d'accès. Dans ce dernier cas, il est possible que l'initialisation fixe directement une liaison à un terme au lieu d'initialiser la variable à libre.

Dans le cas général, cette initialisation a pour but de mémoriser le fait qu'une variable est libre avant d'être liée (une seule fois par branche d'évaluation). Dans la WAM, on distingue une variable libre d'une variable liée à un terme par le fait que le champ valeur de la cellule variable est l'adresse de la variable elle même. D'autres variantes sont possibles : on peut par exemple, imaginer d'utiliser un tag spécifique des variables libres. L'opération d'initialisation d'une variable correspond à un accès en écriture qui doit précéder tout autre accès à cette variable (lecture ou écriture).

b) L'opération de liaison d'une variable : Elle représente une substitution et consiste à assigner une valeur (qui peut être elle même une variable) à une variable initialisée à libre. Cette opération doit être réversible lorsqu'elle est effectuée dans une alternative postérieure à sa création. Dans ce cas, on tabule la "fonction réciproque" c'est à dire qu'on empile l'adresse de la variable liée dans la pile trainée pour pouvoir la réinitialiser plus tard. L'opération de liaison d'une variable correspond à un accès en écriture. Dans la WAM, il est indispensable (à cause des durées de vie induites par le mécanisme d'allocation) de choisir comme représentante de cette classe d'équivalence la variable la plus ancienne. C'est ce que certains auteurs nomment la contrainte d'ancienneté : lorsque l'unificateur se trouve dans le cas de deux variables (libres), il doit lier la plus récente (considérée comme la variable) à la plus ancienne (considérée comme la valeur). On obtient ainsi une représentation arborescente des variables logiques d'une même classe d'équivalence, les pointeurs étant orientés du nœud fils vers le nœud père puisque la durée de vie de ce dernier recouvre celle de ses fils.

c) La déliaison d'une variable : C'est l'opération qui consiste à réinitialiser une variable (à l'état libre) lors du retour arrière. La pile trainée est une pile dans laquelle les adresses des variables liées sont classées par ordre chronologique de liaison, la liaison la plus récemment établie étant sur le sommet de pile. Le retour arrière de la stratégie séquentielle consiste principalement en l'application de l'opération de déliaison sur toutes les variables dont les adresses sont dans la tranche qui correspond à toutes les modifications postérieures au point de choix sur lequel on veut revenir.

d) L'opération de déréférencement : Le déréférencement est l'opération qui consiste à obtenir la valeur de liaison d'une variable logique donnée.

C'est sans doute la plus courante car elle correspond à l'accès en lecture. Cette opération est aussi la plus complexe dans le modèle séquentiel car le fait de représenter l'instance d'un terme par la factorisation de différentes instances de sous-termes crée des ensembles de variables liées. La composition des substitutions est assurée en choisissant une variable comme représentatrice de la classe d'équivalence des variables apparaissant dans la composition de substitution. Une substitution élémentaire étant représentée par une liaison, la composition de substitution se traduit par une chaîne de liaison. Le déréférencement correspond à l'exécution de l'opération composition de substitution.

Des quatre opérations élémentaires sur les variables, le déréférencement est la seule à ne pas pouvoir être exécutée toujours en temps constant même si dans la WAM l'accès

unitaire (sans déréférencer) est constant. En effet, lorsqu'une variable n'est pas représentante de sa classe d'équivalence (i.e. la plus ancienne) il est nécessaire de parcourir la chaîne de liaison des variables jusqu'à trouver la variable ultime, qui est soit libre soit liée à un terme non variable.

La longueur de ces chaînes de liaison est finie, et même s'il n'est pas possible de borner statiquement (à la compilation) cette longueur, le problème d'efficacité du déréférencement ne se pose pas aussi crucialement que l'on pourrait le croire de prime abord : en effet, des études [Chas89c] ont montré que 90% des chaînes de référence ont pour longueur 1. Ceci s'explique par le fait même du déréférencement qui réduit systématiquement les chaînes de liaisons à la longueur 1, leur allongement ne pouvant être provoquée que par une insertion d'un élément en fin de chaîne. Ce dernier cas n'est pas très fréquent car il correspond à l'utilisation d'une variable que l'on ne lie que postérieurement (dans un sous-but plus "à droite"). Or, dans la plupart des programmes, on cherche à instancier le plus rapidement possible les variables en ordonnant les sous-buts de manière à filtrer le plus tôt possible l'ensemble des solutions (on privilégie le filtrage sur la génération pour limiter la largeur de l'arbre de recherche).

Les algorithmes correspondants aux différentes opérations sont donnés ci-après. La notation utilisée s'inspire du langage C. Ainsi, comme dans le langage C, *Var dénote la cellule représentant la variable pointée par le pointeur Var. La notation Var->Tag est une abréviation pour (*Var).Tag. Notons que la cellule représentant la variable peut être soit dans la pile locale soit dans la pile globale. On peut remarquer que tous les pointeurs utilisés sont "taggés" (le tag représente le type de l'objet pointé). Dans la plupart des machines, cette représentation permet de stocker sur un seul mot de mémoire le tag et la valeur (l'adresse) ce qui permet un gain de vitesse important.

```
pilelocale : tableau de termes (tag,Val)
pileglobale: tableau de termes (tag,Val)
piletrainee: tableau d'adresse de terme

void initialiser( Var )
debut
  *Var := ( NonLiee , Var ) ;
fin

void lier( Var , (Tag,Val) )
debut
  *Var := (Tag,Val) ;
  si conditionnelle( Var ) alors
    debut
      Top_Trainee := Top_trainee + 1 ;
      trainee[ Top_trainee ] := Var ;
    fin
fin
```

```
void delier( trainee [ Top_trainee ] )
debut
  Var := trainee [ Top_trainee ] ;
  *Var := ( NonLiee , Var ) ;
  Top_trainee := Top_trainee - 1 ;
fin

(Tag,Val) dereference( Var )
debut
  si Var->Tag = variable alors
    debut
      si Var->Val = Var alors
        retourner ( ( variable , Var ) ) ;
      sinon
        retourner ( dereference( Var->Val ) ) ;
    fin
  sinon
    retourner ( ( Var->Tag , Var->Val ) ) ;
fin
```

3.3.2 Types de liaisons et mécanismes d'implantation

On peut classer les liaisons effectuées lors de l'évaluation d'un programme Prolog en plusieurs catégories. Nous avons rassemblé dans ce paragraphe les principales notions rencontrées dans la littérature afin de préciser leurs différences ou leurs ressemblances.

a) Liaisons héritées / Liaisons synthétisées

Une première façon de classer l'ensemble des liaisons en 2 catégories tient compte uniquement des dates respectives de création des éléments de la liaison :

a.1) liaison héritée : Une liaison héritée est une liaison d'une variable vers un objet (variable ou terme) qui a été créé avant la variable. Il s'agit des liaisons qui correspondent au passage par valeur des langages classique (paramètres d'entrée). Typiquement la liaison (X = a) provoquée par l'unification du but :- p(a) à la tête de clause p(X) permet à l'environnement de la clause appelée d'hériter de l'information produite par l'environnement appelant.

a.2) liaison synthétisée : Au contraire, une liaison synthétisée est une liaison d'une variable vers un objet (variable ou terme) qui a été crée après la variable. Ty-

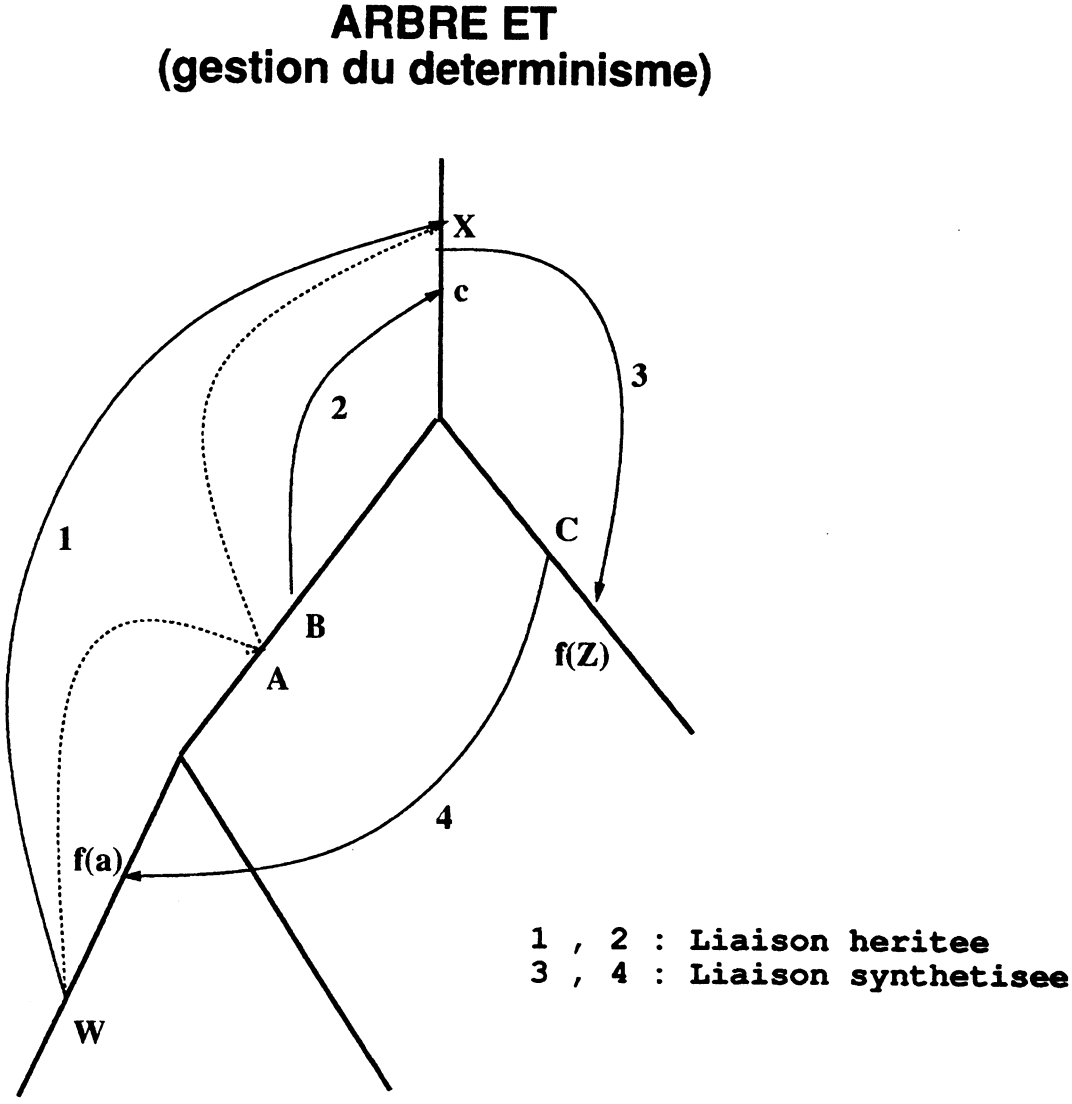


Figure 3.7 : Liaisons héritées - liaisons synthétisées

Chaque segment de l'arbre représente la portion de ressource allouée (en pile) par les différents appels (sous-buts) d'un calcul déterministe. Les arcs flèchés représentent les références: l'origine d'un arc correspond au contexte de la référence alors que son extrémité correspond au contexte référencé. L'arc en pointillé figure un déréférencement nécessaire à l'établissement de la liaison 1. Les liaisons 1 et 2 permettent l'héritage d'informations en provenance de contextes appelants. Les liaisons 3 et 4 sont deux exemples de synthèse de résultats et illustrent la nécessité de la persistance des objets référencés après la disparition des contextes qui les ont créés. Cette représentation permet de visualiser le problème de la compétition entre plusieurs liaisons synthétisées pour une même variable dans le cas d'une évaluation parallèle des branches de l'arbre ET. Dans le cadre d'utilisation conjointe du parallélisme ET et du parallélisme OU, l'arbitrage de cette compétition se ramène au problème de la jointure de flots de solutions

piquement la liaison X = a provoquée par l'unification du but :- q(X) à la tête de clause q(a) permet à l'environnement de la clause appelée de synthétiser de l'information consommée par l'environnement appelant. Rappelons que les liaisons synthétisées dont le champ valeur pointe vers des termes structurés impliquent l'utilisation de la pile globale puisqu'il est potentiellement nécessaire que la durée de vie des termes accessibles via la liaison synthétisée dépasse celle de l'environ de la procédure qui les a créés (données persistantes). Pour les objets qui peuvent être contenus dans les registres arguments, l'absence de problème est du à la "persistance" de ces registres.

Ces deux catégories sont disjointes et recouvrent l'ensemble des liaisons. La distinction entre liaison héritée et liaison synthétisée est surtout importante pour le parallélisme ET.

Un point fort de la programmation logique est d'avoir dans l'unification un mécanisme universel pour le passage de paramètres par valeur ou par référence ce qui se traduit par des procédures concises qui peuvent être utilisées avec plusieurs configurations de données d'entrée et sortie. Cependant cette généralité a sa contrepartie: un temps d'exécution élevé car il est nécessaire d'effectuer les tests à l'exécution pour déterminer dans quel cas on se trouve.

Une partie des recherches consacrées à la programmation logique a porté sur des techniques d'inférence des modes (entrée de terme clos, sortie, mixte, etc...) et/ou des types qui permettent de fournir (en partie) au compilateur des informations permettant de spécialiser l'unification (simple affectation plutôt qu'unification). Nous ne développerons pas cet aspect dans notre thèse car il ne concerne directement que la stratégie d'évaluation des nœuds ET, donc le parallélisme ET.

b) Liaisons conditionnelles / Liaisons inconditionnelles

La deuxième façon de classer les liaisons est de choisir uniquement le critère relatif au non-déterminisme. Cette classification est la plus utilisée dans les modèles multiséquentiels utilisant le parallélisme OU car elle souligne un des problèmes essentiels de ces systèmes qui est la gestion des liaisons multiples.

b.1) liaison inconditionnelle ou universelle: Une liaison inconditionnelle appelée aussi liaison universelle [Chas89c] est une liaison qui porte sur une variable créée postérieurement au point de choix le plus récent (relativement à la date de la liaison). Plus précisément il s'agit d'une variable qui est liée avant d'être potentiellement partagée par les branches d'évaluation qui peuvent apparaître ultérieurement. Les liaisons qui possèdent cette caractéristique n'ont pas besoin d'être référencées dans la pile trainée car elles restent valides durant toute la durée de vie de la variable.

b.2) liaison conditionnelle: Toute liaison qui n'est pas inconditionnelle est dite conditionnelle. Les variables affectées de manière conditionnelle sont celles qui peuvent être (à ce moment là) potentiellement partagées par au moins deux branches d'évaluation : les liaisons qui possèdent cette caractéristique doivent être enregistrées dans la pile trainée car leur validité est donnée par la durée de vie de la branche qui a affecté

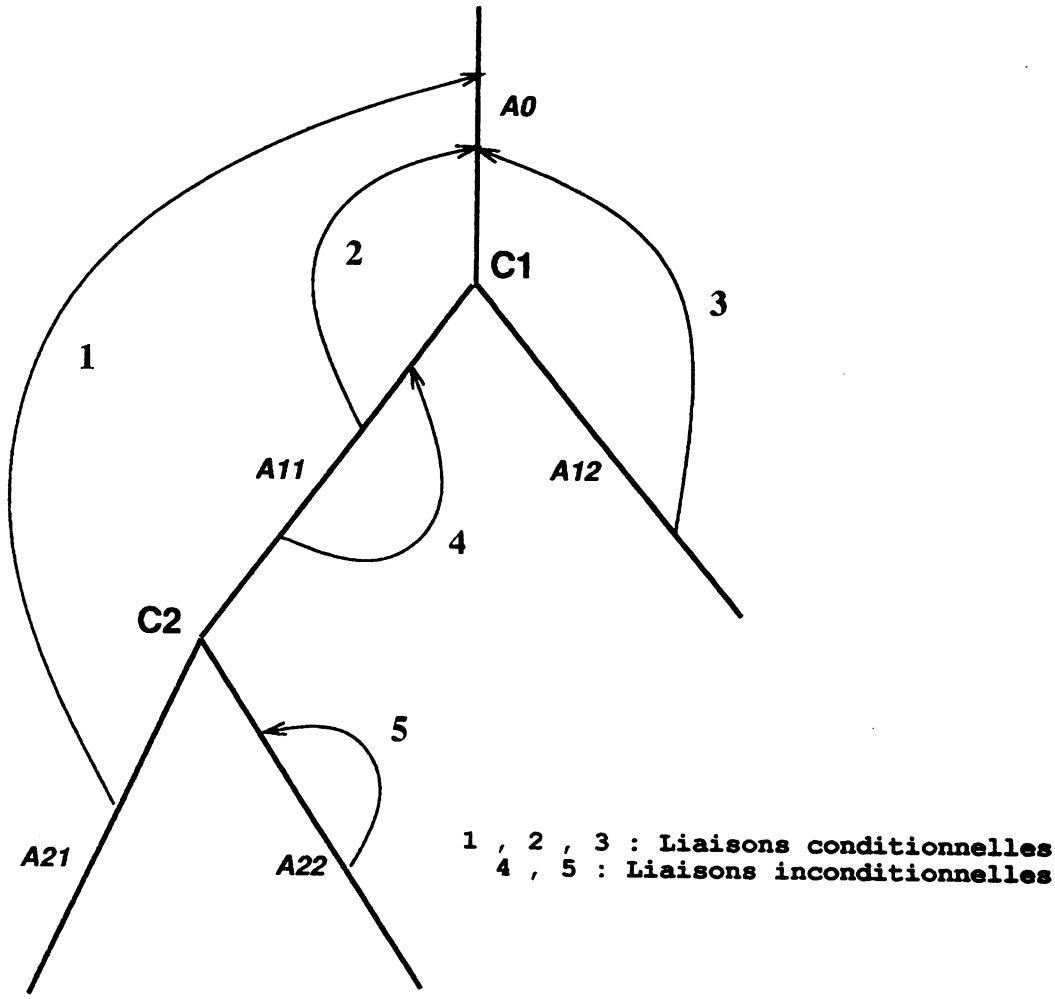


Figure 3.8 : Liaisons conditionnelles - liaisons inconditionnelles

Les segments de l'arbre représentent les portions de piles correspondant aux phases de calcul déterministe: à un segment donné peut correspondre toute une série d'appels de sous-buts. Les arcs représentent les liaisons: l'extrémité représente la définition (cellule de la variable dans le segment où elle a été créée) alors que l'origine représente une référence à la variable. L'origine est dans le segment correspondant au contexte de la liaison.

la variable. Ces liaisons sont donc celles qui portent sur les variables créées antérieurement à un point de choix existant : une variable est liée conditionnellement s'il existe au moins un point de choix entre le moment de sa création et celui de sa liaison. La profondeur de liaison est la différence entre l'adresse de la variable liée et celle du sommet de pile ; cette profondeur est représentative de l'écart temporel entre la création d'une variable et son unique affectation.

Ce critère distinctif existait déjà dans le modèle séquentiel car seules les liaisons conditionnelles sont enregistrées par la pile trainée. Pour le parallélisme OU, cette distinction est encore plus importante car les variables logiques étant à affectation unique, les variables liées inconditionnellement ne sont partagées qu'après leur affectation et ne nécessitent, par conséquent, aucune synchronisation lors d'éventuels accès ultérieurs puisqu'il s'agit uniquement d'accès en lecture.

Dans le cas général, ce critère de distinction ne peut être appliqué qu'à l'exécution car il n'est pas toujours possible de prévoir s'il va y avoir un point de choix créé entre l'allocation d'une variable et sa liaison. Dans le cas d'une stratégie d'évaluation parallèle, le problème est encore plus important car l'ensemble des points de choix n'est plus accédé comme une pile (au sens strict). On peut donc voir disparaître progressivement des points de choix qui ne sont pas les plus récents et, au gré de l'ordonnancement des branches d'évaluation, une liaison conditionnelle peut donc finir par devenir inconditionnelle. Cette distinction est importante car les coûts des mécanismes d'implémentation des deux classes de liaisons sont très différents.

c) Mécanismes de liaisons profondes / liaisons superficielles

Ayant esquissé un critère de classification important pour le parallélisme OU, nous allons examiner différentes implémentations proposées dans la littérature. Pour les liaisons inconditionnelles, nous avons vu que le mécanisme de la machine séquentielle classique (liaison profonde) reste valable. Pour les liaisons conditionnelles, le parallélisme implique une gestion beaucoup plus lourde. Cette gestion repose soit sur le mécanisme de liaison profonde, soit sur le mécanisme de "liaison superficielle". Le mécanisme de "gestion de liaisons multiples" est au cœur des systèmes multi-séquentiels utilisant le parallélisme OU.

c.1) liaison profonde ou globale : Dans la WAM, le mécanisme de gestion des liaisons conditionnelles implique une modification des cellules allouées par l'environnement appelant : lors d'une liaison conditionnelle, on modifie une variable "enfouie profondément" dans la pile. Ceci peut provoquer des conflits de liaison dans le cas du parallélisme OU si l'on conserve exactement la même fonction d'adressage (adressage indirect) d'une variable pour tous les processeurs.

Pour résoudre ce problème une première méthode consiste à choisir une représentation des variables qui augmente la capacité des cellules mémorisant l'état de liaison des variables tout en conservant l'adressage indirect (car celui-ci correspond directement à un mécanisme d'adressage du matériel et reste donc le plus efficace). Cet "élargissement" des cellules variables peut être fait a priori (dès la création) pour toutes les

variables (modèle SRI [Warr87b], modèle OPERA) ou a posteriori pour les variables pour lesquelles la coexistence de multiples valeurs de liaison est détectée à l'exécution (modèle à vecteur de version [Haus87] et d'une certaine manière modèle d'Argonne [Butl86] pour ces liaisons favorisées). Cette dernière solution permet de diminuer le nombre de variables à traiter au prix d'un traitement dynamique.

L'inconvénient principal du partage d'une cellule élargie entre les différents processeurs est l'accès en lecture et écriture qui implique des mécanismes coûteux de synchronisation. Nous verrons plus loin une solution qui ne nécessite pas de synchronisation : le modèle SRI.

c.2) liaison superficielle ou locale : Une deuxième méthode consiste à éviter de modifier l'environnement appelant de manière à faciliter son partage (en lecture seule). Pour cela, on représente les liaisons conditionnelles localement (de manière superficielle) en allouant les ressources nécessaires dans le contexte (superficiel relativement aux piles) qui provoque la liaison.

On garantit ainsi un partage complet des ressources héritées et on facilite l'installation d'une nouvelle tâche. Le prix à payer est relativement élevé : la représentation locale des liaisons superficielles ne permet plus d'utiliser directement la projection de l'espace des noms de variables sur l'espace d'adressage physique. En effet, il est nécessaire de mémoriser localement les liaisons inconditionnelles sous forme d'une liste de doublets (nom de variable, valeur de liaison) à laquelle on accède par une recherche associative.

Bien que cette recherche puisse être accélérée par des techniques de hachage en prenant le champ nom de variable comme clef, cette méthode implique une durée d'accès unitaire à une variable qui n'est plus bornée alors que, dans les mécanismes de liaison profonde, elle équivalait à la durée d'un nombre constant et faible d'accès mémoire. Les auteurs prônant ce type de mécanisme arguent que les liaisons conditionnelles représentent une faible proportion des liaisons réalisées et que, par conséquent, le surcoût élevé du déréférencement est masqué par la facilité d'installation d'une nouvelle tâche.

Si des mesures expérimentales confirment que la majorité des liaisons sont inconditionnelles donc réalisées de manière traditionnelle (mécanisme de liaison profonde), il est beaucoup moins évident que la grande complexité des mécanismes de gestion associative des liaisons ne ralentisse pas globalement le système séquentiel plus qu'il ne permette d'améliorer l'aspect parallèle. Les principaux représentants de cette classe de mécanismes sont le modèle d'Argonne [Butl88] et celui de PEPSys de l'ECRC [Baro88a]

La gestion des liaisons multiples étant le principal trait caractéristique des systèmes multi-séquentiels utilisant le parallélisme OU, nous allons préciser les notions vues précédemment en examinant plus en détail les solutions les plus représentatives.

3.3.3 le modèle SRI

Ce modèle a été proposé (mais non publié) en 1983 par D.H.D. Warren lorsqu'il était au SRI puis redécouvert pour une grande partie par D.S. Warren [Dswa84]. Il est décrit succintement dans [Warr87b]. L'article de D.S. Warren nous a largement inspiré lors

de la conception du modèle OPERA ce qui explique les similitudes entre notre modèle et le modèle SRI.

Dans ce modèle, chaque processeur a son propre tableau de liaisons BA (pour "binding array") qui sert à mémoriser les valeurs des liaisons conditionnelles car elles lui sont propres. Ce tableau est en fait géré comme une pile de vecteurs dont le sommet est Top_BA[Pid].

Les liaisons inconditionnelles sont mémorisées directement dans les piles locales ou globales selon le type de la variable à lier puisque, pour ce type de liaison, il n'y a pas de problème de coexistence de valeurs multiples. La technique utilisée par la WAM reste valide et évite l'indirection supplémentaire par le BA dans le cas d'un accès aux variables liées inconditionnellement ou aux variables libres.

La propriété la plus importante du modèle SRI du point de vue architectural est la possibilité de partager en lecture les piles locales et globales sans synchronisation : les parties accédées en écriture par un processeur donné (le tableau de liaison et son miroir, la pile trainée) étant bien regroupées dans une zone privée du processeur. Cette propriété fondamentale est (à l'origine) intimement liée à l'idée d'utiliser des architectures à mémoire partagée pour implémenter les systèmes Prolog parallèles (projet Gigalips).

Voici les algorithmes correspondant aux opérations de base sur les variables dans le modèle SRI. Le lecteur pourra noter la grande similitude avec le modèle séquentiel classique. Cette similitude (recherchée) a pour but essentiel de limiter la dégradation de performance due au parallélisme en restant compatible avec le modèle séquentiel.

```
pilelocale: tableau de (Tag,Val) ou (NonLiee,IndexBA)
pileglobale:tableau de (Tag,Val) ou (NonLiee,IndexBA)
BA        : tableau de liaison (Tag,Val)
trainee   : tableau de couple (IndexBA,(Tag,Val))

void initialiser( Pid , Var )
debut
   *Var := ( NonLiee , Top_BA[ Pid ] ) ;
   BA[ Pid , *Var ].Tag := NonLiee ;
   /* BA[ Pid , *Var ].Val est non defini */
   Top_BA[ Pid ] := Top_BA[ Pid ] + 1 ;
fin
```

Notons que la gestion des variables se fait en pile : le tableau de liaisons local à un processeur. Les variables sont donc classées par ordre de création et le registre utilisé pour pointer sur le sommet du tableau est sauvegardé dans les points de choix au même titre que les registres d'états habituels de la WAM. L'inconvénient du mélange des variables globales et locales dans ce même tableau de variables provient de l'impossibilité de récupérer l'espace alloué dans le cas de l'optimisation de la récursion terminale alors

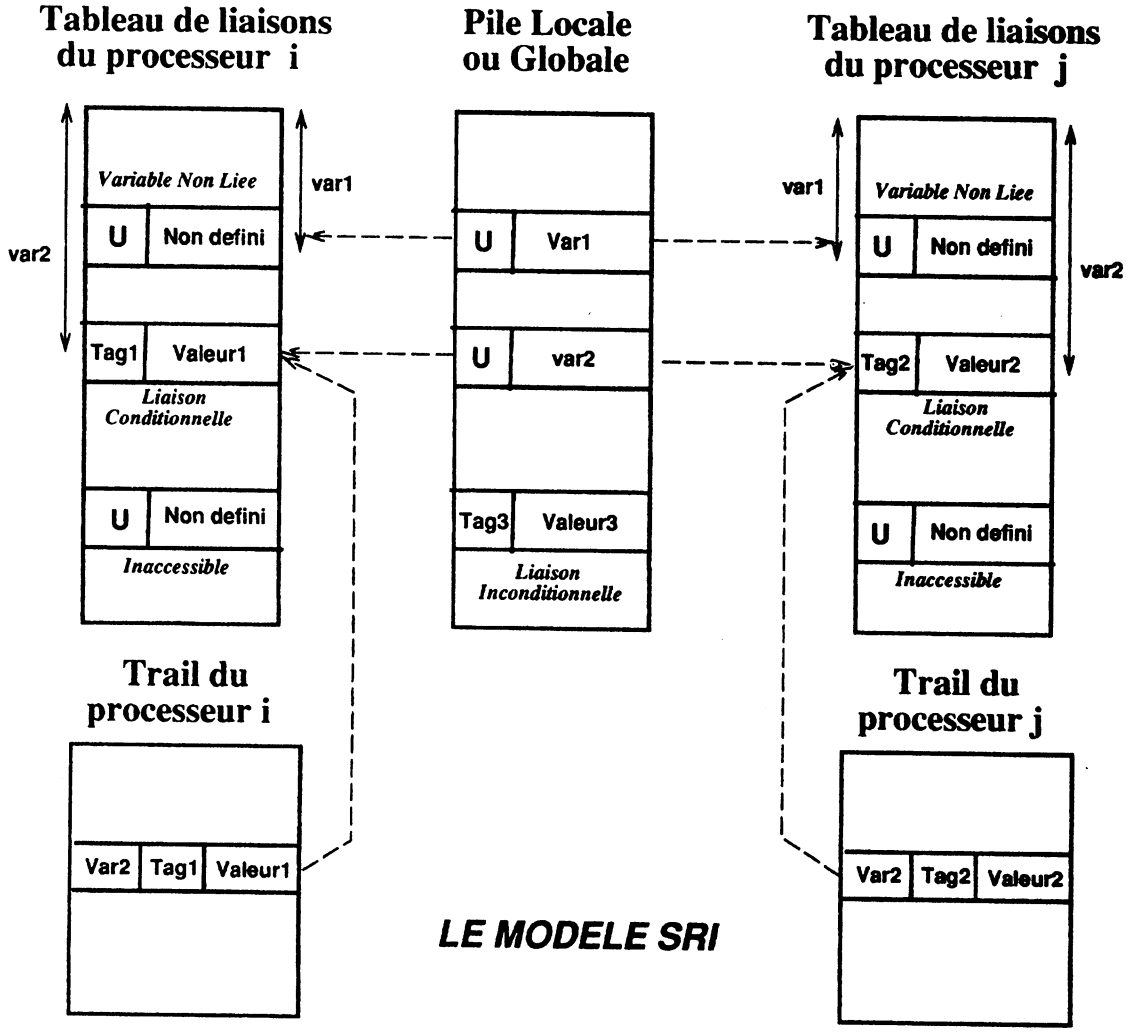


Figure 3.9 : Modèle SRI

que la gestion des environnements de la WAM avait été particulièrement bien étudiée pour récupérer progressivement des parties d'environnement tout au long des phases de calcul déterministe.

Cette optimisation de l'espace mémoire est essentielle pour les gros programmes de type itératif. Sans elle, un programme ayant un calcul déterministe important risque de saturer l'espace mémoire rapidement car il n'y a quasiment aucune récupération de mémoire sans retour arrière. En effet, seul l'environ de la pile locale peut être récupéré (comme dans la WAM) mais pas la cellule du tableau de liaison correspondante car elle n'est pas forcément sur le sommet de pile (plus exactement il peut y avoir une liaison de variable globale à conserver entre la cellule qui n'est plus accessible et le sommet de pile). Pour éliminer cette contrainte, D.H.D. Warren propose d'éclater le tableau de liaisons en deux à l'image des piles locale et globale. Ceci implique l'utilisation de deux registres au lieu d'un seul et donc un accroissement supplémentaire de la taille de l'état à sauvegarder dans les points de choix.

On notera également l'utilisation d'un tag spécial "NonLiée" différent du tag "variable" qui devient spécifique des variables liées (à une autre variable ou à un un terme non variable).

```
void lier( Pid , Var , (Tag,Val) )
debut
si inconditionnelle( Var ) alors
  *Var := (Tag,Val) ;
sinon
  debut
    IndexBA := Var->Val ;
    Top_trainee [ Pid ] := Top_trainee [ Pid ] + 1 ;
    trainee[Pid,Top_trainee[Pid]]:=(IndexBA,(Tag,Val));
    BA[ Pid , IndexBA ] := (Tag,Val) ;
  fin
fin

void delier( Pid, trainee[Pid,Top_trainee[Pid]].IndexBA )
debut
  IndexBA := trainee[ Pid , Top_trainee[Pid] ].IndexBA ;
  Top_trainee[ Pid ] := Top_trainee[ Pid ] - 1 ;
  BA[ Pid , IndexBA ].Tag := NonLiee ;
fin
```

L'opération d'accès en lecture est similaire à celle de la WAM séquentielle dans le cas où il s'agit de liaisons inconditionnelles. Elle ne coûte qu'une indirection et une indexation de plus (par variable) dans les autres cas.

```
(Tag,Val) dereference( Pid , Var )
debut
  si Var->Tag = NonLiee alors
    /* il s'agit d'une variable libre  */
    /* ou bien liee conditionnellement */
    debut
      IndexBA := Var->Val ;
      choix BA[ Pid , IndexBA ].Tag
        cas NonLiee :
          retourner( ( NonLiee , _ ) ) ;
        cas Var :
          retourner(dereference(BA[Pid,IndexBA].Val);
        autres_cas :
          retourner ( BA[ Pid , IndexBA ] ) ;
        retourner ( ( variable , Var ) ) ;
      finchoix
    fin
  sinon
    /* il s'agit d'une liaison inconditionnelle */
    debut
      si Var->Tag = variable alors
        retourner ( dereference( Var->Val ) ) ;
      sinon
        retourner ( ( Var->Tag , Var->Val ) ) ;
    fin
fin
```

D'après D.H.D. Warren [Warr87b] le surcoût global du modèle SRI sur le modèle séquentiel de la WAM serait approximativement compris entre 40% et 60%. On peut remarquer la simplicité du modèle SRI ; c'est un avantage pour une intégration dans le matériel. Par contre, la plus grande quantité de mémoire nécessaire est un inconvénient de cette méthode : dans le cas d'un tableau de liaisons unique par processus, la récupération de place ne peut se faire que lors du retour arrière ce qui peut devenir pénalisant pour des programmes dont la partie déterministe est importante.

3.3.4 modèle à Vecteur de Versions

Ce modèle a été proposé par l'équipe de SICS Haussman, Ciepielski et Haridi [Haus87]. Il est relativement proche du modèle SRI. Le principe en est le suivant :

Chaque variable libre est représentée par une cellule dont l'adresse peut être connue de tous les processeurs. Lorsqu'une variable est liée pour la première fois, on affecte

à cette cellule l'adresse d'un vecteur de versions que l'on alloue dynamiquement. Ce vecteur contient autant de cellules qu'il y a de valeurs de liaison potentielles : Chaque processeur dispose via l'adresse de la variable de l'adresse de base du vecteur et il peut accéder à la cellule qui correspond à la branche qu'il évalue en utilisant un numéro l'identifiant comme index dans le vecteur. Les vecteurs de versions sont désalloués lors du retour arrière effectué par le dernier processeur à les référencer.

L'avantage de cette méthode est qu'elle est très simple et qu'elle peut donc faciliter l'intégration éventuelle de ces mécanismes dans le matériel. Comme le modèle SRI, elle permet de conserver constant le coût des opérations élémentaires sur les variables.

L'initialisation de la cellule variable par l'adresse d'un vecteur de versions s'effectue lors de la première liaison donc il n'est pas possible de prévoir quel processeur va effectuer l'allocation du vecteur de version. En conséquence, il est nécessaire de synchroniser l'accès à ces variables. La propriété de partage des piles locale et globale en lecture seule qui est le point fort du modèle SRI n'est pas vérifiée dans le modèle à vecteur de versions. Cette synchronisation de tous les processeurs est très fréquente puisqu'elle intervient à chaque première liaison de variable. C'est un inconvénient majeur de la méthode qui la rend inutilisable dans le cas de machines comportant un grand nombre de processeurs.

Un autre inconvénient est que la taille des vecteurs est toujours de l'ordre du nombre de processeurs pour toutes les variables même si une faible proportion des processeurs lie une variable donnée. La taille mémoire requise peut donc devenir très importante alors que la taille mémoire réellement utilisée (cellules référencées) est faible. Du point de vue du temps d'exécution, on peut remarquer que l'initialisation de l'ensemble du vecteur à "libre" lors de la première liaison est une opération coûteuse (et très fréquente).

Enfin remarquons que ce modèle ne peut pas fonctionner sans mémoire commune à cause du mécanisme d'accès aux vecteurs.

3.3.5 modèle d'Argonne

Disz, Lusk et Overbeek du laboratoire d'Argonne, ont développé un Prolog parallèle basé sur l' ANL-WAM [Butl86] pour les machines à mémoire partagée. L'idée sous-jacente au modèle est la suivante.

Pour un processeur, la représentation des variables de la WAM rend les opérations d'accès aux variables très efficaces. Dans le cas de plusieurs processeurs, on conserve cette propriété pour un seul processeur. On privilégie alors le processeur qui a alloué la variable : pour lui la représentation de la variable est quasiment la même qu'en séquentiel et les opérations d'accès sont identiques du point de vue coût d'exécution (il y a simplement un bit supplémentaire qui indique que la liaison présente n'est valide que pour le processeur favorisé) ; par contre les autres processeurs qui accèdent à cette variable partagée testent le bit qui signale que la valeur n'est pas pour eux et effectuent une recherche de la valeur correspondant à leur branche en utilisant une table de hachage. Il y a une table de hachage par segment de l'arbre OU. Elle mémorise les liaisons de variables partagées (effectuées sur ce segment).

Le principal inconvénient du modèle d'Argonne est le coût très élevé de l'accès

aux liaisons non favorisées. De plus ce coût n'est pas borné (parcours de chaîne de tables). Pour un processeur favorisé, il en existe beaucoup qui sont défavorisés. Selon les auteurs, les liaisons défavorisées représentent une petite partie de l'ensemble des liaisons réalisées durant l'exécution de tout le programme. Cependant, une simulation du modèle d'Argonne, effectuée par Kish Shen [Shen87], a montré que les accès aux liaisons défavorisées étaient beaucoup plus fréquents qu'on pouvait le supposer a priori (jusqu'à 50% de l'ensemble des références). Nous pensons donc que le surcoût de la gestion des tables de hashcode grève sérieusement les performances globales du système et diminue l'avantage que celles-ci procurent lors de la création d'une nouvelle tâche.

Des variantes plus ou moins complexes de ce modèle ont été étudiées par les chercheurs du projet Gigalips (elles sont obtenues par fusion de plusieurs modèles). Elles sont connues sous le nom de modèle d'Argonne-SRI et modèle Manchester-Argonne en référence aux diverses origines de leurs auteurs. Nous ne les détaillerons pas ici ; les points originaux sont abordés dans les modèles dont ils sont issus. Le lecteur trouvera une description des nombreuses variantes de ces modèles dans [Warr87a].

3.3.6 modèle de PEPSys

Le modèle PEPSys a été développé a l'E.C.R.C. [Baro88a]. Il est basé sur la technique des table de hachage. Il y a une table de hachage par processus qui sert à mémoriser les liaisons profondes de manière à éviter de modifier l'environment global qui reste partageable en lecture seule entre les processus issus d'un même nœud. Cette table est créée à l'installation de la tâche.

3.3.7 Synthèse des méthodes de gestion des liaisons multiples

On peut distinguer deux grandes classes de méthode de gestion des liaisons multiples car elles ont pour origine deux principes de base différents :

- Dans la première classe, les processus partagent l'adresse d'une variable donc l'adresse de la représentation des différentes liaisons de cette variable. Cette adresse unique identifie l'ensemble des états de liaisons d'une variable correspondant aux processus OU. La gestion de la multiplicité de ces liaison est effectuée en aval de l'adressage de la variable ce qui permet de conserver les performances de la technique séquentielle. L'ensemble des liaisons d'une même variable est de cardinalité bornée par le nombre de processus issus du processus créateur de la variable (dans la majorité des cas la borne utilisée ne dépasse pas le nombre de processeurs si il n'y a pas de "multi-programmation"). Cette propriété garantit une représentation mémoire de cet ensemble suffisamment simple pour permettre un accès très efficace (simple indexation dans le cas d'une mémoire commune ou adresse identique en cas de mémoires différentes). L'inconvénient de ce modèle réside dans la composition parallèle (installation d'une tâche) dont le coût est

au moins de l'ordre du nombre de variables communes aux deux branches d'évaluation à dupliquer (duplication de la partie modifiable du contexte commun). L'impact de ce problème peut être réduit par des mécanismes de copie partielle. Les propriétés de cette classe de modèles sont telles qu'il est envisageable d'intégrer une bonne partie de ces mécanismes dans le matériel.

- La deuxième classe de méthodes de gestion des liaisons multiples utilise le principe de la représentation locale des modifications : une modification (liaison de variable) est représentée dans le contexte qui la provoque et non dans le contexte sur lequel porte la modification. Au contraire de la première classe de méthodes qui utilise une adresse commune unique (accès direct indexé) pour repérer la représentation d'une liaison, cette classe de méthode repose sur des mécanismes d'accès associatif à l'ensemble des liaisons de variables effectuées par le contexte courant mais ne portant pas sur ce contexte. Contrairement à la première classe de modèle de mémoire, ce principe de représentation permet une composition parallèle indépendante de l'historique d'évaluation : en d'autres termes l'installation d'une tâche est à coût constant. En contrepartie, les opérations élémentaires d'accès aux variables ne se font plus en temps constant et la longueur des chaînes de tables de hachage que l'on peut être amené à parcourir lors de l'accès à une variable n'est pas bornée. Ce point est un handicap sérieux pour l'intégration dans le matériel car l'accès à une variable qui est une des opérations les plus courantes dans Prolog n'est plus à coût constant (ni même borné!). La gestion des tables de hachage est une opération beaucoup plus coûteuse qu'une simple indexation. De plus l'initialisation systématique des tables de hashcode est un surcoût inévitable qui peut grever les performances pour les programmes ayant un nombre de branches mortes (menant à une échec) importantes.

Le modèle SRI et le modèle à vecteur de version font partie de la première classe alors que le modèle PEPSys fait partie de la seconde. Le modèle d'Argonne serait à classer dans la catégorie des modèles à recherche associative de par le mécanisme de gestion des liaisons non favorisées bien que le mécanisme des liaisons favorisées entre dans la première catégorie.

DEUX CLASSES DE MECANISMES DE GESTION DES LIAISONS CONDITIONNELLES MULTIPLES

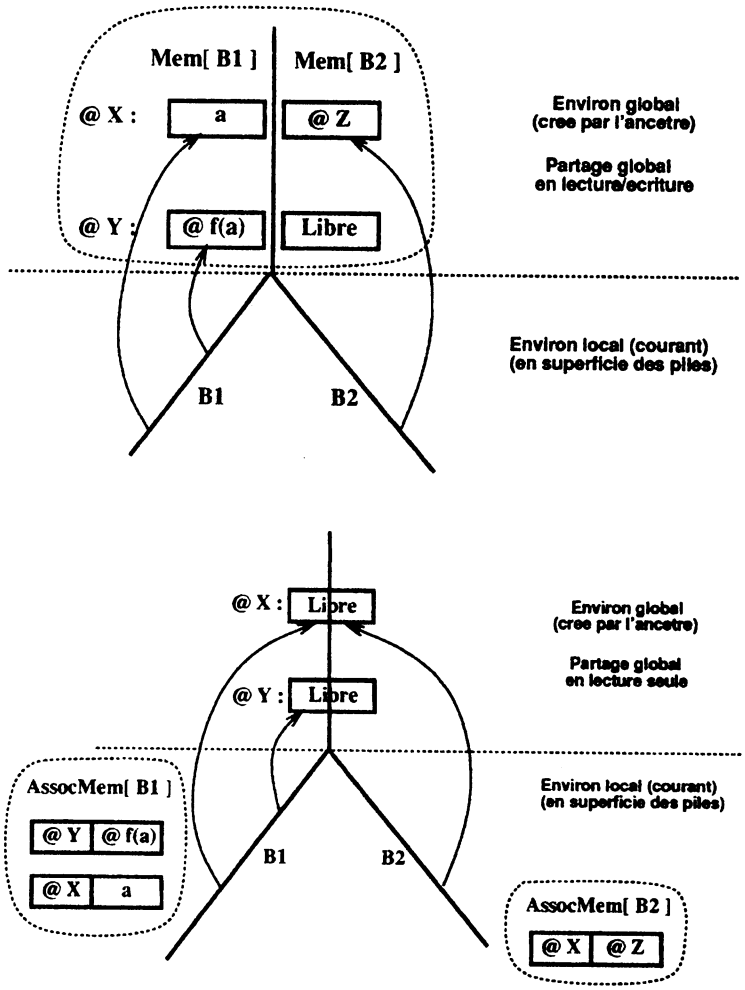


Figure 3.10 : Classes de mécanismes de gestion des liaison multiples

La première classe de mécanismes est telle que l'on partage les différentes valeurs de liaison d'une même variable sont référencées par une même adresse (ex: a et @Z sont deux valeurs de liaison possibles pour un même variable référencable par @X).

La seconde classe de mécanismes est fondée sur l'accès associatif aux valeurs de liaison, l'adresse de la variable jouant le rôle de clef d'accès. Chaque branche dispose de sa propre mémoire associative dans laquelle sont stockées toutes les modifications de l'environnement.

Chapitre 4

OPERA - PRINCIPES GENERAUX

4.1 Préliminaire

L'objectif d'OPERA est la conception d'un système multi-séquentiel OU parallèle sur une machine multi-processeurs sans mémoire commune (Supernode) comportant un grand nombre de processeurs. La réalisation d'un prototype permettra de vérifier cette conception. Notre approche est voisine de celle des projets sur machine à mémoire commune : il s'agit de définir une machine abstraite de type WAM ainsi que les mécanismes et règles permettant la coopération parallèle de telles machines à l'exécution d'un même programme.

Ce chapitre décrit les principes du modèle d'exécution d'OPERA sous les contraintes architecturales détaillées dans la section 4.2. Ces principes généraux ne sont pas liés à l'architecture Supernode et peuvent être appliqués à bien d'autres cas de machines MIMD. La section 4.3 précise les modifications apportées à l'opérateur de calcul séquentiel (la WAM) pour faciliter le parallélisme. Le contrôle du parallélisme est une tâche essentielle à effectuer et c'est l'une des originalités du modèle OPERA, aussi nous lui avons entièrement consacré le chapitre 5. La section 4.4 termine ce chapitre en présentant quelques aspects important du parallélisme que nous n'avons pas eu le temps de traiter complètement.

La projection de l'architecture logique du système OPERA sur l'architecture physique des machines de type Supernode, ainsi que les différents points techniques afférents, font l'objet du chapitre 6.

4.2 Contraintes matérielles d'OPERA

Le modèle multi-séquentiel d'OPERA correspond à un schéma type d'algorithme parallèle dit de la ferme de processeurs ("process farm"). Le principe de ce schéma est d'avoir plusieurs unités de travail (processus) partageant le même code et coopérant entre elles à

exécuter l'algorithme. Le principe de la coopération est le transfert de tâches d'une unité surchargée à une unité sous chargée. Un processeur physique peut supporter plusieurs unités de travail (multi-programmation) cependant, pour des raisons techniques que nous évoquerons plus tard, nous avons choisi d'assimiler la notion d'unité de travail (logique) à la notion de processeur (physique). Dans la suite de notre exposé, nous utiliserons indifféremment l'une ou l'autre (sauf mention explicite).

Dans OPERA, les unités de travail sont des machines abstraites Prolog séquentielles (WAM). Les tâches créées et échangées entre ces unités sont les points de choix (nœud OU) de l'arbre correspondant au programme exécuté. Les transferts de tâches nécessitent la gestion des ressources physiques des processeurs et des moyens de communications.

4.2.1 Les deux types de problèmes à résoudre.

Le parallélisme dans OPERA a pour principal objectif l'obtention d'un gain d'efficacité (en temps de calcul) par rapport aux machines séquentielles. Le modèle proposé doit être suffisamment indépendant de la machine cible pour permettre le portage du système sur des architectures voisines.

Pour ce faire, nous avons choisi de bien distinguer deux types de travaux nécessaires à la résolution d'un programme OU parallèle : l'évaluation des branches de l'arbre OU et le contrôle du parallélisme.

a) La partie opérative (évaluation des branches de l'arbre OU)

Ce calcul séquentiel classique est assuré par chaque unité de travail qui exécute une TWAM, machine abstraite de type WAM légèrement modifiée pour prendre en compte le parallélisme.

Dans le modèle OPERA nous avons cherché à rester le plus près possible du modèle séquentiel. Ainsi, cette partie opérative est totalement indépendante de son environnement (nombre de processeurs, médium de communication interprocesseurs donc, en particulier, indépendance vis à vis de la topologie du réseau d'interconnexion des processeurs et/ou de la technique d'acheminement des messages). Les interactions entre une unité de travail et son environnement sont peu nombreuses et limitées à des protocoles de communication très stricts. Comme nous le verrons par la suite, ces interactions ne sont jamais traitées directement par l'unité de calcul Prolog mais par des processus d'interface qui gèrent complètement le dialogue entre une unité de calcul et son environnement.

Cette indépendance de l'unité de travail par rapport à l'architecture qui la supporte doit permettre de profiter de l'évolution des techniques de compilation de Prolog séquentiel sans remettre en cause les principes d'exécution parallèle retenus. Elle doit également faciliter l'implantation de ce modèle sur différentes plateformes ou dans un environnement hétérogène. Dans ce chapitre, nous ne traiterons que du contrôle local à une unité de travail en ne faisant aucune hypothèse sur la manière dont est structuré son environnement.

b) Le contrôle des ressources

La gestion du parallélisme est un problème complexe qui dépend à la fois du programme à exécuter mais aussi et surtout de l'architecture cible. Le contrôle du parallélisme consiste principalement à gérer l'allocation des tâches aux processeurs. C'est un problème critique car le coût d'installation d'une tâche sur un processeur peut être tel qu'il annule le gain obtenu par parallélisation. Il est donc nécessaire de limiter ou d'éliminer (quand c'est possible) ce risque par une stratégie de contrôle appropriée.

Nous avons choisi de "sortir" complètement le contrôle du parallélisme de la machine Prolog pour pouvoir expérimenter différentes stratégies du parallélisme sans remettre en cause la TWAM.

Plusieurs organisations de ce contrôle sont possibles : centralisé, hiérarchisé ou complètement réparti sur les unités de travail. Nous examinerons ces solutions dans le chapitre sur l'ordonnancement. A ce niveau de notre étude, nous pouvons simplement constater que le contrôle de l'allocation des processeurs est une fonction qui met en jeu tous les processeurs, donc qui sous entend le maintien d'un état global. Le maintien d'un état global exact nécessite de synchroniser tous les processeurs. L'allocation des médiums de communication est dépendante de l'allocation des processeurs : après avoir effectué un choix de deux processeurs, il s'agit d'allouer une liaison bipoint pour transférer un travail de l'un à l'autre.

c) Les critères de choix

Les propriétés des machines à mémoire distribuée comme la machine cible de l'expérience (le Supernode) ont guidé nos choix. Nous avions le souci d'obtenir un prototype "réaliste" c'est à dire d'obtenir des performances significatives des caractéristiques architecturales propres à ces machines.

Avant d'étudier des solutions spécifiques, nous avons déterminé l'importance de certains traits matériels et leur influence possible sur la conception ou le comportement du système Prolog parallèle.

Nous n'avons pas hésité à exploiter à fond certains traits matériels dès qu'ils nous paraissaient significatifs de ce type de machine.

4.2.2 Absence de mémoire commune

Le premier principe (ou contrainte) d'OPERA est le choix du modèle "de copie" plutôt que du modèle à partage de mémoire. Ce choix provient des propriété de la cible privilégiée d'OPERA : les machines à mémoire distribuée telle que le Supernode. Une solution aurait été naturellement d'émuler une mémoire commune distribuée, et par là, de se ramener à un système multi-séquentiel classique en partageant logiquement la mémoire. Cette solution a été rejetée pour des raisons liées aux propriétés des architectures à mémoire distribuée :

1. un temps de communication excessif pour des accès d'un grain correspondant aux objets élémentaires de la WAM. Les machines à mémoire distribuée ont un temps

d'amorçage de communication qui privilégie les transfert de données importantes.

2. l'absence de dispositif matériel de type translation d'adresse virtuelle vers adresse réelle (pas de MMU disponible sur les Transputers),

3. la simulation par logiciel d'un MMU aurait ralenti considérablement le calcul et biaisé le rapport entre temps de calcul et temps d'entrées/sorties,

4. La gestion d'une mémoire virtuelle distribuée est un problème complexe encore ouvert.

5. le partage de mémoire est difficilement réalisable dans le cas d'un architecture hétérogène (ex : réseau local interconnectant des machines différentes)..

Cette solution restrictive permet une plus grande simplicité de la portabilité du système: il beaucoup plus facile de ne pas utiliser d'échange par mémoire sur une machine à mémoire commune que de simuler une mémoire commune sur une machine n'en ayant pas.

Le coût de transfert d'une tâche dépend du volume des données à copier. Une stratégie de contrôle du parallélisme doit en tenir compte. Les performances globales du système sont très dépendantes des performances des techniques et médium de communication d'une part et d'autre part de la répartition des données et de la fréquence des échanges entre unités de travail.

4.2.3 Moyen d'interconnexion des unités de travail

La seconde contrainte est la nécessité d'exploiter au mieux les propriétés spécifiques du réseau de communication de telles machines pour obtenir des performances absolues significatives de manière à faire ressortir les points réellement bloquants. Les architectures visées permettent généralement

1. un débit optimal pour des tailles de données spécifiques (nombreux petits messages de contrôle) ou croissant avec la taille des données (petit nombre de gros messages de données),

2. la possibilité de transfert en parallèle au calcul (DMA) avec une dégradation limitée de l'efficacité du calcul (vol de cycles mémoire),

3. la possibilité de communication directe entre des processeurs quelconques. Ce dernier point est une nécessité et il est essentiel pour l'efficacité globale du système car il correspond bien au modèle d'une ferme de processeurs où n'importe qui peut échanger du travail avec n'importe qui à un instant donné. Cette liaison bi-point dynamique peut correspondre à une liaison directe physique statique (maillage complet), dynamique (réseau reconfigurable) ou simulée par une technique d'acheminement de messages (routage).

Deux solutions sont envisageables selon que l'architecture cible permet la reconfiguration dynamique ou se compose d'un réseau incomplètement maillé de processeurs :

a) Réseau logique complet obtenu par connexion dynamique

Il n'est pas envisageable d'avoir un réseau d'interconnexion complet (chaque processeur relié à chaque autre) pour un grand nombre de processeurs. De même il ne peut être réalisé par un bus commun dont l'inconvénient est de conserver une bande passante d'E/S constante alors que l'on augmente le nombre de processeurs. Les processeurs ne peuvent pas avoir un grand nombre de port de connexions indépendants (problèmes d'intégration et de connectique).

L'utilisation d'un réseau reconfigurable dynamiquement pour établir ces liaisons "à la demande" suppose l'existence d'un allocateur de connexions logiques qui gère dynamiquement le partage des ressources de communications. C'est le cas des machines dont le réseau d'interconnexion est composé de commutateurs de circuits reconfigurables à la demande (ex : le Supernode est basé sur un réseau de Clos à 3 étages de "crossbar"). L'allocateur de connexion pilote le réseau d'interconnexion. Il doit donc toujours exister un moyen de communiquer entre une unité de travail souhaitant obtenir une connexion et l'allocateur de connexion. On distingue donc le réseau de contrôle, qui doit être disponible à tout instant pour transporter de petits messages de contrôle, du réseau de données dont les connexions sont temporaires et doivent permettre de transporter de gros messages (données).

Sur les petites machines qui comportent un nombre de processeurs de l'ordre de quelques dizaines, il est possible d'obtenir un accès uniforme (du point de vue coût de connexion et de transfert) à chacun des processeurs. C'est le cas de la machine Tnode que nous avons utilisée pour notre prototype : on peut connecter jusqu'à 36 processeurs ayant chacuns 4 liens de communication sur un "crossbar" réalisé spécifiquement par NEC pour le projet Esprit auquel nous avons participé.

Pour les machines comportant un plus grand nombre de processeurs, la réalisation d'un "crossbar" matériel devient très coûteuse voire impossible (nombre de point de connexion de l'ordre du carré du nombre de processeur). Il devient nécessaire de recourir à des réseaux plus complexes. De tels réseaux sont organisés en grappes de processeurs, les coût de connexions entre grappes sont plus élevés que ceux internes à une grappe. La stratégie de contrôle du parallélisme doit tenir compte de ce surcoût car il influe sur celui de l'installation d'une tâche selon qu'on l'installe ou non dans la même grappe de processeurs. Par ailleurs, la réalisation du contrôle peut nécessiter plusieurs niveaux et rendre ainsi difficile l'implantation centralisée de la stratégie de contrôle (cf chapitre 5). C'est le cas des machine Supernode haut de gamme (Les Meganodes peuvent comporter jusqu'à 1024 processeurs) qui offrent deux niveaux de coûts de transfert et de connexions selon que ces connexions sont internes à une grappe (Tnode) ou inter-grappes (passage dans le réseau de Clos à 3 étages de commutateurs). En outre, l'établissement des connexions physiques dynamiques est beaucoup plus lourd dans ce cas car il fait intervenir un grand nombre d'opérateurs matériels distribués dans la machine et le contrôle de ces opérateurs nécessite beaucoup de synchronisations. L'intérêt de la configuration dynamique est alors beaucoup moins évident, surtout si, comme dans le cas du Méganode, le contrôle de la reconfiguration du réseau d'interconnexion nécessite l'exécution d'algorithmes complexes.

b) Réseau physique statique

On peut s'affranchir de la contrainte d'un réseau (logique) complet en utilisant un réseau maillé statique connexe dont les nœuds seraient les unités de travail (par exemple tore) et en acheminant les messages de nœud en nœud depuis le nœud émetteur jusqu'au nœud récepteur (routage).

Les coûts de communication et donc d'installation de tâches dépendent alors de la distance des deux processeurs concernés. Si l'on évite par le routage la nécessité d'un allocateur de connexion, la stratégie de contrôle du parallélisme doit tenir compte des distances entre processeurs. Par ailleurs, si le routage n'est pas assuré sur chaque nœud par un coprocesseur spécifique, le calcul de chacun des nœuds intermédiaires est dégradé en proportion des données acheminées. Cette dégradation globale justifie l'introduction systématique de tels coprocesseurs dans la nouvelle génération de machines parallèles (IPSC2/T9000).

En l'absence de tels coprocesseurs, une solution simplificatrice consiste a n'utiliser que les connexions directes que procure la topologie de l'architecture cible. Cette solution permet d'utiliser au mieux la bande passante du médium de communication (point critique du fait de la taille importante des messages) car le découpage des messages en paquets et le contrôle de flux sont inutiles ; de plus on élimine les entêtes de messages que nécessite le routage. Par contre, elle oblige a utiliser un contrôle réparti plus difficile à mettre au point. De plus le nombre réduit d'interlocuteurs (voisins directs) diminue beaucoup l'intérêt d'utiliser de stratégies d'allocation de processeurs sophistiquées. Des fonctions de charge peu discrimantes peuvent alors être utilisées pour sélectionner le voisin vers lequel un processeur surchargé va se décharger.

c) bilan

Des caractéristiques des machines parallèles à mémoire distribuée, il ressort un certain nombre de points :

- La conception d'une machine abstraite Prolog doit autoriser le transfert de donnée par bloc (DMA) et l'asynchronisme entre transfert et calcul de façon à exploiter au mieux les coprocesseurs de communications.
- La conception d'une stratégie de contrôle doit prendre en compte les différents coûts de communication liés à l'architecture c'est à dire à la localisation des processeurs échangeant du travail
- L'implantation de cette stratégie dépend elle même de cette architecture. Une implantation distribuée devient nécessaire dès que les coûts des communications de contrôle deviennent importants.
- Du point de vue du modèle et de la stratégie, les problèmes de fond posés par la réalisation des communications entre unités de travail sont identiques.

Que se soit par routage ou par connexion bi-point dynamique, le prototype OPERA devait être implanté sur un Tnode de 16 à 32 Transputers. Nous avons réalisé deux

noyaux de communications [Favr89a] sur réseau de Transputers pour évaluer les coûts de transfert. Nous ne décrirons pas ces noyaux ici, le lecteur trouvera un brève description de ce système de communication dans l'annexe qui traite de l'environnement de programmation C parallèle et le petit système que nous avons développé à cette occasion. Nous avons finalement choisi une implantation par connexion dynamique du fait des coûts de connexions constants et de la possibilité de conserver une architecture centralisée pour le contrôle. Cependant, sur une architecture telle que celle du Meganode, cette simplicité ne peut être conservée ni pour la réalisation des communications bi-points ni pour l'implantation de la stratégie de contrôle.

4.3 Modèle d'OPERA - "Partie Opérative"

Le modèle d'exécution d'OPERA est le modèle multiséquentiel. Dans ce modèle, on considère un ensemble d'unités de travail toutes identiques. Chaque unité de travail comporte une unité de traitement (*Calculateur*) qui est une machine virtuelle TWAM ("Transputer Warren Abstract Machine"). Dans la version actuelle, chaque unité de traitement possède une copie complète du code du programme. Notons que cette contrainte est très liée au matériel utilisé et ne constitue pas réellement un trait essentiel du modèle.

Les techniques d'implémentations qui ont été retenues pour OPERA, sont basées sur des techniques de compilation efficaces (WAM) car un des objectifs du projet est de démontrer expérimentalement la supériorité des systèmes parallèles sur les systèmes séquentiels les plus performants. OPERA exécute les programmes séquentiels (déterministes) presque aussi rapidement que les systèmes commerciaux les plus rapides (Quintus Prolog sur sun 3-50, BIM Prolog).

4.3.1 L'unité de travail élémentaire : la TWAM

La TWAM est une adaptation de la machine virtuelle décrite dans le chapitre 1 spécialement conçue pour faciliter les opérations liées au parallélisme. Le préfixe T du sigle provient de la concrétisation de ce modèle d'unité de traitement sur un processeur : le Transputer.

Cette machine virtuelle est composée d'un jeu d'instruction opérant sur des registres et des structures gérés en plusieurs piles. Les sources des programmes Prolog sont compilés dans ce langage intermédiaire par un compilateur lui même écrit en Prolog. Ce compilateur est basé sur celui de Peter Van Roy de Berkeley [VanR84] auquel nous avons apporté quelques modifications : le parallélisme n'a d'influence que sur les structures de données de cette machine virtuelle (les structures de données de la WAM ont été adaptées au parallélisme) et sur la sémantique de certaines de ces instructions. Ceci permet de changer facilement de générateur de code sans remise en cause des principes d'OPERA.

La TWAM utilise principalement les instructions de la WAM mais en étend les structures de données pour prendre en compte le parallélisme. Le choix de ne pas

ajouter d'instructions supplémentaires est délibéré. Il a été fait pour garantir une plus grande compatibilité avec les systèmes séquentiels traditionnels et tenter de profiter ainsi de leurs améliorations.

Rappelons que la WAM utilise trois zones allouées en pile :

1. la pile `locale` pour stocker les environs (variables locales) des procédures et les points de choix (état des registres de la WAM avant d'évaluer une alternative),

2. la pile `globale` pour stocker les termes et plus généralement toute donnée persistante,

3. et enfin, la pile `traînée` qui permet d'enregistrer toutes les liaisons de variables des piles `locale` et `globale` en vue de leur déliaison lors du retour arrière.

La TWAM étend les structures de données de la WAM pour permettre une opération de copie des portions de piles la plus efficace possible ainsi que pour rendre plus souple la gestion du non-déterminisme. Les piles `locale` et `globale` de la WAM ont été réorganisées en quatres piles.

a) La pile de point de Choix

Contrairement à la WAM où les environs et point de choix sont entrelacés dans la pile `locale`, les points de choix sont gérés dans une zone spéciale de la TWAM appelée `pile de points de choix`.

Le principal avantage de cette solution réside dans le fait qu'on peut alors isoler la communication avec le monde extérieur : la gestion du non-déterminisme n'est plus étroitement liée à la stratégie d'allocation en pile des environnements donc à la stratégie de parcours de l'arbre ET (gestion du déterminisme). Cela simplifie le jeu d'instructions de la WAM. De plus cette modification ne dégrade en rien l'exécution séquentielle et est maintenant adoptée par des compilateurs pour machines séquentielles (SICSTUS Prolog).

Le second avantage est une meilleure gestion de la mémoire, car les points de choix peuvent être supprimés dès qu'il ne sont plus utilisés, ce qui n'est pas le cas dans la WAM puisque l'entrelacement avec les environs interdit de récupérer la zone mémoire occupée par les points de choix qui ne sont pas sur le sommet de pile. Dans le modèle OPERA, les points de choix sont organisés en liste doublement chaînée, permettant ainsi un degré de liberté beaucoup plus grand pour sa gestion (insertion / accès / suppression). Ce dernier aspect est très important pour la mise en place d'une stratégie de gestion du parallélisme OU.

L'accès à cette pile doit être partagé entre l'unité de traitement Prolog et l'interface avec son environnement. Une simple section critique suffit à maintenir la cohérence des accès à cet ensemble de points de choix : les accès aux registres définissant la file de points de choix se font en exclusion mutuelle.

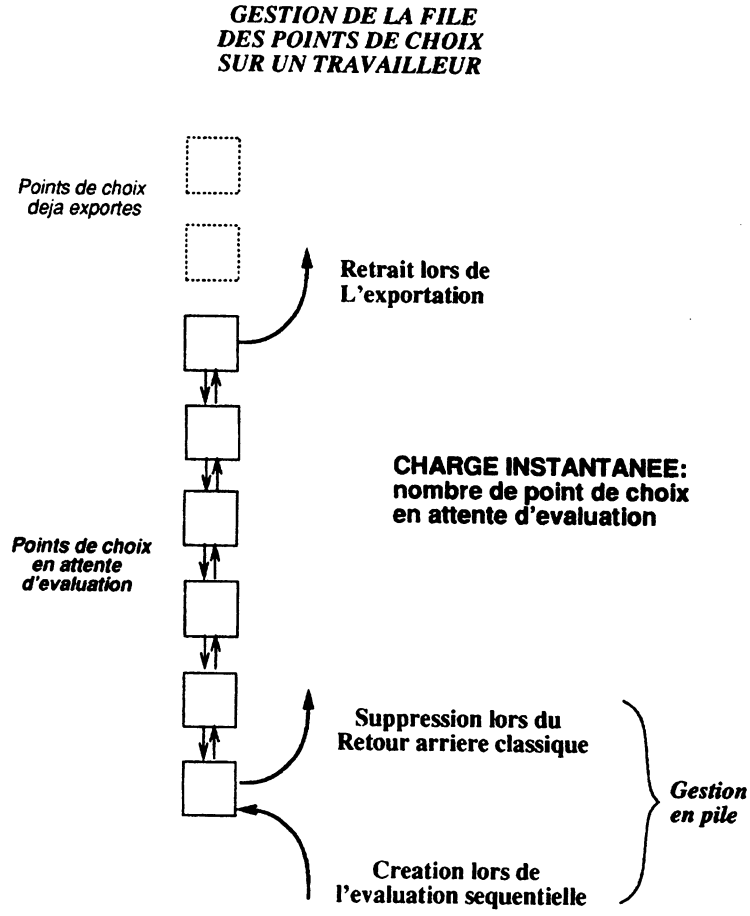


Figure 4.1 : File de points de choix

La pile des points de choix est gérée comme une file. Elle est représentée par une liste de points de choix doublement chaînée.

b) La pile Variable

La `pile variable` est une autre structure additionnelle de la TWAM qui permet de faciliter le traitement des variables dans le cas du parallélisme. Elle est utilisée pour stocker toutes les variables logiques. Dans la TWAM, on différencie les références à un terme des variables logiques alors que dans la WAM elles sont confondues et résident soit dans la pile `locale`, soit dans la pile `globale`, soit dans les registres.

Cette distinction provient du fait que les variables logiques peuvent avoir une représentation mémoire relativement importante si l'on considère leurs attributs : leur valeur (libre ou référence à un terme), leur tag, leur date de liaison (cf section 4.3.3).

Dans la TWAM, les variables sont accessibles via des pointeurs enregistrés dans les piles locale et globale. La récupération de l'espace dans cette pile est effectuée lors du retour-arrière. Chaque variable comporte deux cellules :

- la valeur (un pointeur sur le terme)
- la date de la liaison (valeur du registre CLOCK).

La partition de la pile globale (création de la pile variable) a pour objet de faciliter les opérations de déliaison des variables liées postérieurement au point de choix concerné par l'exportation. Pour ce faire, il suffit de parcourir la portion de la pile variable exportée et de défaire les liaisons invalides. Ces liaisons sont reconnues invalides grâce à la datation de la variable. Cette date est simplement comparée à la date associée au point de choix exporté. La date est simplement la profondeur dans l'arbre de recherche d'un point de choix c'est à dire un simple compteur incrémenté à la création d'un point de choix.

Le prix à payer est la place nécessaire pour représenter cette date dans le descripteur de variable, le coût accru d'une liaison (écriture de la date), le coût de déréférencement (une indirection de plus que pour la WAM). Ces coûts sont faibles et constants ; ils sont indépendants du degré de parallélisme. Par contre, on remplace le parcours exhaustif de la traînée pour trouver les liaisons de variables à défaire par un simple parcours de la portion de pile variable.

4.3.2 Copie des piles

Dans OPERA, la communication entre deux TWAMs correspond à l'émission par une TWAM de portions de piles à destination d'une TWAM inactive. La mise en oeuvre de cette communication doit exploiter au mieux les propriétés du réseau de communication entre processeurs qui sont

- une efficacité (débit) croissante avec le volume d'informations échangées,
- un "parallélisme" de la communication et du calcul (le processeur de communication fonctionne en mode DMA en volant des cycles de bus au processeur de calcul chaque fois qu'il accède à la mémoire).

STRUCTURE DE LA TWAM

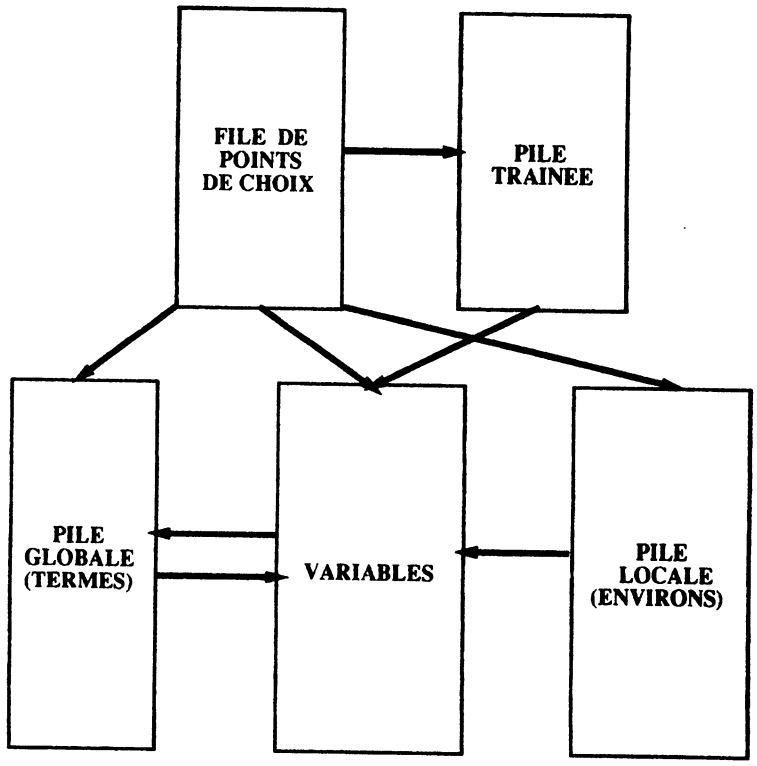


Figure 4.2 : Les zones mémoires de la TWAM

Le modèle OPERA utilise 4 piles et une file : la pile locale qui contient les nœuds ET (environs), la pile globale qui contient les termes, la pile variable qui contient les variables logiques, la pile traînée où sont enregistrées l'adresse des cellules modifiées et la file qui contient les noeud OU (points de choix). Les flèches indiquent les références entre ces zones de mémoire.

La première propriété provient du fait qu'une communication présente un coût d'amorçage important, dont le poids dans le coût global de communication décroît quand le volume d'informations croît. Ce coût d'amorçage est dû à l'opération d'initialisation des communications et à l'opération de composition des messages.

La seconde propriété est intéressante dès que "le vol de cycles mémoire" due à la communication ralentit de façon marginale le processeur. Ces propriétés sont partagées, à des degrés divers, par l'ensemble des architectures réseau de processeurs communicants.

Du fait de la grande efficacité de la copie de bloc mémoire par DMA, nous avons choisi de faire la copie complète des piles plutôt que de décoder les termes et de n'envoyer que ce qui est potentiellement accessible. Néanmoins nous pensons que ce type de copie de terme "intelligente" pourrait être utile dans plusieurs cas:

- en cas de saturation mémoire d'un processeur, si l'on veut utiliser un autre processeur libre comme extension de pile pendant que le processeur saturé effectue un algorithme de ramasse-miettes. Cette technique pourrait être utilisée pour copier les termes arguments du prédicat évalué sur le processeur d'extension.

- en cas (plus général) de répartition du code sur des sites différents , ou d'accès à un processeur spécialisé (dans les entrées/sorties par exemple). Cette technique serait particulièrement utile pour la mise en place d'un mécanisme de RPC (Remote Predicate Call).

- en cas de conversion de format dans un système hétérogène.

- enfin si le surcoût de la copie des données inutiles (inévitable dans la technique de copie en bloc) devient supérieur à celui du décodage de la copie "intelligente". Cette opportunité est difficile à déceler car ce n'est qu'à l'exécution que l'on peut savoir quels sont les éléments de termes effectivement utilisés.

Notre prototype n'utilise actuellement que la technique de copie en bloc mais nous avons cherché à réduire au minimum son impact sur le calcul séquentiel en ayant recours au parallélisme entre calcul et entrées/sorties. Le fonctionnement de la TWAM émettrice est le suivant:

Dès son élection (par la partie contrôle) comme exportatrice de travail, elle amorce l'émission des parties de piles concernées puis elle continue le travail en cours parallèlement au transfert.

La TWAM réceptrice procède de façon symétrique en amorçant les réceptions mais doit attendre la fin du transfert des piles pour exécuter le travail correspondant. L'exécution de ce travail doit être précédé de la déliaison des variables invalides. Cependant, si l'on prend la précaution de transférer en premier la portion de la pile variable, il est alors possible à la TWAM réceptrice de procéder à cette déliaison durant la réception des autres portions de piles (en fait ce traitement pourrait être effectué au vol moyennant l'ajout un matériel spécialisé)

Notons également que la technique de duplication des données permet éventuellement d'effectuer une conversion de format lors de la copie des données (réalisable

en pipeline de la copie), ce qui n'est évidement pas le cas lorsque les unités de travail se partagent la mémoire (donc la représentation des données). Cette conversion à la copie peut être nécessaire si le modèle OPERA est implanté sur une machine parallèle hétérogène (représentation mémoire dépendante de l'unité de travail).

L'indépendance du modèle vis à vis de l'hétérogénéité du support matériel peut être particulièrement intéressante si l'on conçoit une application Prolog parallèle comme un ensemble d'experts coopérants et si ces experts résident sur des sites aux propriétés différentes (exemple : réseau local de stations de travail, machine parallèle et machine frontale). Dans le cas plus simple de multi-programmation des processeurs physiques (plusieurs TWAM par processeur), la conversion à la copie peut se résumer à une simple translation d'adresse si le matériel ne la gère pas (cas des Transputer).

4.3.3 La technique de datation

La technique de datation est celle utilisée dans le modèle KABU-WAKE [Kumo86]. Le prototype construit a permis d'obtenir des performances absolues (temps d'exécution) relativement faibles Nous avons repris cette technique pour OPERA en l'améliorant pour permettre de copier les piles de la machine abstraite Prolog pendant leur modification (recouvrement des entrées-sorties par du calcul).

En ce qui concerne les variables, il est nécessaire que l'importateur travaille sur leur valeur à la date où a été crée le point de choix à évaluer. Les variables logiques étant à assignation unique, une variable possède un graphe de transition d'état très simple : initialement elle est libre (non affectée) puis est liée à un terme (affectée par un pointeur sur le terme). Restituer l'état d'une variable lors d'un retour arrière nécessite d'avoir mémorisé sa modification le cas échéant.

La technique traditionnelle de la pile dite "traînée" est particulièrement utile lorsqu'on effectue un retour arrière car les déliaisons sont effectuées en ordre chronologique inverse des liaisons. Dans le cas d'exportation d'un point de choix, les déliaisons à effectuer n'ont rien à voir avec l'ordre dans lequel elles ont été enregistrées dans la pile traînée.

Plutôt que d'effectuer une recherche dans l'ensemble de cette pile pour déterminer les déliaisons nécessaires, nous avons doublé le système de repérage des liaisons. La pile traînée a été conservée car elle reste le système le plus efficace pour le retour arrière séquentiel ; nous avons ajouté une date de liaison (attribut supplémentaire d'une cellule variable) qui vaut $+\infty$ si la variable n'est pas encore liée. On lui affecte la valeur du registre Clock au moment de la liaison. Clock est un compteur local à l'unité de travail qui vaut initialement 1 et qui correspond au niveau de profondeur du point de choix dans l'arbre OU. Le compteur Clock, faisant partie de l'état courant, est sauvegardé dans les points de choix. La datation des variables n'est d'aucune utilité pour le moteur d'évaluation séquentiel qui se contente de la mettre à jour mais, lors de l'importation, cette datation permet de restituer le bon état des variables. Une autre modification a été apportée à l'implémentation traditionnelle : les variables ont été regroupées dans une pile spéciale pour un parcours plus rapide lors de la recherche des déliaisons à effectuer lors d'une importation.

Exportateur		Accès concurrent	Transfert		Importateur	
Date	Valeur	par le *Calculateur*	Date	Valeur	Date	Valeur
$+\infty$	Libre	non	$+\infty$	Libre	$+\infty$	Libre
$d \leq D_{ref}$	$V \neq$ Libre	non	d	V	d	V
$d > D_{ref}$	$V \neq$ Libre	non	d	V	$+\infty$	Libre
$d > D_{ref}$	$V \neq$ Libre	Liaison	$> D_{ref}$	?	$+\infty$	Libre
$+\infty$	Libre	Déliaison	$> D_{ref}$	?	$+\infty$	Libre

Tableau 4.1 : Datation et cohérence du transfert

Les trois premières lignes correspondent au cas ou l'unité de transfert de l'unité de travail exportatrice est la seule a accéder aux portions de piles qu'elle émet. Les deux dernières lignes correspondent au cas ou l'unité de calcul de l'unité de travail exportatrice modifie (liaison, resp. déliaison) une cellule pendant que l'unité de transfert la lit et l'émet.

Les piles peuvent être modifiée pendant que l'on en copie une partie (cas des liaisons conditionnelles). La cohérence du transfert est assurée par la technique de datation. La continuation du calcul par l'émettrice pendant le transfert suppose que toutes les modifications de portions de piles en cours de transfert ne perturbent pas la cohérence de ces zones. Les seules modifications possibles sont les opérations de liaison, déliaison de variables, d'empilement ou de désempilement.

Liaison-Déliaison Les liaisons inconditionnelles ne posent pas de problème puisque les modifications de mémoire qu'elles occasionnent sont antérieures au transfert ; les valeurs de liaisons sont ensuite partagées en lecture seule entre les différentes branches d'évaluation.

La déliaison en réception des liaisons conditionnelles créées postérieurement à la partie exportée garantit la validité des liaisons importées (cf figure 4.3). Pour les variables liées conditionnellement, les modifications de mémoire peuvent être effectuées pendant que l'on transfère ce bloc mémoire. Les liaisons et déliaisons de telles variables ne sont susceptibles de poser problème que si ces opérations portent sur des variables en cours de transfert. Dans ce cas, la datation permet d'assurer la cohérence du transfert. Si l'on choisit de représenter la date sur un seul mot mémoire, son accès en exclusion mutelle (entre unité de transfert et unité de traitement) est garantit par l'arbitre du bus d'accès à la mémoire. Selon notre convention de codage des variables libres, une variable libre est une variable qui est liée à l'infini ($+\infty$). Cette convention permet de pouvoir délier en réception des variables inconditionnelles même dans le cas où la lecture du champ valeur a été effectuée pendant sa modification par l'unité de traitement.

Le tableau 4.1 montre que quelle que soit la valeur reçue (même altérée) si la date de liaison est postérieure à la date de référence (celle du point de choix reçue) la liaison

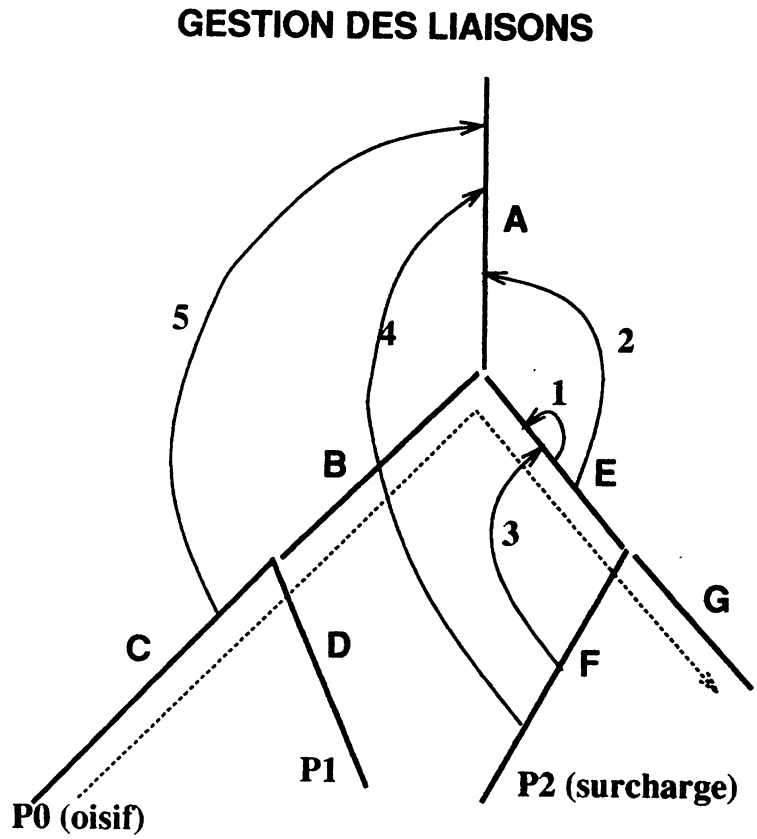


Figure 4.3 : Gestion des liaisons

Les arcs flechés représentent les liaisons. L'origine d'un tel arc est le segment de pile qui correspond à l'environnement courant au moment de la liaison. Le segment pointé par l'extrémité flechée correspond à l'environnement où a été créée la cellule représentant la variable. Lorsque le processeur P0 termine l'évaluation de C, il effectue un retour arrière en invalidant les liaisons de type 5. Ensuite il importe A et E. Les liaisons de type 1 et 2 sont conservées telles qu'elles ont été importées alors que les liaisons de type 3 et 4 sont invalidées après importation car leur date de liaison est postérieure à la date d'importation.

La copie incrémentale est une optimisation qui consiste à n'importer que E (puisque la plus grand partie de A est commune) ainsi que les modifications valides de A (liaisons de type 2) Les liaisons ne type 3 sont invalidées en utilisant la date. Les liaisons de type 4 ne sont pas importées.

doit être invalidée.

Les opérations élémentaires sur les variables sont alors :

```
type ( Valeur , Date ) Var_logique;

void init( Var_logique * Variable )
debut
   * Variable := ( Variable , infinity ) ;
fin

void lier( Var_logique * Variable , Valeur )
debut
   * Variable := ( Valeur , Clock ) ;
fin

/* lors du retour arriere sequentiel */

void delier_local( Var_logique * * Trail )
debut
  * (* Trail) := ( * Trail , infinity ) ;
  Trail := Trail - 1 ;
fin

/* lors du retour arriere parallele (importation) */

void delier_nonlocal( Var_logique * Variable , Date_Ref )
debut
  si Variable->date >= Date_Ref
    * Variable := ( Variable , infinity ) ;
  Variable := Variable + 1 ;
fin
```

Empilement-Désempilement Pour les empilements, ils sont nécessairement postérieurs au point de choix exporté donc ne font pas partie des zones transférées. Les désempilements vérifient la même propriété sauf dans un cas particulier. Ce cas correspond à la situation où la TWAM exportatrice termine sa résolvante avant la fin du transfert et où la seule résolvante en attente est celle en cours de transfert. Il y a alors conflit entre retour arrière et exportation sur la dernière clause du dernier point de choix.

Le conflit peut être résolu de deux façons, chacune exigeant une synchronisation spécifique :

1. Une première solution est que la tâche exportatrice récupère pour elle le travail exporté (retour arrière normal). Elle doit donc signifier à la TWAM réceptrice un échec du transfert. Cette solution nécessite que l'architecture puisse interrompre une copie en DMA. Si ce n'est pas le cas, on doit découper les messages en plusieurs paquets pour reprendre régulièrement le contrôle et tester l'opportunité du transfert. L'inconvénient de cette méthode est d'ajouter un temps de contrôle important (diminution de la bande passante de transfert) même si le transfert est valide.

2. Une seconde solution est que la tâche émettrice demande du travail à une autre tâche. Dans ce cas, elle doit différer l'amorçage des transferts d'importation après la terminaison de l'exportation en cours garantissant ainsi la préservation des portions de piles en cours de transfert.

4.3.4 La technique de la traînée avec valeur

Cette technique est une alternative à la technique de datation. Outre l'adresse des variables liées conditionnellement, chaque entrée de la pile traînée comporte un champ dans lequel on duplique la valeur assignée à la variable. Ceci est une optimisation car on pourrait très bien retrouver cette valeur dans le tableau de liaisons par indirection sur l'adresse enregistrée dans la pile traînée classique. Le champ valeur de la traînée n'est pas accédé par les opérations utilisées dans la stratégie séquentielle. En fait cette optimisation n'est introduite que pour faciliter la reconstruction incrémentale d'une branche d'évaluation. Cette reconstruction est assurée par une nouvelle opération élémentaire: la "reliaison" dont l'algorithme suit.

```
void relier( Pid , Trail[ Pid , ptr_Trail[Pid] ] )
debut
 (IndexBA,(Tag,Val)) := Trail[Pid,ptr_Trail[Pid]];
  ptr_Trail[ Pid ] := ptr_Trail[ Pid ] + 1;
  BA[ Pid , IndexBA ] := (Tag,Val);
fin
```

La technique de la traînée avec valeur a pour inconvénient de nécessiter un parcours de la pile traînée pour effectuer les liaisons. Or, dans cette pile, les liaisons portant sur d'autres zones mémoire sont mélangées à celles qui portent sur la zone à traiter. Une taille de pile traînée importante défavorise donc cette méthode, par contre une taille de pile variable importante peut défavoriser la méthode de datation si le nombre de variables à traiter est faible devant le nombre de variables à parcourir.

4.3.5 Optimisations de la copie

Dans notre premier prototype, lors de l'échange de travail entre deux unités de travail, la totalité des piles est copiée. Plus les piles sont de taille importante, plus la durée de copie est importante. Cependant, la technique de copie peut être optimisée :

a) Copie incrémentale

Comme on l'a vu, l'exécution d'un programme Prolog peut être représentée par un chemin dans arbre OU depuis la racine de l'arbre. Lors d'un transfert de travail entre processeurs, l'importateur doit avoir une description exacte de la partie du chemin à partir de laquelle il démarre son parcours. Deux parcours quelconques ont donc nécessairement une portion de chemin en commun.

Dans la WAM, les données utilisées par le programme sont organisées en piles qui croissent avec la profondeur dans l'arbre. Si le processeur importateur a déjà participé à la résolution d'une partie du programme, il dispose en mémoire d'une partie des piles du processus exportateur. Il suffit donc de transférer les zones non communes.

Cette méthode, également proposée dans [Ali90], offre l'avantage de réduire la taille de pile à copier. Elle n'est possible que si les représentations mémoire de la partie commune sur chacun des deux processeurs sont strictement identiques, ce qui n'est pas indispensable dans la méthode de copie complète.

La détermination des portions de piles à transférer est un problème simple à résoudre lorsque l'on travaille en mémoire commune car chacun des processeurs dispose de l'accès à l'ensemble de la représentation de l'arbre d'évaluation. Par contre, dans le cas du modèle OPERA où l'on s'interdit d'utiliser de la mémoire commune, le problème est plus complexe. La détermination des portions de piles à transférer ou, ce qui revient au même, la détermination de la partie de pile commune ne relève pas de l'unité de travail qui ignore tout de ses congénères mais de la partie contrôle qui gère les appariements (unité exportatrice, unité importatrice). La seule conséquence directe sur la partie opérative est la nécessité, pour celle-ci, de fournir à la partie contrôle, lors de chaque transfert, les informations pertinentes (dont les tailles de piles) pour que cette dernière puisse maintenir un historique des échanges déjà effectués nécessaire pour sélectionner les échanges les moins coûteux à réaliser. Ces informations sont restituées à la partie opérative (le cas échéant) pour permettre l'émission ou la réception de parties non communes.

b) Copie paresseuse

La technique de copie paresseuse consiste à ne copier les données que lorsque cela est vraiment nécessaire. On réduit le coût de copie en réduisant le volume des données transférées. La copie paresseuse d'éléments de terme est de grain trop faible pour être utilisable sur les architectures actuelles. La copie paresseuse de bloc de mémoire nous paraît être beaucoup plus réaliste, la technique de pagination en est la preuve.

Si l'architecture cible dispose d'une unité de gestion mémoire (MMU), l'utilisation de ce type de mécanisme peut être envisagé. En l'absence de dispositif matériel appro-

prié (cas des Transputer) le problème revient à savoir si une donnée est déjà présente en mémoire (dans l'espace d'adressage local à l'unité de travail) ou s'il est nécessaire d'effectuer une copie d'une donnée distante. La solution d'émuler un MMU est irréaliste car le gain de temps obtenu par la réduction des tailles transférées serait probablement négligeable devant le ralentissement considérable des très nombreux accès élémentaires à la mémoire.

Tout comme la technique de pagination, la technique de copie paresseuse que nous proposons ci-après repose sur l'hypothèse d'une stabilité de comportement des programmes Prolog: du fait de la nature du langage Prolog, la plupart des accès à une pile sont proches du sommet de pile. Cette hypothèse est réaliste car les variables des prédicats Prolog n'ont qu'une portée locale à une clause et la grande majorité des chaînes de références sont de longueur 1. On peut donc faire l'hypothèse que, pour chacune des piles de la machine abstraite Prolog, la probabilité d'accès diminue avec la distance entre la cellule considérée et le sommet de pile. Cette règle pourrait d'ailleurs être utilisée dans un algorithme de remplacement de pages.

Pour pallier à l'absence d'une MMU, on pourrait simplifier en utilisant un registre par pile qui scinderait la pile en deux zones : la portion du sommet de pile (initialement vide) qui contient les données déjà copiées et la portion du fond de pile qui permettrait de réserver l'espace d'adressage nécessaire à l'accueil éventuel de données complémentaires. Chaque opération de copie de bloc rapprocherait ce pointeur du fond de pile. L'avantage de cette technique naïve est d'être extrêmement simple et d'introduire un surcoût limité (à condition que le matériel se charge de la comparaison de chaque accès à une pile et du registre correspondant). Du point de vue temps, comparer une adresse au registre limite est d'un coût voisin de celui de la consultation d'un tableau de pages déjà présentes. Du point de vue place la solution du registre barrière est plus économique que la solution qui consisterait à maintenir un tableau de toutes les pages indiquant pour chaque entrée, si la page est présente et, dans le cas contraire, sur quel nœud aller la chercher. Un autre avantage de cette technique est qu'il n'est pas nécessaire que les blocs échangés soient de taille fixe: cela laisse toute latitude pour la technique de copie et on pourrait imaginer de réduire cette taille pour les blocs éloignés du sommet de pile.

On pourrait également améliorer cette technique en permettant un déréférencement non local: lorsque le registre barrière se retrouverait loin du sommet de pile les accès à la valeur ultime d'une chaîne de référence pourraient être traités de manière distante en ne ramenant que la valeur ultime sans modifier la barrière (lecture distante sans mémorisation locale) s'il s'agit d'une lecture ou en forçant la copie jusqu'à ce point s'il s'agit d'une écriture (avec, dans ce dernier cas, l'avantage de factoriser les transferts de blocs ce qui diminue le surcoût de contrôle).

Un problème lié à la copie paresseuse est que l'accès à une valeur peut devenir bloquant. Par conséquent cette technique n'est intéressante que dans la mesure où elle est utilisée conjointement

- à la multi-programmation d'un processeur (en plaçant plusieurs TWAMs par processeur),

- à l'utilisation d'une forme de préchargement pour amener le paquet de pages "immédiatement utilisées"

Quel que soit la technique employée pour l'implanter, la copie paresseuse repose sur l'hypothèse d'une stabilité du comportement en mémoire des programmes Prolog. La technique de copie incrémentale repose une propriété du problème: toute évaluation de programme OU parallèle correspond à un parcours d'arbre. Cette dernière technique est difficilement implantable dans le matériel (à moins de vouloir construire une architecture spécialisée) alors que la première peut reposer sur des mécanismes qui existent déjà sur les architectures comportant des MMU. Il est intéressant de noter que les deux techniques ne sont pas exclusives.

c) Exportations multiples (diffusion).

Enfin on peut envisager d'exporter simultanément des points de choix vers plusieurs unités de travail inactives. Mise à part l'initialisation qui permet de choisir la répartition des points de choix exportés entre les unités destinataires, le traitement peut s'effectuer en parallèle puisqu'il s'agit d'opérations de lecture. Cette technique doit permettre d'améliorer notablement les performances si l'architecture cible permet la diffusion, notamment lorsque le nombre de TWAMs inactives est plus important que celui des TWAMs surchargées.

4.4 Méta-contrôle

Le langage supporté par la première version d'OPERA est le Prolog "standard" incluant la coupure et quelques prédicats prédéfinis pour les entrées/sorties. Les prédicats Prolog permettant de gérer les bases de faits et de clauses (prédicats prédfinis du type assert/retract) ne sont pas pris en compte dans notre prototype car la technique de compilation utilisée dans OPERA se marie difficilement avec les prédicats qui modifient le programme dynamiquement. Les prédicats de base de données "record" ne sont pas pris en compte non plus dans cette version car les effets de bord qu'ils occasionnent posent des problèmes de synchronisation dans le cas du parallélisme OU.

4.4.1 Les effets de bord

Faute de temps, nous n'avons pas pu traiter les prédicats à effets de bords dans la version actuelle d'OPERA (le règlement général de ce problème nécessite un gros effort de codage). Voici, rapidement, quelques idées directrices quant à la manière dont nous comptons aborder et résoudre le problème.

Les prédicats à effet de bords sont en général soit des prédicats prédéfinis dont l'évaluation provoque la modification de l'état d'un l'objet de l'environnement soit des prédicats dont la valeur dépend de celle de l'état d'un l'objet de l'environnement. L'exemple le plus simple est celui des prédicats entrées/sorties: le premier cas peut être illustré par le prédicat **write(X)** qui modifie implicitement l'état du flot de sortie, le

prédicat de lecture de terme **get(X)** illustre le second cas. Dans le même ordre d'idée, les prédicats de gestion de la base des clauses font partie des prédicats dits à "effets de bord". La caractéristique commune à tous ces prédicats est qu'ils lisent et/ou modifient l'état global du système.

Le problème qui se pose aux concepteurs des systèmes multiséquentiels peut s'énoncer ainsi :

Comment fournir au programmeur un ensemble de prédicats prédéfinis cohérents respectant la sémantique qui existait dans l'exécution séquentielle ?

Garantir le respect de l'ordre séquentiel dans les modifications de l'état global est fondamental pour que le programmeur puisse prédire le résultat de son programme (qui est lui même obtenu par effet de bord). Que l'exécution se fasse en séquence ou en parallèle au gré de la stratégie OU du système, le résultat doit demeurer le même. Pour reprendre notre exemple, les sorties doivent arriver dans le même ordre que celui obtenu par l'exécution du programme sur un seul processeur.

Ce problème se ramène au problème classique de l'accès concurrent à une ressource commune (l'état de l'objet à modifier). Les unités de travail ont alors soit le rôle de lecteur, soit le rôle d'écrivain, selon la nature du prédicat qu'elles exécutent. Plusieurs lecteurs peuvent accéder à la ressource en même temps et mais un seul écrivain y accède à un instant donné. Cependant, l'ordre des lectures/écritures doit respecter l'ordre défini par la stratégie séquentielle. Pour ne pas bloquer trop longtemps les unités de travail, on ne considère pas l'environnement comme une ressource unique, mais on divise l'état global en autant d'états locaux qu'il y a de ressources indépendantes, chacune de ces ressources étant gérée par un processus qui lui est propre.

L'exécution séquentielle fournit un ordre total (le temps) sur les requêtes de lecture/écriture soumises à une ressource donnée. L'exécution multi-séquentielle (potentiellement parallèle) fournit autant de flots de requêtes qu'il y a de parties d'exécution séquentielle ; chacun d'eux est ordonné selon l'ordre total précité. La solution au problème consiste à synchroniser ces flots de requêtes concurrents (à un instant donné il y en a au plus n si n est le nombre d'unités de travail) en respectant l'ordre total sauf dans le cas des lectures qui peuvent être simultanées. Ce dernier point donne à la solution parallèle un petit avantage sur la solution séquentielle.

Le problème de la gestion cohérente des effets de bord dans l'environnement multiséquentiel OPERA peut donc se réduire au problème suivant : à chaque instant il faut maintenir dans le système une structure permettant de savoir quelle est, parmi les unités de travail émettrices d'un flot de requêtes, celle qui est la première dans l'ordre séquentiel. En d'autre terme, quelle est l'unité de travail qui évalue la branche la plus "à gauche" dans l'arbre d'évaluation. La gestion de cette structure globale relève de la partie contrôle car c'est cette dernière qui maintient l'état de l'arbre OU du programme. Lors de l'exécution d'un prédicat à effet de bord, la partie opérative fait appel à la partie contrôle qui, au vu de cette structure, établira une connexion logique entre l'unité de travail et le processus serveur gestionnaire de la ressource.

Pour clore cette section, nous dirons simplement que, bien que cet aspect n'ait pas été abordé dans notre réalisation, nous avons conçu l'architecture de notre prototype de manière à pouvoir y inclure ultérieurement ce type de mécanismes. Une description

plus précise d'une réalisation peut être trouvée dans [More91].

4.4.2 Le contrôle par la coupure

La description détaillée de la méthode utilisée pour l'implémentation des diverses variations de l'opérateur de coupure ("cut") dans un environnement parallèle exigerait l'écriture d'un chapitre complet. Ici, une description simplifiée de cette méthode est présentée.

Le problème est d'inhiber le parallélisme lors du franchissement d'une coupure afin d'éviter que les branches de l'arbre OU qui peuvent être éliminées par cette coupure ne soient évaluées inutilement. L'idée de base pour résoudre ce problème consiste à compter les clauses ayant des coupures à franchir. Un compteur, matérialisé par le nouveau registre CC, est incrémenté chaque fois que l'on entre dans un prédicat qui contient au moins une coupure. Ce compteur est sauvegardé 2 fois (B.CC et B.CCclause) dans le point de choix. Une de ces copies est utilisée comme compteur local à un prédicat, elle est modifiée à chaque alternative et à chaque coupure. Ce compteur local indique si le parallélisme est possible ou pas (nous n'effectuons pas de calcul spéculatif sur des branches susceptibles d'être coupées plus tard).

Par rapport à l'implémentation séquentielle, cette méthode introduit les surcoûts suivants :

- l'incrémentation du compteur de clauses "coupées" au début de chaque prédicat contenant une coupure,
- le traitement du compteur et des sauvegardes respectives à chaque nouvelle alternative et à chaque coupure.

Il faut noter que, dans le cas des instructions du groupe **try**, les coûts existent aussi pour les prédicats "non-coupés". Le lecteur intéressé par une description détaillée de la méthode peut se reporter à la thèse de Claudio Geyer [Geye91].

Chapitre 5

Partie contrôle, régulation de charge

5.1 Structure du Chapitre

Ce chapitre décrit les problèmes du contrôle du parallélisme et plus particulièrement le problème de la régulation de charge. Dans la section 5.2, nous montrons que le gain de performance qu'il est possible d'obtenir par le parallélisme est limité d'une manière générale et qu'un degré de parallélisme trop élevé peut provoquer une baisse d'efficacité. Dans la section 5.3 nous examinerons le problème de la limitation du degré de parallélisme pour garantir un gain de performance. Nous dégageons deux conditions déterminant l'opportunité du parallélisme et posons le problème de la régulation de charge à partir de ces conditions. La section se termine par une proposition d'architecture permettant le maintien de l'état global exact du système sur chaque nœud du réseau de processeurs en vue de faciliter le contrôle du parallélisme. Différentes solutions au problème du placement des processus de contrôle y sont discutées. La section (5.4) décrit le cas de Prolog parallèle et les choix qui ont été faits pour l'ordonnancement des tâches dans le système OPERA.

5.2 Limite du gain de performance dû au parallélisme

Nous allons examiner le problème de l'exploitation du parallélisme dans le but d'augmenter l'efficacité globale du système (c'est l'un des principaux objectifs d'OPERA). Nous ne considérerons l'efficacité que du point de vue temporel c'est-à-dire le gain de performance obtenu par rapport à une exécution séquentielle. Nous allons montrer très simplement que, pour une certaine classe de problèmes sur un type d'architecture de machines multiprocesseurs, le parallélisme ne peut pas fournir de gain de performance linéaire. Il existe une borne à ce gain de performance lorsqu'on augmente le nombre

de processeurs utilisés pour résoudre un même problème et qu'au delà de cette borne on peut provoquer une baisse de performance. Nous allons montrer ceci dans le cas d'une architecture "idéale" du type réseau de processeurs sans mémoire commune. De même, nous considérerons que le problème se prête lui même de façon idéale à la parallélisation. Dans un deuxième temps, nous examinons plus concrètement en quoi les hypothèses sont "idéales" et quelles sont les conséquences du point de vue des machines et problèmes réels.

5.2.1 "Modèle" de machine parallèle

Les hypothèses sur l'architecture matérielle (HMi) sont les suivantes :

HM1 On dispose d'un nombre de processeurs aussi grand qu'on veut. Tous les processeurs sont identiques.

HM2 Chacun des processeurs possède une mémoire locale de taille infinie. Pour un site donné, cette mémoire a un accès uniforme du point de vue coût (en d'autre terme il n'existe pas de hérarchie mémoire sur un site : ni cache, ni même registres).

HM3 Chaque processeur possède un nombre fini (d) de voies de communications (lien série, bus , etc...) permettant de le connecter à d'autres processeurs. Par la suite, on désignera par "lien" ce médium de communication en faisant abstraction de la manière dont il est réalisé.

HM4 Chacun des liens entre processeurs est bidirectionnel c'est à dire qu'il permet la transmission de messages simultanément dans un sens et dans l'autre.

HM5 Un processeur peut émettre et recevoir en parallèle par ces liens.

HM6 La durée du transfert est une fonction f croissante de la taille transférée t telle que $\forall t, f(t) > K$ où K est une constante strictement positive.

5.2.2 "Modèle" de programme parallèle

Les hypothèses (HPj) sur la parallélisation du problème sont les suivantes :

HP1 Le problème à résoudre est de taille finie c'est à dire que son exécution séquentielle se termine. On suppose qu'il est décomposable en un nombre quelconque (aussi grand que l'on veut) de tâches indépendantes (i.e. sans relation de précédence).

HP2 Toutes les tâches sont activables dès l'instant $T0$ où on commence à résoudre le problème : en d'autre termes, aucune tâche n'apparaît après l'instant $T0$.

HP3 A l'instant initial $T0$, toutes les ressources (code+données) nécessaires à la résolution du problème sont présentes sur un processeur unique nommé processeur racine.

HP4 Conventionnellement, on décrète la fin de la résolution du problème lorsque tous les processeurs ont terminé d'exécuter le sous-ensemble des tâches qui leur a été alloué. Cela implique qu'il existe un mécanisme de détection de la fin globale (dont la durée d'exécution est nulle).

HP5 Toutes les tâches doivent être exécutées.

5.2.3 Limite au gain de performance obtenu par parallélisation

Définition préliminaire: La **charge exacte** à l'instant t d'un processeur est la durée entre cet instant de référence t et l'instant où le processeur termine l'évaluation de l'ensemble des tâches qui lui a été assigné (sachant qu'aucune autre tâche ne lui sera assignée ultérieurement). La charge exacte est donc une fonction du temps positive strictement décroissante et qui finit par s'annuler. Par la suite nous simplifierons en utilisant le terme **charge** pour **charge exacte.**

Le problème : La question est: "sous les hypothèses énoncées précédemment, quel gain de performance maximum peut-on espérer en parallélisant c'est à dire en distribuant les tâches sur les processeurs ?"

La réponse peut se présenter de la forme suivante: initialement, on dispose de l'ensemble des tâches (HP1) sur un même processeur racine (HP3). Celui-ci dispose de suffisament de ressources mémoires (HM2) pour exécuter en séquence toutes les tâches. On notera $T_{seq} = T_1$ le temps pris pour résoudre le problème de cette façon.

Les tâches étant indépendantes (HP1) et activables dès le début (HP2), l'ordre d'allocation des tâches au processeur peut être quelconque. On peut également décider de ouper l'ensemble des tâches en au plus $d+1$ sous-ensembles de manière à répartir la charge entre le processeur racine et ses d voisins directs. Tous ces processeurs peuvent à leur tour distribuer une partie du travail reçu à l'étape précédente. Le travail à faire peut donc se diffuser de proche en proche dans le réseau.

Si l'on note $T_{par} = T_n$ le temps mis pour résoudre le travail sur n processeurs et C_n la durée des transferts nécessaires à la diffusion du travail sur ces n processeurs, alors, dans le meilleur des cas (répartition équitable des tâches entres les processeurs) on a:

$$T_n = \frac{T_1}{n} + C_n$$

Le gain de performance S_n, obtenu par utilisation de n processeurs, s'exprime donc sous la forme:

$$S_n = \frac{T_{seq}}{T_{par}} = \frac{T_1}{T_n} = \frac{nT_1}{nC_n + T_1}$$

Il dépend donc du coût de diffusion du travail sur les n processeurs.

a) à coût de diffusion borné

Si l'on admet que la suite C_n converge vers une limite C quand le nombre de processeurs tend vers l'infini, alors

$$\lim_{n\to\infty} S_n = \frac{T_{seq}}{C}$$

Le gain de performance croît asymptotiquement en fonction du nombre de processeurs jusqu'à une limite dépendante du rapport temps de calcul sur temps d'entrées-sorties. Ce rapport est lié au problème et aux caractéristiques de l'architecture.

b) à coût de diffusion croissant

La convergence de C_n vers une limite finie implique la possibilité de diffuser en un temps fini une information depuis un processeur vers un nombre infini d'autres processeurs. En d'autres termes, le coût élémentaire de communication décroît à l'inverse de nombre de processeurs ce qui est contradictoire avec l'hypothèse HM3 (limitation de la bande passante) et HM6 (coût minimum d'un transfert élémentaire: composition du message et amorçage du transfert).

Si l'on prend en compte l'hypothèse HM6, C_n croît toujours avec n d'au moins K. Le gain de performance ne peut donc croître indéfiniment et la parallélisation finit par devenir une source de ralentissement: en effet, la fonction S_n admet un extrémum. La dérivée de S_n

$$S'_n = \frac{(T_1 + nC_n)T_1 - (nC'_n + C_n)nT_1}{(nC_n + T_1)^2} = \frac{T_1}{(nC_n + T_1)^2}(T_1 - n^2C'_n)$$

s'annule pour $T_1 - n^2C'_n = 0$ car l'hypothèse HM6 implique que

$$\forall n, C'_n > 0$$

c) degré de parallélisme optimum

Il existe donc un nombre optimum n_{opt} de processeurs tel que le gain de performance est à son maximum. Ce nombre est tel que

$$T_{seq} = n^2_{opt}C'_{n_{opt}} \qquad (1)$$

Il dépend de la taille du problème à résoudre (plus celle-ci est importante, plus n_{opt} est grand) et de la capacité de diffusion de l'architecture matérielle.

En reprenant les hypothèses sur le matériel HM3 à HM6 on peut donner une idée de la croissance du coût de diffusion:

- en 0 pas on atteint 1 processeur
- en 1 pas on atteint au plus $1 + d$ processeurs
- en p pas on atteint au plus

$$1 + d + \ldots + d^p = \sum_{i=0}^{i=p} d^i = \frac{d^{p+1} - 1}{d - 1}$$

processeurs (dans le cas où la topologie du réseau de processeur est un arbre).

Le coût global de diffusion est de la forme:

$$C_{\left(\frac{d^{p+1}-1}{d-1}\right)} = pK$$

donc

$$C_n = O(K \log_d n)$$

$$C'_n = O\left(\frac{K}{n \log d}\right) > 0$$

Dans ce cas, l'équation (1) permet de déterminer n_{opt} :

$$n_{opt} \simeq \lfloor \frac{T_1 \log d}{K} \rfloor$$

Dans le cas plus simple où l'on dispose d'une architecture à bus unique sans possibilité de diffusion ($d = 1$) les n processeurs ne sont atteints qu'en $n - 1$ pas de durée K soit :

$$C'_n = K \Rightarrow n_{opt} = \lfloor \sqrt{\frac{T_1}{K}} \rfloor$$

d) bilan

La conclusion à tout ceci est la suivante : "Le parallélisme n'est pas toujours source d'un gain de performance"

A problème constant, pour un grand nombre de processeurs, les échanges entre processeurs et, plus généralement, la gestion du parallélisme ont un coût prépondérant par rapport au gain apporté par la parallélisation.

Des gains "quasi-linéaires" sont toujours obtenus pour un nombre faible de processeurs et sont d'autant plus faciles à obtenir que le temps d'exécution séquentiellement est élevé. Pour un même algorithme abstrait, plus sa réalisation (son implantation sur une machine réelle) est de mauvaise qualité plus la courbe du gain de performances S_n est proche de la fonction y = x. En d'autres termes, une implantation efficace donnant un temps séquentiel T inférieur de X fois à celui d'une implantation inefficace, offre

des possibilités de gains de performances inférieures à celle offerte par l'implantation inefficace car :

$$X > 1 \Rightarrow \frac{S_n(XT)}{S_n(T)} = \frac{nXT}{nT} \cdot \frac{nC_n + T}{nC_n + XT} = \frac{XnC_n + XT}{nC_n + XT} > 1$$

Par conséquent la "quasi-linéarité" du gain de performance n'est pas en soit, la garantie de la qualité d'un système parallèle. Au voisinage de n_{opt}, le gain de performances augmente très peu. Rares sont les publications concernant Prolog Parallèle où figurent les gains de performances en fonction du nombre de processeurs et les performances absolues. Il faut également tenir compte des performances en valeur absolue relativement à l'architecture cible. En particulier, le rapport temps de calcul sur temps d'entrées-sorties est un critère important. Tant que le ratio T_n/C_n reste supérieur à celui qui caractérise l'architecture cible, l'application gagnera à être encore parallélisée ; en dessous de cette limite, la parallélisation est inopportune. Cette limite intrinsèque de l'architecture permet de définir le grain minimum des tâches. Pour OPERA, nous avons choisi la technique de compilation plutôt que la technique d'interprétation car nous voulions vérifier nos hypothèses de travail dans un cadre réaliste jugé significatif de l'état courant des machines parallèles à mémoire distribuée.

5.2.4 En pratique

Les hypothèses retenues dans la section 5.2.1 correspondent au cas idéal. Dans la réalité, il faut nuancer :

HM1 on ne dispose que d'un nombre limité de processeurs : actuellement, de quelques dizaines à quelques centaines pour les machines MIMD basées sur des processeurs 32 bits. Ce nombre de processeurs borne le parallélisme physique donc le gain de performance que l'on peut espérer. Dans le cas où le découpage du problème en tâches est explicite, qu'il ne peut être remis en cause, et que le nombre total de tâches est supérieur au nombre de processeurs disponibles, chaque processeur doit exécuter plusieurs tâches et on doit recourir à la multiprogrammation. Celle-ci nécessite une quantité de mémoire importante et coûte plus cher que l'exécution en séquence des tâches (celle-ci n'est pas toujours possible). Par contre, la multiprogrammation est souvent utilisée pour recouvrir les temps d'entrées/sorties par du calcul dès que l'on dispose de processeurs d'E/S fonctionnant en parallèle avec ce calcul. Dans ce cas, la multi-programmation est un facteur d'amélioration car elle diminue le coût des communications supporté par les processeurs de calculs.

HM2 chacun des processeurs ne dispose que d'une quantité finie de mémoire. La saturation temporaire des ressources peut introduire des retards dans l'évaluation de certaines tâches. La capacité limitée de la mémoire joue sur le fait qu'on ne peut pas installer en bloc toutes les tâches exécutables. Certaines tâches parallèles doivent être sérialisées artificiellement pour tenir dans la mémoire et, à la limite, une tâche peut

être découpée en une suite de sous-tâches (d'où une multiplication des amorçages de transfert).

HM3-4-5 chaque processeur possède un nombre limité de voies de communications (4 liens de type série dans le cas des Transputer). Ceci implique que le diamètre des graphes de processeurs que l'on peut construire augmente rapidement avec le nombre de processeurs. Dans le cas général, l'installation des tâches et leurs communications ne se réduisent pas à une diffusion 1 processeur vers n processeurs de tâches indépendantes connues a priori ; la topologie en arbre de degré $d-1$ n'est donc pas toujours la plus intéressante. La capacité de diffusion de la machine est moindre par rapport au cas envisagé dans la section précédente; cela réduit d'autant la borne n_{opt}. De plus, un processeur ne peut émettre et recevoir réellement en parallèle sur ces d liens que si ses ports sont des unités autonomes qui accèdent à une mémoire permettant plusieurs accès simultanément. Dans le cas des Transputers le bus externe unique constitue une sérialisation obligatoire bien que les accès se fassent par des unités de DMA indépendantes.

HM6 la durée du transfert élémentaire (entre 2 nœuds voisins) est une fonction f généralement de la forme

$$f(t) = K + Dt$$

où K est une constante non nulle qui représente le coût d'amorçage du message (ex : constitution d'un entête) et D une constante définissant le débit d'un lien. Le degré limité des nœuds (processeurs) du réseau d'interconnexion nécessite d'acheminer des communications entre tâches non voisines. Ces routes intermédiaires peuvent être empruntées par les communications issues de plusieurs paires de tâches communicantes. Ce multiplexage peut nécessiter le découpage des messages en paquets et leur réassemblage. Une communication avec routage intermédiaire présente un accroissement apparent du coefficient K avec la distance entre le nœud source et nœud destination ainsi qu'un débit moindre (gestion des données de routage, contrôle de flux...). Ce débit peut être dégradé en fonction du taux de conflits sur les portions de route communes entre sites communicants. Par ailleurs, le fonctionnement des nœuds intermédiaires (vitesse de calcul) peut être fortement dégradé si le routage n'est pas assuré par un processeur spécialisé.

HP1-3 Le problème à résoudre est composé d'un nombre fini de tâches : on travaille dans un domaine discret où la tâche élémentaire est la modification d'un bit. Les tâches ne sont pas indépendantes et il existe toujours une relation de précédence (arbre dans le cas de Prolog) plus ou moins contraignante ; en d'autre terme, toutes les tâches ne sont pas activables dès l'instant $T0$.

HP4 Dans un réseau de processeurs, la détection de la fin globale de l'évaluation du programme parallèle a une durée d'exécution qui ne peut être négligée.

5.2.5 Des gains super-linéaires

Jusqu'à maintenant, nous avons raisonné à "problème constant" (HP5) comme si toutes les tâches devaient systématiquement être exécutées, quelques soient les circonstances. Cependant, l'hypothèse (HP5) peut ne pas être vérifiée dans certains cas et dans ces cas la parallélisation peut permettre des gains largement super-linéaires. Il s'agit

- soit de cas prévisibles car l'annulation de tâches est "spécifiée" par le programmeur
- soit ces gains sont inattendus car les tâches éliminées par parallélisation sont des tâches générées implicitement par la compilation et/ou le système d'exploitation. Ces tâches qui portent sur la gestion des ressources matérielles sont, en général, invisibles du programmeur.

Pour illustrer notre propos examinons deux exemples :

a) Cas de la recherche de solutions

La résolution de problèmes combinatoires pour lesquels on ne cherche qu'un sous-ensemble de l'ensemble des solutions. est un exemple du premier cas de figure. Pour ce type de problème, l'exploration exhaustive de toutes les voies de recherches n'est pas nécessaire et, en changeant la stratégie de recherche, on peut obtenir plus rapidement le sous-ensemble souhaité et économiser ainsi le parcours d'un grand nombre d'autres voies de recherche. La parallélisation de ce type de problème peut donner un gain de performance "super-linéaire" car le parallélisme est lui même source d'une variation de stratégie. Dans le cas du parallélisme OU en Prolog, la compétition des processeurs pour évaluer le méta-prédicat "une-solution" permet souvent de trouver une solution beaucoup plus rapidement que si un seul processeur ne l'avait cherché seul. L'obtention de la première solution (dans le temps) rend caduques les autres voies de recherches : certaines tâches ne sont donc pas exécutées. On peut même obtenir un gain "infini" dans le cas de programme comportant une branche infinie : la stratégie séquentielle conduit au parcours (infini) de cette branche alors que la stratégie multi-séquentielle peut permettre à un autre processeur de trouver la solution recherchée et de terminer ainsi le programme.

b) Gains super-linéaires liés au système

Les ressources locales accessibles par un processeur n'ont pas un accès uniforme du point de vue coût : les registres sont accessibles plus rapidement que la mémoire cache qui est, elle même, plus rapide que la mémoire principale... L'augmentation de la quantité globale de ces ressources critiques (par utilisation de plusieurs processeurs) peut accroître considérablement les performances : en particulier les effets de cache peuvent permettre l'obtention de gain de performance "super-linéaires". Dans une machine parallèle, le nombre plus important de ces ressources critiques permet d'éviter les tâches implicites qui consistent à faire migrer des données entre les niveaux de hiérarchie de stockage.

L'exemple (naïf) qui suit illustre ce type de phénomène. Supposons que l'on exécute le programme suivant sur une machine comportant un seul registre et que les opérateurs nécessitent la présence de l'opérande dans le registre.

```
t tableau de P entier
for( j = 0; j < N; j++ )
  for( i = 0;  i < P; i++ )
    t[i] = t[i] * t[i]
```

Sur un seul processeur la variable t[i] devra être chargée en registre puis remise en mémoire après multiplication pour permettre le traitement de l'élément suivant. Cela représente une complexité de l'ordre de $N \times P$ tâches de migration d'élément à traiter (migration de type mémoire-registre et registre-mémoire).

En supposant que l'on dispose de P processeurs identiques ayant chacun un seul registre et que l'on ait parallélisé l'algorithme en distribuant chaque élément de tableau et le traitement qui s'y rapporte sur chaque processeur

```
for( j = 0; j < N; j++ )
  t = t * t
```

Cette distribution permet de maintenir chaque élément en permanence dans le registre du processeur correspondant. On a ainsi, économisé tous les chargements / sauvegardes de registres (à l'exception du chargement initial). La complexité est alors de l'ordre de 1 chargement de registre pour 1 processeur donné. Remarque : cet exemple n'a d'autre intérêt que son but pédagogique ; la parallélisation automatique de langage comme Fortran peut engendrer ce type de phénomène.

Dans le cas de cache de pages, le coût important de transit d'une page entre mémoire principale et mémoire secondaire accentue ce phénomène. Ce cas est analogue à ce qui se passe en séquentiel : pour un problème donné lorsque l'on diminue la taille du cache les fautes de pages deviennent tellement nombreuses que l'on observe parfois un écroulement des performances bien supérieur au facteur de diminution de la taille du cache.

Les algorithmes de gestion mémoire sont un autre exemple de tâches implicites dont l'exécution éventuelle est liée à la disponibilité des ressources. Dans les cas ou la parallélisation correspond à une augmentation de la quantité de ressources disponibles pour exécuter un même [1] programme, les conditions de déclenchement des algorithmes ramasse-miettes sont modifiées et le fonctionnement global peut être nettement moins dégradé qu'en séquentiel.

5.3 Problème général de la régulation de charge

Le problème de la régulation de charge peut s'énoncer ainsi :

Un nombre n fini de processeurs exécutent des tâches de durée finie. L'exécution d'une tâche engendre un nombre indéterminé d'autres tâches. Ces tâches sont indépen-

[1] au sens du code écrit par l'utilisateur

dantes. On prend pour origine du temps $T0 = 0$ et on note $T(i,n)$ ($i \in [1..n]$) l'instant où le processeur i termine la dernière tâche qui lui a été allouée. On a

$$T_{seq} = T(1,1) \; et \; T_{par}(n) = \max_{i \in [1..n]} T(i,n)$$

$T_{par}(n)$ est minimum lorsque $\forall i \in [1..n], \forall j \in [1..n], T(i,n) = T(j,n)$, ce qui correspond à une charge équilibrée.

En l'absence d'une connaissance particulière sur la durée du problème à résoudre, une heuristique de placement de tâches sur les processeurs consiste à essayer de maintenir en permanence la situation d'équilibre pour que l'on puisse s'arrêter dans une situation proche de la situation idéale. Cette heuristique sous entend que le problème à résoudre peut être découpé en sous-problèmes un grand nombre de fois. C'est le cas des problèmes Prolog OU parallèles puisque, potentiellement, chaque point de choix est une opportunité de répartition.

La régulation de charge consiste à apparier un processeur sous-chargé et un processeur surchargé pour équilibrer localement la charge. Une fois sélectionnée la paire de processeurs, on procède au découpage du problème en cours d'évaluation sur l'unité surchargée puis au transfert vers l'unité oisive de la partie découpée. L'opération élémentaire de régulation de charge est donc la combinaison de deux phases :

1. La sélection globale d'une paire de processeurs (surchargé,oisif) parmi l'ensemble des processeurs.

2. La sélection locale d'une partie à exporter parmi les tâches en attente d'évaluation par le processeur surchargé.

Nous discuterons l'aspect "sélection globale" dans la section 5.3.2, examinons tout d'abord les contraintes que doit respecter la règle de sélection locale.

5.3.1 Conditions de parallélisation

L'exportation d'une tâche du processeur surchargé vers un processeur oisif présente un coût dépendant de la tâche transférée. Ce coût se répartit non uniformément sur les processeurs exportateur, importateur et éventuellement leur environnement (coût induit par le conflit d'accès au réseau de communication). Pour simplifier, on ignorera ce dernier coût dans ce qui suit.

Le problème de la sélection locale s'énonce ainsi :

"Comment garantir que l'exportation d'une tâche vers un autre processeur produise un accroissement de performance par rapport à la solution laissant l'exécution d'une tâche au processeur qui l'a créée ?"

De manière générale, l'exécution de deux tâches, présentes initialement sur la même unité de travail, ne doit être réalisée en parallèle par cette unité de travail et son environnement que si le coût d'évaluation de la tâche exportée est supérieur au surcoût nécessaire (localement) à son exportation et, symétriquement, que si le coût d'évaluation de la tâche restante est supérieur au coût d'importation supporté par l'environnement.

La deuxième condition moins intuitive peut s'interpréter par: il faut éviter de donner un travail qui fera défaut plus tard ou, plus exactement, ne pas exporter un travail qui ne sera pas effectué avant que l'exportateur ne redevienne lui-même inactif donc demandeur de travail.

Plus formellement:

Soit T_l le coût de la tâche locale, T_ϵ celui de la tâche exportable, E le surcoût de l'exportation, I celui de l'importation. La première condition s'écrit $T_e > E$, la seconde $T_l > I$.

$$on\ a\ (T_e > E) \Leftrightarrow (T_l + T_e > T_l + E)$$

$$et\ (T_l > I) \Leftrightarrow (T_l + T_e > T_e + I)$$

$$soit\ ((T_e > E) \wedge (T_l > I)) \Leftrightarrow (T_l + T_e > max(T_l + E, I + T_e))$$

c'est à dire que $(T_e > E \wedge T_l > I)$ est nécessaire et suffisante pour assurer que l'exécution parallèle $T_{par} = (max(T_l + E, I + T_e))$ soit plus efficace que l'exécution séquentielle $T_{seq} = (T_l + T_e)$.

Par la suite, nous noterons C_{min} la condition $(T_e > E)$ et C_{max} la condition $(T_l > I)$ pour rappeler que la quantité T_ϵ de travail à exporter doit être telle que

$$E < T_\epsilon < T_{seq} - I$$

où T_{seq} est la durée totale d'évaluation restant à faire dans le cas où le processeur surchargé résout seul le problème.

Voici une autre formulation de ces conditions:

$$(T_e > E \wedge T_l > I) \Rightarrow (T_e + T_l > E + I)$$

La contraposée s'interprète par "si le temps total des transferts et des créations/destructions de processus $(E+I)$ est trop important devant les temps de calcul effectif, le parallélisme n'est plus justifié". On retrouve ainsi la limite inférieure du grain de parallélisme évoquée en 5.2.3.

Nous avons dégagé deux conditions C_{min} et C_{max} qui définissent localement l'opportunité de parallélisation. Le problème qui subsiste est le suivant:

"comment déterminer le couple de processeurs à apparier ?"

5.3.2 Maintenance d'un état global exact du système

La règle d'appariement est appliquée chaque fois qu'il y a conjonction entre les événements "il existe un processeur sous-chargé" et "il existe un processeur surchargé" (indépendamment de l'ordre dans lequel se produisent ces évenements). Cette règle de sélection d'une paire de processeurs porte sur un ensemble de processeurs. Cela suppose

GRAIN DE PARALLELISME

Tl : duree d'execution de la tache locale
Te : duree d'execution de la tache exportee
E : surcout local d'exportation
I : surcout d'importation

$$(Te > E) \Leftrightarrow (Tl+Te > Tl+E)$$
et
$$(Tl > I) \Leftrightarrow (Tl+Te > Te+I)$$

donc
$$(Te>E) \;\&\; (Tl>I) \Leftrightarrow (Tl+Te > \max(Tl+E, I+Te))$$

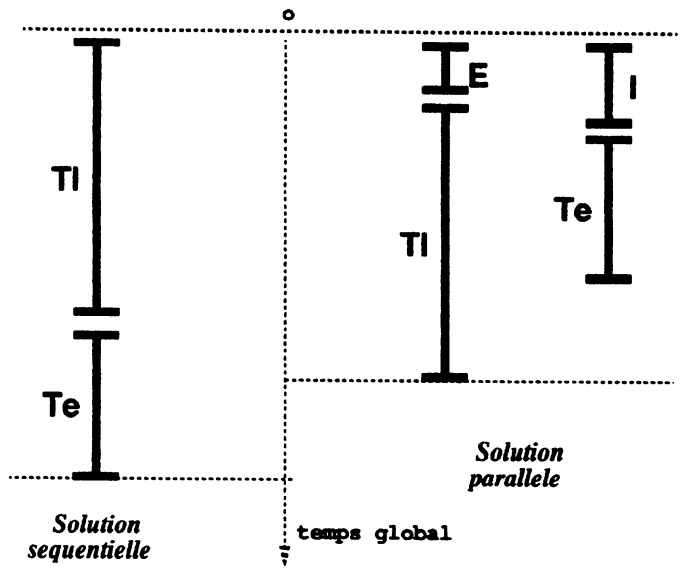


Figure 5.1 : Conditions de parallélisation

que le processeur qui exécute cette règle de sélection, dispose de l'état de charge de cet ensemble c'est à dire de la charge de chacun des autres processeurs.

Si les machines à mémoire commune facilitent le maintient d'un état global accessible par tous les processeurs (cf section 5.4.1), il n'en est pas de même pour les machines sans mémoire commune. Pour celles-ci, l'état global réel $E = (e_1, e_2, \ldots, e_n)$ est réparti car constitué des états locaux $e_i (i \in [1..n])$ des n processeurs. Prendre une bonne décision d'appariement suppose de disposer de E lors de la prise de décision.

5.3.3 Solution centralisée

Une première solution est de "simuler" le partage de mémoire en rassemblant dans la mémoire locale d'un site spécialisé (unité de contrôle) les différents états locaux e_i des autres sites (unités de travail). A chaque transition d'état local $e_i \rightarrow e'_i$, les processeurs envoient un message à l'unité spécialisée qui met alors à jour sa représentation de l'état global. Après cette phase d'acquisition des états locaux une décision peut être prise. La suite des décisions d'appariement de processeurs peut être gérée très simplement par un algorithme séquentiel.

Examinons le coût d'un cycle (acquisition de l'état global, prise de décision). Quel que soit le moyen d'acheminement des messages contenant les états locaux vers l'unité spécialisée dans la prise de décision, le coût de la phase d'acquisition de l'état global exact est limité par la capacité d'absorption de l'unité spécialisée : elle doit recevoir n états locaux par ces d ports. Au mieux l'acquisition est faite en parallèle ce qui donne un temps de l'ordre de $\frac{n}{d}$ dans le meilleur des cas (pas de conflits d'accès au médium de communication). Pour cette solution centralisée, seule une décision pourra être prise à un instant donné : cette décision provoque une modification de l'état global ; la représentation de ce dernier devra alors être réactualisée au cycle suivant.

Il est clair que le cycle (acquisition de l'état global, prise de décision) règle la vitesse du système. C'est un point critique du point de vue de l'efficacité du système car ce temps de cycle est une composante des coûts d'importation et d'exportation : un temps de cycle important augmente le grain minimum (cf conditions de parallélisation) et limite beaucoup le nombre des opportunités de parallélisation qui garantissent un accroissement de performances. La solution centralisée n'est donc viable que si le temps de service de l'unité spécialisée est faible devant le grain des tâches à répartir sur les unités de travail.

5.3.4 Solution répartie

Une autre solution consiste à augmenter le nombre de décisions prises simultanément dans un même cycle. Cette solution consiste à dupliquer l'état global sur les n sites pour pouvoir prendre n décisions en un seul cycle. Dans cette solution il n'est pas nécessaire de disposer d'unités spécialisées, on maintient sur chaque processeur une structure de données représentant l'état global E pour qu'il soit disponible immédiatement lorsqu'on exécute (localement) la règle de sélection globale. Si chaque unité dispose du même état global E et possède un moyen d'auto-identification alors il est possible de prendre en

un seul cycle n décisions cohérentes : en effet, chaque unité peut prévoir les choix qui sont effectués pendant ce cycle par les autres unités et utiliser son identificateur pour paramètrer la fonction de sélection globale. La cohérence des décisions nécessite que la fonction de sélection globale soit déterministe. Pour cette solution la durée de la phase de décision(s) est du même ordre que pour la solution centralisée. Elle a pour avantage d'augmenter la fréquence globale des décisions de régulation de charge.

Examinons maintenant la phase acquisition d'état dans le cas réparti. Le maintien de cet état global suppose qu'à chaque transition d'état local $e_i \rightarrow e'_i$ le processeur i diffuse sa modification d'état vers tous les autres processeurs. Son exactitude suppose que toutes les transitions de tous les processeurs soient prises en compte : cela signifie qu'à chaque transition d'une partie de E, on doit connaître exactement l'état des autres parties. En conclusion, le maintien d'un état global exact sur chaque processeur nécessite qu'à chaque transition locale, chaque processeur fournisse a tous les autres son nouvel état (diffusion "tous-vers-tous").

Le coût de ce schéma de communication globale est extrêmement dépendant de la topologie de l'architecture cible : il croît avec le diamètre du graphe de processeur. Un graphe complet a pour avantage de fournir E en 1 pas. La réalisation d'un tel graphe pour un nombre important de processeurs est encore techniquement impossible.

Pour relier un nombre important de processeurs, on doit recourir à des réseaux d'interconnexions et à des techniques de routage et/ou de reconfiguration. Le problème qui consiste à trouver un graphe de diamètre k qui contient le maximum de processeurs ayant d ports de connexions chacun, est un problème très complexe connu en théorie des graphes sous le nom de "*problème du graphe* (d, k)" [Memm82]. On ne connaît de solution à ce problème que pour quelques valeurs particulières de (d, k). Les topologies optimales ne sont, en géneral, pas régulières et sont très différentes selon les valeurs de (d, k). Il ne semble pas intéressant d'en exploiter les propriétés particulières si l'on prend en compte l'objectif initial de portabilité de Prolog sur une gamme de machines facilement extensible.

5.3.5 Opérateur de diffusion "tous vers tous"

Nous proposons ci-après une méthode de construction de réseaux d'interconnexion reconfigurables qui permet de réaliser le schéma de communication "tous-vers-tous" nécessaire au maintien de l'état global sur chaque processeur. Nous montrons que l'algorithme proposé est optimal du point de vue temps et taux d'occupation des unités de transfert. Cette architecture à mémoire distribuée permet l'interconnexion d'un grand nombre de processeurs tout en ayant un nombre de port d limité.

L'idée principale est de reconfigurer par phases le réseau d'interconnexion pour augmenter le nombre de voisins "directs" d'un processeur. Du point de vue matériel, il faut disposer d'un couple multiplexeur/démultiplexeur de petite taille (< 10) par port bidirectionnel de processeur. Le contrôle de la sélection de la voie multiplexée/démultiplexée étant assuré localement : lors de la détection d'une marque de fin de message sur la voie courante on reconfigure en passant à la voie suivante. Le nombre de voies du multiplexeur i est l'ordre du graphe.

Ordre	Diam.	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$
$i=1$	1	2	3	4	5	6	7	8
$i=2$	3	6	21	52	105	186	301	456
$i=3$	7	42	903	8164	44205	173166	543907	1456008
$i=4$	15	1806	1631721	19990852	...	...	...	...

Tableau 5.1 : Graphe G(d,i) : nombre de processeurs interconnectés

La technique d'acheminement des messages d'un nœud vers un autre est un mélange de routage et de reconfiguration : à chaque phase de reconfiguration on établi des portions de route qui permettent aux messages de se rapprocher de leur destination finale. Il s'agit de routage déterministe.

La figure ci-après (5.2) illustre la méthode de construction des graphes sur des exemples simples où le nombre d de ports des processeurs est égal à 1 (resp. 2) et où le nombre de voies du multiplexeur/démultiplexeur est 0, 1 et 2.

De manière générale, on construit un graphe d'ordre i à partir d'un certain nombre de sous-graphe d'ordre $i-1$ considérés comme des macro-nœuds. En ajoutant une voie supplémentaire à chaque multiplexeur/démultiplexeur (à $i-1$ entrées) on crée un réseau complet entre les macro-nœuds (cf Figure 5.3).

La table suivante donne une idée de la croissance de la taille des graphes (en nombre de processeurs) par rapport à la taille des multiplexeurs/démultiplexeurs (i) et aux nombre de ports de chaque processeur (d) :

Le problème de la diffusion "tous-vers-tous" dans le graphe $G(d,i)$ peut se ramener au problème de la diffusion "tous-vers-tous" dans les graphes $G(d,i-1)$ qui le compose. En effet, diffuser 1 unité d'information de chaque sommet du graphe $G(d,i)$ vers chacun des autres sommets de ce graphe peut être décomposé en 3 étapes :

1. Diffuser 1 unité d'information de chaque sommet du graphe $G(d,i-1)$ vers chacun des autres sommets de ce même graphe. Cette opération est appliquée simultanément à tous les sous-graphes $G(d,i-1)$ dont l'union constitue le graphe $G(d,i-1)$. Cette première phase correspond à la diffusion "tous-vers-tous" dans un graphe d'ordre inférieur (à l'ordre $i-1$). On notera donc $T(d,i-1)$ son temps d'exécution ; l'unité de temps est la durée de transfert d'une unité d'information élémentaire (l'état local d'un sommet). Pendant cette étape le réseau d'interconnexion des sommets peut évoluer dynamiquement par utilisation de connexion d'ordre j telles que $j \in [1..i-1]$. Les connexions d'ordre i ne sont pas nécessaires.

2. A la fin de l'étape 1, tous les nœuds d'un même graphe $G(d,i-1)$ sont équivalents car ils ont tous le même état cumulé. Cet état cumulé représente l'état global du graphe $G(d,i-1)$ correspondant : sa taille est donc de $N(d,i-1)$ unités élémentaires. La deuxième étape consiste à "diffuser partiellement" les états des sous-graphes $G(d,i-1)$ entre eux ce qui correspond à une transmission bidirectionnelle simultanément à travers les $d \times N(d,i-1)$ arcs d'ordre i issus

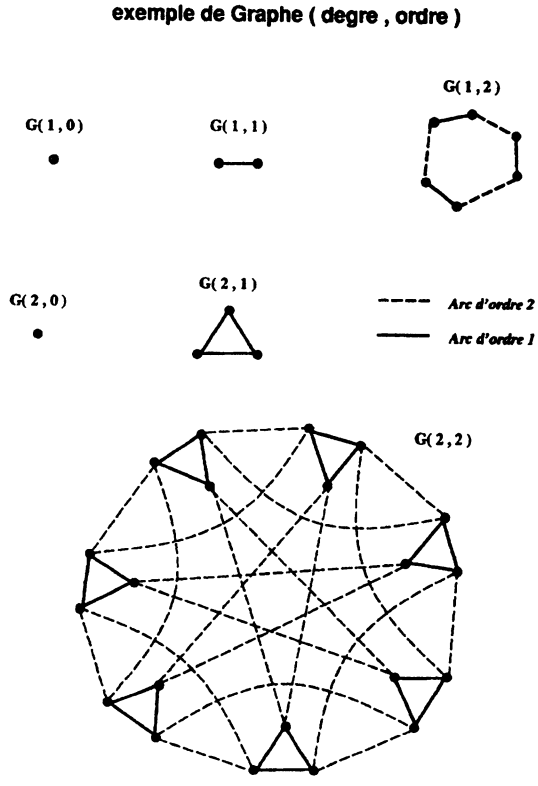


Figure 5.2 : opérateur de diffusion

Chaque sommet du graphe représente un processeur et les multiplexeurs / démultiplexeurs associés à chacun de ses d ports de communication. Les connexions figurées en pointillés représentent un état de connexion différent de celui représenté par les traits pleins. Quel que soit l'état de configuration courant, un processeur possède d voisins directs.

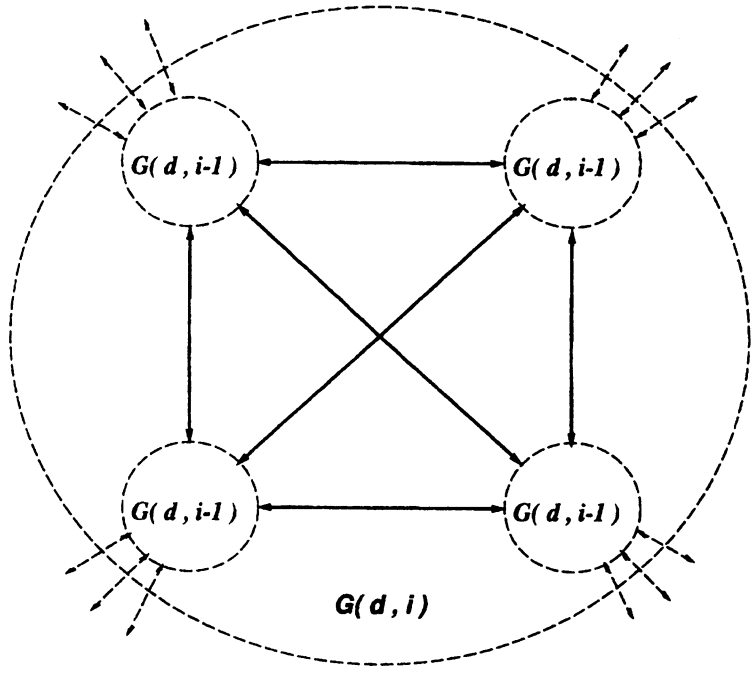


Figure 5.3 : méthode de construction de l'opérateur de diffusion

L'architecture proposée est hiérarchique. On peut accéder en un pas aux nœuds accessibles par les arcs d'ordre 1. Plus l'ordre du graphe i est important plus le nombre de nœuds est grand et plus le nombre de phase de configuration ("diamètre" du graphe) augmente. Les processeurs d'une même grappe $G(d, i)$ sont privilégiés pour les communications point à point.

des graphes $G(d, i-1)$. Pendant cette étape seules les connexions d'ordre i sont utilisées. La durée de l'étape 2 est donc $N(d, i-1)$ unités de temps.

3. Après l'étape 2 tous les sommets des sous-graphes $G(d, i-1)$ ont reçu une partie différente de l'état global. En fait sur chaque sommet d'un sous-graphe $G(d, i-1)$, on a alors l'état global de ce même sous-graphe $G(d, i-1)$ obtenu à l'étape 1 et l'état global de d sous-graphes $G(d, i-1)$ connecté à ce sommet par des arcs d'ordre i durant l'étape 2. L'étape 3 consiste à diffuser "tous-vers-tous" ces $N(d, i-1)$ états, chacuns d'eux étant de taille $d \times N(d, i-1)$. L'étape 3 dure donc $d \times N(d, i-1) \times T(d, i-1)$ unités de temps.

Pour tout graphe d'ordre $i > 0$, on a donc

$$T(d, i) = T_{etape1} + T_{etape2} + T_{etape3}$$

avec

- $T_{etape1} = T(d, i-1)$
- $T_{etape2} = N(d, i-1)$
- $T_{etape3} = d \times N(d, i-1) \times T(d, i-1)$

Remarque : Les sommets du graphe $G(d, i)$ ayant un degré égal à d, il est nécessaire de procéder à une reconfiguration entre l'étape 1 et l'étape 2, de même qu'entre l'étape 2 et l'étape 3. Les étapes 1 et 3 peuvent elles mêmes donner lieu à un certain nombre de reconfigurations si $i > 2$. Pour simplifier, nous négligerons les durées nécessaires à l'établissement de nouvelles connexions.

Pour le graphe d'ordre 0 qui correspond à un seul sommet, le problème de la diffusion "tous-vers-tous" ne se pose pas puisque cet unique sommet possède déjà l'état global du graphe $G(d, 0)$ donc $T(d, 0) = 0$.

En résumé : Le temps $T(d, i)$ pris pour une diffusion "tous-vers-tous" dans un graphe $G(d, i)$ est tel que :

$$\forall\, d \geq 0\ , \begin{cases} \forall i > 0, T(d, i) = T(d, i-1) + N(d, i-1) + d \times N(d, i-1) \times T(d, i-1) \\ T(d, 0) = 0 \end{cases}$$

En sommant les égalités, on obtient la formule générique suivante :

$$\forall\, d \geq 0\ ,\ \forall i \geq\ 0, T(d, i) = \sum_{j=0}^{j=\,i-1} N(d, j)^2$$

La formule précédente montre que le temps de diffusion "tous-vers-tous" augmente très rapidement avec l'ordre de graphe. Cette formule traduit le fait que cette famille

de graphes de processeurs est très dense. En fait, on peut montrer (en repartant des égalités qui définissent $T(d,i)$ et par récurrence sur i), que le temps de diffusion "tous vers tous" est directement proportionnel au nombre de processeurs total du graphe c'est-à-dire que l'on se trouve dans un cas d'optimalité dans la mesure où, à chaque instant, chaque ressource matérielle (port de processeur) est utilisée pour véhiculer une information utile (i.e. qui n'a pas encore été reçue : pas de duplication de messages).

$$\forall\, d > 0\ ,\ \forall i \geq\ 0, T(d,i) = \frac{N(d,i) - 1}{d}$$

conclusion : Avec l'opérateur de diffusion "tous vers tous" que nous proposons, la phase d'acquisition de l'état dans la solution distribuée est optimale (elle n'est limitée que par la capacité d'absorption d'un nœud). La solution distribuée offre donc le même temps de cycle que la solution centralisée avec pour avantage sur cette dernière un plus grand nombre de décisions prises dans un même cycle.

5.3.6 Maintenance d'un état approché

En pratique, les performances en calcul des processeurs d'aujourd'hui sont très importantes devant les performances des coprocesseurs d'entrées-sorties (même pour les Transputer). On ne peut pas envisager de faire fonctionner un grand nombre de processeurs de manière synchrone en maintenant un état global exact en permanence sur chaque nœud. Il y a plusieurs raisons à cela :

- Le nombre de transitions d'états dans le système est en général très élevé ; de plus, il augmente avec le nombre de processeurs. Par conséquent, plus le nombre de processeurs est grand, plus le nombre de cycles de contrôle (acquisition d'état, décision de régulation de charge) augmente. Or nous avons montré que la durée de la phase d'acquisition d'état croît au moins en proportion du nombre de processeurs. On peut raisonnablement penser qu'il en est de même pour la complexité de la fonction de sélection globale. Si le nombre de cycles de contrôle et leur durée augmente en proportion du nombre de processeurs alors que les capacités d'entrées-sorties et de traitement de chaque processeur restent les même (constantes architecturales) on ne peut pas maintenir exact l'état global et augmenter indéfiniment le nombre de processeurs : cette croissance du coût du contrôle de la parallélisation impose un grain minimal de plus en plus gros alors que le nombre croissant de processeurs implique un découpage du problème de plus en plus fin !

- L'asynchronisme des machines MIMD que nous visons se prête mal à la reconstruction d'une exécution en phases synchrones (contrôle,calcul,contrôle,...). La partie opérative est en général plus efficace que la partie contrôle car on lui consacre beaucoup plus de ressources. Les transitions d'états d'une partie opérative sont donc très nombreuses et toutes ne sont pas forcément significatives pour le contrôle surtout si ce dernier a été simplifié en vue de diminuer son temps de

service. L'exactitude de l'état est inutile si le contrôle est basé sur des heuristiques très approximatives.

- L'asynchronisme entre partie contrôle et partie opérative est inévitable. L'état de la partie opérative peut donc évoluer entre le moment ou celle-ci fourni une information à la partie contrôle et ou celle-ci la reçoit et la traite. Cette déformation temporelle de l'image du système est d'autant plus importante que le médium de communication qui la véhicule est lent. Autrement dit, à amplitude constante, une transition d'état d'une unité de calcul a d'autant moins d'importance qu'elle est loin dans le temps (donc dans l'espace) de l'instant où elle est prise en compte par une unité de contrôle. On peut tirer parti de l'asynchronisme entre partie contrôle et partie opérative pour recouvrir la durée du cycle de contrôle par du calcul. L'anticipation des décisions peut permettre de réduire le surcoût de contrôle perçu par les unités de travail et diminuer ainsi le grain minimum exploitable donc augmenter le degré de parallélisme optimum (n_{opt}).

De ces remarques on peut tirer la conclusion suivante: Lorsqu'on augmente le nombre de processeurs, il est plus efficace de maintenir un état approché du système et d'appliquer des heuristiques simples que de maintenir un état exact qui permet des stratégies de contrôle sophistiquées.

La réduction de la durée du cycle de contrôle peut être obtenue par

- simplification de la stratégie de contrôle qui ne porte plus que sur une information très synthétique (donc approximative)

- simplification de la phase d'acquisition d'état en diminuant le nombre de transitions prises en compte. Cette diminution peut être obtenue en ne sélectionnant qu'un sous-ensemble des états (e_i) ce qui diminue le volume de données à tranférer dans un cycle : par exemple on pourrait choisir de ne consulter que les voisins directs. Plus généralement, l'acquisition de l'état peut être simplifiée si l'on tient compte de l'amortissement de l'amplitude des transitions en fonction des coûts de transmission

Sur les machines sans mémoire commune, il faut se contenter de travailler sur des "images approchées" de l'état global du système obtenues par échantillonnage des états locaux. Ceci complique sérieusement les algorithmes par rapport à ce qu'il est possible de réaliser sur une machine parallèle comportant de la mémoire partagée: pour ce type d'architecture, l'accès atomique à la mémoire est géré par l'arbitre de bus (il constitue l'outil de synchronisation élémentaire) et des mécanismes de verrouillage implantés dans le matériel permettent de disposer d'un état global exact.

5.3.7 Solution répartie avec état approché

Il faut concevoir un algorithme complètement réparti de manière à ce que deux processeurs distincts ne risquent pas de prendre des décisions contradictoires. Dans la

solution répartie, les processeurs qui maintiennent un état approché du systéme, ne disposent plus de la même information. Par conséquent, pour un processeur donné, il n'est plus possible de prévoir quelles seront les décisions prises par les autres processeurs dans le même cycle. Pour ne pas prendre de décisions contradictoires il faudrait que le sous-ensemble de processeurs susceptibles d'entrer en conflit se synchronisent localement pour vérifier l'absence de contradiction. Cette solution n'est pas simple car il faut pouvoir déterminer ce sous-ensemble et, en admettant que cela soit possible, on doit quand même utiliser des synchronisations coûteuses pour harmoniser le sous-ensemble de décisions. Une autre solution est de laisser se produire les conflits et d'utiliser des protocoles avec gestion des erreurs. Ces protocoles sont beaucoup plus complexes a mettre en oeuvre que ceux utilisés dans le cas du maintien du même état (exact) sur tout les sites. De plus, en pratique, la mise au point d'un algorithme distribué est très délicate. Pour ces raisons, nous n'avons pas retenu l'approche du contrôle complètement distribué pour notre premier prototype mais cette solution reste à étudier plus précisement après mise au point du prototype et validation du modèle OPERA.

5.3.8 Contrôle Hiérarchisé

La solution la plus simple du point de vue algorithme est une architecture à contrôle centralisé sur un processeur spécialisé. En effet, il suffit que chaque unité de travail transmette ses transitions d'état à un seul processus de contrôle qui les traite séquentiellement. L'avantage de la solution de sérialisation est la simplification de l'accès à l'état global et la simplification de l'algorithme de décision : on ne prend qu'une seule décision d'appariement à la fois (même si plusieurs transferts de tâche peuvent coexister simultanément). Ceci élimine le problème des décisions contradictoires. Son inconvénient est le goulot d'étranglement potentiel que constitue l'unique processus de contrôle : l'ensemble des autres processus (ceux réalisant les unités de travail) communiquent avec lui ; son temps de service augmente avec le nombre de processeurs à gérer.

Une architecture hiérarchisée réduit l'importance de ce problème. Un premier niveau de processus se partage le contrôle des unités de travail. Les processus d'un niveau donné pilotent des groupes du niveau inférieur et le dernier niveau est le processus maître. Dans cette solution, le problème de la gestion des décisions conflictuelles entre deux processus de même niveau hiérarchique n'existe pas car les ensembles d'unités de travail qu'ils gèrent sont disjoints. Les échanges entre ces grappes de processeurs sont toutefois possibles ; ils sont gérés de manière centralisée par le supérieur hiérarchique des deux processus de contrôle des grappes. L'inconvénient est ici la conception de la "commande hiérarchisée" c'est à dire les interactions entre niveaux. Par contre, l'inertie du contrôle, due à la latence de communication entre niveaux, peut être un facteur de stabilité de la commande que l'on peut chercher à exploiter. Ce type d'architecture de contrôle est courante ; on peut citer en exemple l'architecture de la machine Multi-PSI2 de l'ICOT [Furu90].

Les différents niveaux de contrôle s'échangent entre eux de nombreux messages de petite taille (quelques octets) dans un format connu alors que les unités de travail s'échangent des données qui peuvent être de taille variable (taille qui peut être

importante dans le cas d'OPERA).

On distingue donc deux réseaux de communication :

1. le réseau de communication de données qui relie les unités de travail entre elles. Ce réseau peut éventuellement être reconfiguré si le coût de reconfiguration est faible et si la taille des données est importante. C'est le cas d'OPERA : ce réseau véhicule un petit nombre de grand messages.

2. le réseau de contrôle par lequel circulent les informations de charge et les messages de contrôle. Si la partie contrôle est responsable de la configuration du réseau de données, il est impératif que chaque unité de travail puisse disposer en permanence d'un accès à la partie contrôle. Un réseau de contrôle correspondant au contrôle hiérarchique peut être un arbre. Ce réseau véhicule un grand nombre de petits messages.

5.4 Application au cas particulier de Prolog OU parallèle

Dans le cas de Prolog parallèle, la partie contrôle doit principalement assurer la fonction d'ordonnancement des tâches (parties d'arbre) à évaluer. Elle doit également assurer le maintien d'une structure permettant de reconstruire "l'ordre séquentiel" pour la gestion des prédicats à effet de bord.

L'ordonnanceur doit être conçu de telle manière que les unités de travail soient actives la plus grande partie du temps tout en limitant le surcoût dû au parallélisme. La première contrainte implique de générer suffisamment de parallélisme tandis la seconde implique que la granularité de chaque nouvelle tâche parallèle reste supérieure au surcoût inhérent à sa création.

L'ordonnancement en Prolog est plus difficile à effectuer sur des architectures sans mémoire partagée que sur des machines à mémoire partagée pour deux raisons :

- d'une part, le coût plus élevé de la création de tâche,

- et d'autre part, l'absence de structure de données globale accessible simplement : les données globales doivent nécessairement être soit distribuées soit centralisées et, dans les deux cas l'accès se fait par envoi de messages dont le coût est très supérieur à celui d'un simple accès à la mémoire.

Le coût de création d'une tâche Prolog exige un grain important de la tâche; le manque de structure globale, c'est à dire le manque d'une partie de mémoire commune, oblige l'ordonnanceur à maintenir un état approché du système pour éviter de coûteux algorithmes de synchronisation globale dans cet environnement distribué.

La difficulté de la réalisation de l'ordonnancement des tâches augmente avec le nombre de processeurs de la machine parallèle. De ce fait, dans le modèle d'implantation

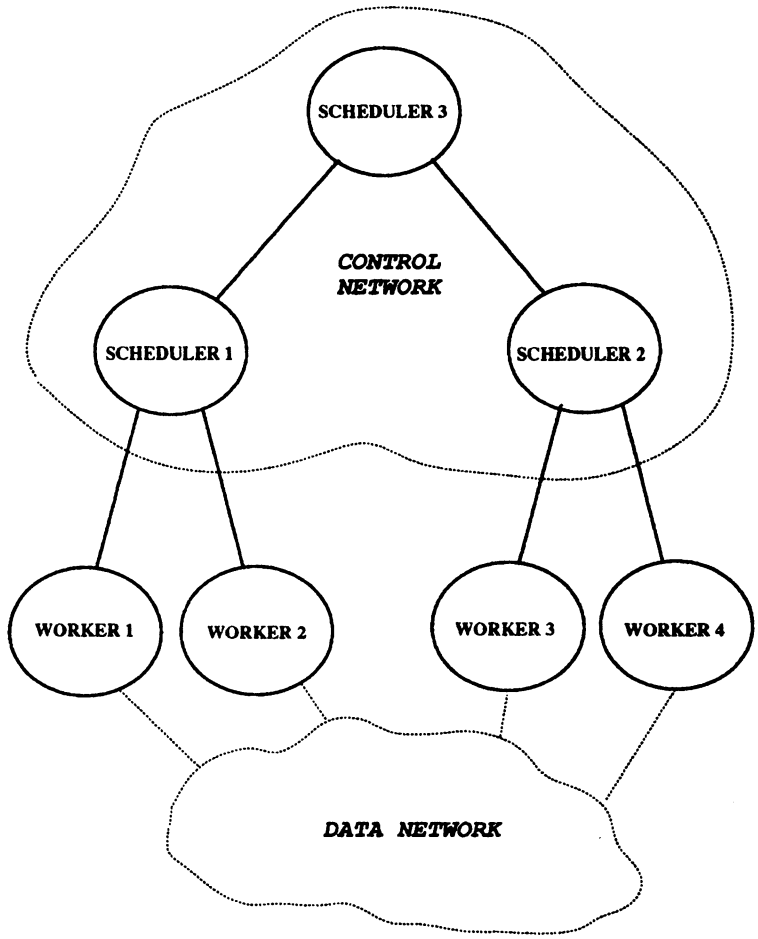


Figure 5.4 : contrôle hiérarchique

d'OPERA, une part importante des ressources de calcul est dédiée à la fonction d'ordonnancement. C'est l'une des originalités de notre système par rapport aux systèmes multiséquentiels à mémoire commune.

La section 5.4.1 présent quelques ordonnanceurs utilisés dans les systèmes Prolog parallèle (il s'agit de ceux qui ont été développé pour AURORA par le groupe Gigalips) tandis que la section 5.4.2 est consacrée à OPERA.

5.4.1 Quelques cas d'ordonnancement en Prolog

L'ordonnancement est un domaine où les recherches sont très actives, tout particulièrement dans le projet AURORA pour lequel plusieurs ordonnanceurs ont été testés. Les principaux sont l'ordonnanceur d'Argonne, celui de Manchester et le "wavefront scheduler".

Ces ordonnanceurs ont été développés pour AURORA donc pour des machines à mémoire commune. Cette propriété est la base de leur conception : tous les processeurs ont accès à l'ensemble de l'arbre d'évaluation. On dispose donc d'un état global exact pour prendre une décision d'ordonnancement. Il est cependant nécessaire de synchroniser les processus pour que les lectures et les modifications de cet état global restent cohérentes.

Les trois ordonnanceurs d'AURORA cités dans cette section ont en commun la stratégie de sélection d'un travail sur une branche donnée (règle de sélection locale) : sur une branche en cours d'évaluation on choisit toujours le travail le plus proche de la racine. Le choix de la branche (règle de sélection globale) est à l'origine de la variation de ces ordonnanceurs.

Cette heuristique de sélection locale dite "du plus haut d'abord", permet d'accéder d'abord au point de choix qui donne les tâches de forte granularité. Plus l'on se rapproche des feuilles, plus la granularité est faible, plus le risque de ralentissement par parallélisation est grand (cf section 5.2).

Les alternatives d'une même procédure sont évaluées dans l'ordre fixé par la stratégie séquentielle ce qui permet de conserver la sémantique standard (déterministe) pour l'ordonnancement des "effets de bord" créés directement ou indirectement par les clauses d'une même procédure. Les concepteurs du modèle à vecteur de versions [Haus87] ont proposé de ne pas fixer l'ordre d'évaluation des alternatives d'un même point de choix. Cela offre un degré de liberté supplémentaire à l'ordonnanceur. Cependant, il n'est plus possible de compiler complètement l'enchaînement des clauses car il n'est plus statique : on ne sait plus, avant exécution, quel sera la clause suivante à être exécutée. Il est nécessaire de maintenir un chaînage dynamique et un compteur supplémentaire sauvegardé dans chaque point de choix qui dénombre le nombre d'alternatives non encore évaluées de ce point de choix.

Dans le modèle AURORA, une distinction est faite entre les nœuds publics et les nœuds privés. Les nœuds publics sont ceux qui sont proches de la racine de l'arbre ; les nœuds privés sont proches des feuilles courantes de l'arbre. Ces derniers ne sont accédés que par un seul processeur et ont de ce fait, une structure plus simple que les premiers.

Ordonnanceur de Manchester : Le principe de l'ordonnanceur de Manchester [Cald89] est de donner du travail aux processeurs le plus tôt possible. L'ordonnancement (recherche de travail et installation de la tâche) est effectué par le processeur inactif car il n'a alors rien d'autre à faire. Pour cela, lors de la recherche d'un travail, on sélectionne le point de choix le plus proche de la position où le processeur est devenu oisif. Cette heuristique vise surtout à diminuer le temps de recherche. Le critère de proximité est également celui retenu dans le modèle MUSE [Ali90] car il permet de diminuer le coût d'installation d'une tâche.

Une optimisation de l'ordonnancement consiste à modifier la structure de l'arbre de manière à éliminer les point de choix morts : ces derniers sont ceux qui n'ont plus d'alternatives en attente d'évaluation. Cette élimination accélère le parcours de l'arbre lors de la recherche de travail, au prix d'une traitement supplémentaire lors de l'exécution de la dernière alternative de chaque point de choix. L'amélioration n'est pas évidente dans le cas où ce traitement supplémentaire à été fait pour des points de choix qui ne sont pas accédés lors de la recherche de travail.

Les données utilisées par l'ordonnanceur de Manchester sont en partie distribuées dans chaque nœud de l'arbre et en partie rassemblées dans des tableaux globaux. Les données globales permettent d'évaluer le coût de migration (le nombre d'entrée de la pile trainée est significatif du travail à réaliser pour installer le tableau de liaison de la tâche créée). La structure de donnée locale à un nœud consiste en un vecteur binaire : à chaque processeur correspond un bit de cette table. Cet indicateur est positionné pour tous les processeurs en train d'évaluer une partie issue de ce nœud. Cela permet de déterminer l'ensemble des processeurs travaillant dans le voisinage. La sélection d'un élément de cet ensemble est effectuée en utilisant les données globales avec pour critère la minimisation du coût de migration. Si aucun bit n'est positionné dans le nœud courant on remonte plus loin dans l'arbre élargissant ainsi le champ de recherche des voisins

L'inconvénient principal de cette méthode réside dans le fait que la gestion de l'ordonnancement est globale ce qui implique des synchronisations entre les processeurs. Ceci peut devenir un goulot d'étranglement dans le cas d'un grand nombre de processeurs.

Ordonnanceur d'Argonne Dans l'ordonnanceur d'Argonne [Butl88] le choix est de minimiser le nombre de structures globales et de répartir les décisions d'ordonnancement. Le principe est le suivant :

On cherche dans le nœud courant s'il n'y a pas d'autre travail : une table de bits associée à chaque nœud indique s'il y a du travail disponible dans les arbres issus des alternatives de ce point de choix. Si aucun des bits n'est positionné, on itère la recherche dans le nœud père. Le processeur cherchant du travail se déplace ainsi dans l'arbre en suivant ces indicateurs.

L'intérêt de cette solution est la localité de la décision qui diminue le risque de conflit d'accès en écriture (les consultations des vecteurs binaires peuvent être effectuées simultanément par plusieurs processeurs).

Ordonnanceur "wavefront" Le front de la vague ("wavefront scheduler") [Bran88] a pour principe de faciliter la recherche de travail en maintenant un chaînage entre les nœuds qui sont à la limite de la partie publique et de la partie privée. Compte tenu de la stratégie de sélection du point de choix le plus ancien sur une branche donnée, cet ensemble de nœuds représente la zone en cours d'évaluation ("wavefront") donc les sources potentielles de travail. Le "wavefront scheduler" maintien également une structure de données mémorisant les relations entre deux nœuds voisins dans cette liste : il s'agit du point de choix le plus récent commun aux deux branches. Cette information permet de ne modifier que le strict nécessaire dans le tableau de liaison : le processeur oisif effectue un retour arrière jusqu'à ce point commun puis descend sur la branche sélectionnée en installant le reste du tableau de liaison.

Les mesures de gain de performances d'AURORA sur un Sequent Symetry à 20 processeurs [Carl90] montrent que les différents ordonnanceurs cités permettent d'obtenir des résultats sensiblement équivalents : en général de l'ordre d'un point d'écart pour 20 processeurs. Ceci remet en question l'emploi de techniques d'ordonnancement sophistiquées et complexes à mettre en œuvre pour des machines comportant un petit nombre de processeurs.

5.4.2 régulation de charge dans le modèle OPERA

Le problème de la stratégie d'exécution parallèle ou règle de coopération entre unités de travail est de garantir un accroissement de performance par parallélisation. Il se pose de façon cruciale dans OPERA du fait de

- l'absence de connaissance a priori du potentiel de travail parallèle exploitable,
- l'inexistence d'un état global d'exécution,
- des coûts d'installation du parallélisme élevés et fonction du volume de données échangées.

Les performances des systèmes multiséquentiels dépendent énormémemt du couple : stratégie de régulation de charge, efficacité des transferts. On doit comparer le coût de transfert avec le gain que l'on estime obtenir grâce à l'utilisation du parallèlisme. La règle de sélection locale vise à garantir l'accroissement de parallélisme en respectant les conditions C_{min} : $(T_e > E)$ et C_{max} : $(T_l > I)$. La vérification de ces conditions pose deux problèmes distincts :

1. estimer et minimiser les termes E et I.
2. évaluer la charge du processeur surchargé (et de la partie exportée).

La règle de sélection globale vise à équilibrer la charge entre les processeurs. Elle s'appuie sur les informations de charge fournies par les unités de travail.

a) Notion de tâche

Une tâche correspond à un parcours d'une branche de l'arbre de recherche depuis un noeud initial jusqu'à un noeud terminal. Nous avons choisi le point de choix comme unité élémentaire d'exportation. La pile choix décrit les tâches en attente (clauses d'un point de choix). Les volumes de données à transférer (piles) sont donc connus à l'exécution. Une TWAM produit des tâches par paquet (clauses d'un point de choix). Le coût d'exportation des alternatives d'un même point de choix est identique car les zones à transférer sont les même c'est pourquoi nous avons choisi d'exporter en une seule fois toutes les alternatives d'un même point de choix qui n'ont pas encore été évaluées. Remarquons que cela ne signifie pas que toutes les alternatives d'un point de choix seront évaluées par le même processeur car un même point de choix peut être exporté plusieurs fois.

b) Estimation et minimisation des temps de communication

Le temps d'importation I et, dans une moindre mesure, celui d'exportation E (retard par vol de cycles) dépendent du volume de données à transférer (les piles). Ils s'expriment comme la somme d'une constante (surcoût du processeur pour l'amorçage du transfert) et d'une fonction croissante du volume. Cette fonction est un paramètre architectural de la machine.

Il est intéressant de minimiser le coût des transferts et par conséquent, les coûts d'importation et d'exportation. La taille décroissante des piles avec l'âge d'un point de choix donne une règle simple :

- n'exporter que du travail appartenant aux points de choix les plus vieux d'une TWAM,
- sélectionner une TWAM exportatrice qui minimise les coûts de transfert.

Cette règle joue favorablement sur les deux inéquations en diminuant le grain de parallélisme exportable ainsi que la charge de travail à conserver.

Un point de choix est un ensemble de tâches (clauses restant à exécuter) partageant le même environnement initial. Il s'ensuit que l'on peut exporter une partie des tâches d'un même point de choix pour un coût d'importation/exportation identique à celui d'une seule tâche. L'exportation d'un groupe de tâches joue favorablement sur la première inéquation (C_{min}) puisque le temps de travail est la somme des durées d'exécution des tâches exportées. Par contre, l'effet est défavorable sur la seconde inéquation C_{max} car le temps de travail résiduel après exportation est réduit en proportion du nombre de tâches exportées. Le nombre de tâches à exporter est donc un compromis à établir dans le cadre de la contrainte qui garantit l'efficacité du parallélisme.

c) Problème de l'obtention de la charge

La notion de charge du *Calculateur* est un problème important : dans le cas général le temps d'évaluation d'un but est imprévisible. Le calculer exactement revient à évaluer

complètement le programme. Une solution à ce problème serait de confier à l'utilisateur le choix d'un système de pondération de chaque branche d'évaluation mais cette façon de définir la charge serait lourde pour le programmeur et supposerait une modification du langage.

La vérification des contraintes C_{min} et C_{max} est la seule manière de garantir un accroissement des performances chaque fois que cela est possible. Cette vérification implique que l'on puisse estimer d'une manière raisonnablement précise le temps de traitement d'un point de choix ou d'un groupe de points de choix. deux approches sont possible :

Determination dynamique Ce domaine de recherche étant encore très récent, aucun résultat n'a pour l'instant été trouvé. L'estimation du temps de traitement d'un point de choix à l'exécution ne peut reposer que sur la l'existence d'un paramètre de la machine Prolog qui reste stable au cours du temps. De plus cette mesure doit être peu coûteuse de manière à ne pas ralentir exagérément l'exécution séquentielle de la machine Prolog.

Determination statique Les recherches sur l'estimation à la compilation du poids des prédicats [Herm90] [Tick88] n'ont pour l'instant permis que de les ranger suivant deux classes : parallélisable, non parallélisable. Ces méthodes permettent de donner un poids au prédicats (une charge approximative). Elles montrent leurs limites pour des programmes utilisant la récursivité croisée. Plus généralement, tout programme dont la durée d'exécution dépend du jeu de données initial et qui comporte des structures de type itératives ne peut être pondéré précisement. (Le problème de fond réside dans l'indécidabilité du problème de l'arrêt). Ces méthodes sont très lourdes à mettre en œuvre.

d) Solution retenue pour OPERA

Compte-tenu des limitations dues à la complexité intrinsèque du problème, nous avons choisi de définir la charge à partir de paramètres évaluables dynamiquement (au cours de l'exécution du programme). Nous présentons ici quelques points significatifs des différentes alternatives. Le choix retenu en définitive a pour intérêt principal de réduire au minimum le temps nécessaire à l'évaluation de la charge de l'unité de travail.

Vis à vis de son environnement la TWAM se comporte comme un producteur et un consommateur de points de choix. Le temps total d'évaluation d'un programme est une fonction strictement croissante du nombre de points de choix crées lors de son exécution. Notons que cette fonction n'est pas connue exactement mais qu'elle est toujours croissante. L'état de charge visible d'une unité de travail peut donc être représenté par l'ensemble des points de choix existant à un instant donné.

Décharger une unité de travail consiste à prendre un sous-ensemble de ses points de choix et à les transmettre à l'environnement pour évaluation. Symétriquement, une unité de travail ayant une charge nulle demande à son environnement un ensemble de

points de choix à évaluer. Nous préciserons plus-tard la stratégie globale employée pour réguler la charge entre plusieurs unités de travail.

La détermination de ce sous-ensemble de points de choix à exporter est très importante: plusieurs points sont à considérer:

- Un point de choix n'est exportable que si son évaluation n'est pas conditionnée par un "cut". On évite ainsi d'évaluer des points de choix susceptibles d'être détruits avant leur évaluation. Il est nécessaire d'inhiber le parallélisme lorsqu'on crée un point de choix pour une clause contenant un ou plusieurs "cut" (un simple compteur de "cut" décrémenté signale l'opportunité de l'exportation lorsqu'il devient nul).

- On doit éviter d'exporter un point de choix en cours d'évaluation par l'unité de calcul (*Calculateur*) du processeur surchargé.

Examinons maintenant la structure de l'ensemble des points de choix et les critères de division de cet ensemble pour en extraire une partie à exporter. La stratégie d'évaluation utilisée par le *Calculateur* étant "en profondeur d'abord", ce processus accède à l'ensemble des choix comme à une pile. Le point de choix le plus ancien (fond de pile) est donc celui qui est le plus éloigné du front de calcul (sommet de pile) du *Calculateur*. Cet écart d'adresses correspond à un éloignement dans le temps ce maximise T_l. En ce qui concerne les transferts, la taille mémoire (taille cumulée de l'ensemble des fonds des piles) nécessaire à l'évaluation de ce point de choix est plus petite que celle nécessaire à tout autre point de choix local. De plus, le point de choix le plus ancien n'a pas d'ancêtre donc pas de trainée à transmettre. la stratégie de choix du "plus ancien" tend à vérifier la condition C_{max}

Pour tendre vers la condition C_{min} on peut transférer plusieurs points de choix au cours du même appariement. Pour respecter la stratégie décrite ci-avant on se limite à transférer les plus anciens. Cette factorisation des transferts a pour avantage d'augmenter T_e. Elle permet également de diminuer (factoriser) les coûts d'amorçage de transfert. Par contre, la taille des données transférées est plus importante car il faut transférer les portions de piles qui correspondent au point de choix exporté le plus récent (la portion de pile traînée correspondante doit également être transférée). L'augmentation de T_e va donc de pair avec l'augmentation de E et de I.

Le nombre de points de choix à exporter est un paramètre du système qu'il faut pouvoir adapter en fonction du programme. Pour simplifier le prototype, nous avons choisi de fixer ce seuil à un seul point choix exporté.

e) Régulation de charge

Jusqu'alors nous avons considéré l'environnement d'une unité de travail comme un tout. En fait, l'environnement est constitué de plusieurs unités de travail (*Travailleur*) identiques qui participent à une résolution plus rapide d'un même problème. Chacune des N unités de travail possède une charge locale. L'ensemble des N charges locales constitue la charge globale du système. La régulation de charge du système consiste

donc à répartir dynamiquement les points de choix entres les unités de travail de façon à maintenir équilibrées les N charges locales.

La charge à l'instant t d'une unité de travail peut être représentée par le nombre de points de choix produits par cette unité de travail qui n'ont pas encore été explorés à l'instant t. Selon le nombre de points de choix en attente, nous répartissons les unités en trois classes (ou macro-états):

unité oisive: c'est le cas lorsqu'une unité de travail n'a aucun point de choix dans sa pile ET que le *Calculateur* est inactif. Conventionnellement l'état de charge d'un tel processeur est négative (-1). Une telle unité est candidate à l'importation d'un point de choix en provenance d'une unité surchargée (s'il en existe une) et se met en attente de la partie contrôle qui lui fournira une voie de communication vers une unité à décharger[2].

unité isolée: c'est une unité dont le nombre de point de choix est inférieure au seuil de surcharge S et qui n'est pas oisive. C'est le cas lorsqu'une unité de travail n'a aucun point de choix exportable dans sa pile (charge = 0) mais que son *Calculateur* est actif (c'est par exemple, le cas d'un programme déterministe), ou bien lorsque cette unité de travail n'a qu'un seul point de choix (charge = 1). Pour ce dernier cas on considère qu'il est inutile d'exporter un point de choix qui peut faire défaut tout de suite après (condition C_{max}). La valeur minimale de S est donc 2. Pour les programmes de testés avec notre prototype nous avons fixé (empiriquement[3]) la valeur $S = 10$. En pratique, il est intéressant d'augmenter le seuil S du nombre de points de choix à partir duquel l'unité de travail est considérée comme surchargée car cela permet de conserver suffisamment de travail localement avant de consacrer du temps à le partager avec une autre unité de travail. Ce seuil a pour rôle principal de garantir une distance minimale entre le front de calcul et la partie exportable. Une telle unité est dite isolée car elle travaille de manière autonome et ne perturbe en rien le reste du système (pas de communication).

unité en surcharge: toute unité ayant au moins S points de choix est dite en surcharge, c'est-à-dire qu'elle a plus de points de choix quelle ne peut en consommer à un instant donné donc elle est candidate à l'exportation de points de choix vers une unité inactive (s'il y en a). Une unité surchargée signale sa charge à la partie contrôle qui lui fournira éventuellement une voie de communication vers une unité oisive à charger.

L'état de chaque unité de travail est échantillonné périodiquement par un processus *espion* placé sur chaque unité de travail et fonctionnant de manière asynchrone par rapport au moteur d'évaluation séquentiel. L'échantillon est transmis à l'ordonnanceur via le réseau de contrôle. La partie contrôle maintien l'état local de chaque unité de travail selon le graphe d'état représenté sur la figure 5.5:

On remarquera l'ajout d'un état de transfert qui permet d'isoler les unités de travail

[2]Le macro-état "oisif" (qui se réduit à un état unique) est l'état initial de toutes les unités de travail à l'exception d'une seule sur laquelle on place le problème à résoudre.

[3]après une série de mesures

en cours d'exportation ou d'importation. Cette isolation permet de ne pas les prendre en compte dans la règle de sélection globale alors que leurs unités d'échanges sont déjà allouées pour des transferts.

Notons également que, parmi les 3 autres macro-états seul l'état oisif est certain. En effet, la technique d'échantillonnage et la latence des communications font que l'état dans lequel la partie contrôle voit les unités de travail peut être différent de l'état dans lequel elles se trouvent réellement au moment où l'on applique la règle de sélection globale. Dans ce cas des décisions inopportunes peuvent être prises. Le protocole de discussion entre unités appariées doit donc être capable de traiter cette éventualité. L'augmentation de la fréquence d'échantillonnage permet d'éviter une trop grande dérive de l'état global vu par la partie contrôle par rapport à l'état global réel du système ce qui diminue le nombre des tentatives de transfert infructueuses. cependant cette augmentation de fréquence a pour inconvénient de surcharger du processeur de l'unité de travail. Les unités de travail en cours d'échange transmettent leur nouvel état local exact dès la fin du transfert des piles (ou de la marque signalant l'absence de travail exportable). Ceci permet également de limiter la dérive de l'image approchée.

L'ordonnanceur maintient un tableau donnant le macro état courant de chacun des processeurs qu'il pilote.

L'algorithme de contrôle exécuté par l'ordonnaceur est le suivant :

```
Soit I l'ensemble des unites oisives
Soit R l'ensemble des unites isolees en cours d'evaluation
Soit S l'ensemble des unites surchargees
Soit T l'ensemble des unites en cours de transfert

initialement :
  R = le processeur racine (celui qui commence l'evaluation)
  I = les N processeurs sauf le processeur racine
  S = {}
  T = {}

tant que cardinal(I) < N
debut
  attendre une requete d'un de ses dependant hierarchique p
  enlever p de l'ensemble auquel il appartenait
  etat[p] := etat recu
  selon etat[p]
    cas etat[p] = oisif
      debut
        si S <> {} alors
          debut
            S = S - {s} ou s est le processeur le plus charge
            apparier( s , p ) ;
```

```
          fin
        /* sinon le cas surcharge declenchera le transfert */
      fin
    cas etat[p] = surcharge
      debut
        si I <> {} alors
          debut
            I = I - {i}
            apparier( p , i );
          fin
        /* sinon le cas souscharge declenchera le transfert */
      fin
fin

apparier( s , i )
debut
   T = T + {s,i}
   etablir une route (logique) entre s et I
   signaler a s son port d'exportation
   signaler a i son port d'importation
fin
```

La régulation de charge est effectuée au cours de l'exécution du programme et peut se dérouler en parallèle avec l'évaluation multi-séquentielle de ce programme, il est donc très important que cette régulation soit rapide donc simple. Le tri des unités de travail par charge décroissante permet d'accélérer la sélection d'une unité surchargée lors de la réception d'un message "unité oisive". Si toutes les unités de travail ont un coût d'accès uniforme les unités oisives sont considérés comme équivalentes. En cas de copie incrémentale, on peut par exemple chercher celle qui est le plus près dans l'arbre de l'unité surchargée (c'est celle qui a la plus grande partie commune avec l'unité surchargée).

hiérarchisation Un point important dans la régulation de charge est qu'elle peut-être hierarchisée lorsque l'on groupe des unités de travail en grappe. Cela est particulièrement intéressant lorsque la structure en grappe est justifiée par des coûts de communication entre grappes supérieurs aux coûts de communication entre deux nœuds d'une même grappe (c'est le cas des machines Supernode [Munt88] qui comportent plus de 32 processeurs de travail).

A chaque niveau de la hiérarchie, la fonction de répartion des points de choix est calculée à partir de l'ensemble des charges des sous-systèmes. Au niveau le plus bas le sous-système est une unité de travail. Cette solution permet de centraliser sur un même processeur (spécialisé) la fonction de régulation de charge des sous-système hiérarchiquement dépendants. La centralisation de la charge permet d'effectuer très rapidement

un choix pour la répartition des points de choix, cependant, si le nombre de processeurs est élevé, le serveur qui exécute l'algorithme de répartition et le mécanisme de collecte des charges locales peuvent devenir un goulot d'étranglement. La hiérarchisation de la partie contrôle a pour but de limiter cet effet. Elle permet également d'introduire une certaine latence dans le "temps de réaction" du système, ce qui peut être bénéfique dans les cas limites : les variations de charges de faibles amplitudes disparaissent lorsque l'on remonte dans la hiérarchie.

La structure hiérarchique de la partie contrôle reflèle l'architecture du réseau de contrôle des machines de la gamme Supernode utilisées pour implémenter notre prototype : il s'agit d'un ensemble de processeurs de contrôle interconnectés par une hiérarchie de bus

L'implantation effective du contrôle hiérarchique sur le Méganode n'a pas encore été réalisée mais les principes de base de la régulation de charge ont été validés sur une machine de bas de gamme (Tnode) qui correspond à une grappe élémentaire des machines Meganode.

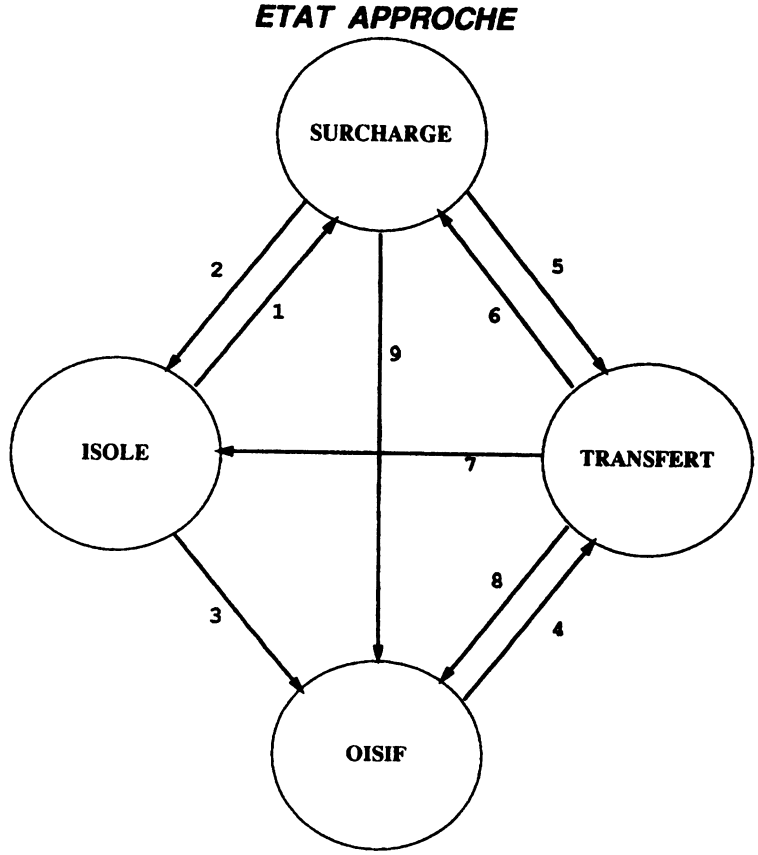


Figure 5.5 : graphe d'état d'une unité de travail

Ce graphe de transition d'état (approché) permet à la partie contrôle de prendre des décisions de régulation de charge. Toutes les transitions sont effectuées lors de la réception de messages en provenance des unités de travail. Les transitions 1 et 2 sont provoquées par l'échantillonnage de l'état d'une unité de travail par son processus *espion*. 3 et 9 sont émises par le processus *Calcul* lorsque celui-ci n'a plus rien à évaluer. Notons que la transition 9 traduit le fait que l'état maintenu par la partie contrôle n'est qu'une image approchée de l'état exact de l'unité de travail : la transition exacte 2 peut ne pas être détectée par le processus espion qui n'échantillonne que de temps à autre. La conjonction des transitions 4 et 5 en provenance de deux unités de travail provoque un transfert de données entre ces unités. Les transitions 6, 7 et 8 permettent aux deux unités de transferts de signaler le nouvel état de chacune des unités de travail après la tentative de transfert. Ces transitions permettent d'éviter une dérive trop grande de l'état vu par la partie contrôle par rapport à l'état exact de la partie opérative. Les transitions 8 et 9 sont des conséquences exemplaires de cette dérive et des protocoles "avec erreur" qui en résultent. En particulier, 8 traduit un échec du transfert : le processeur qui était apparemment surchargé n'a plus de travail disponible au moment où on lui demande d'en exporter.

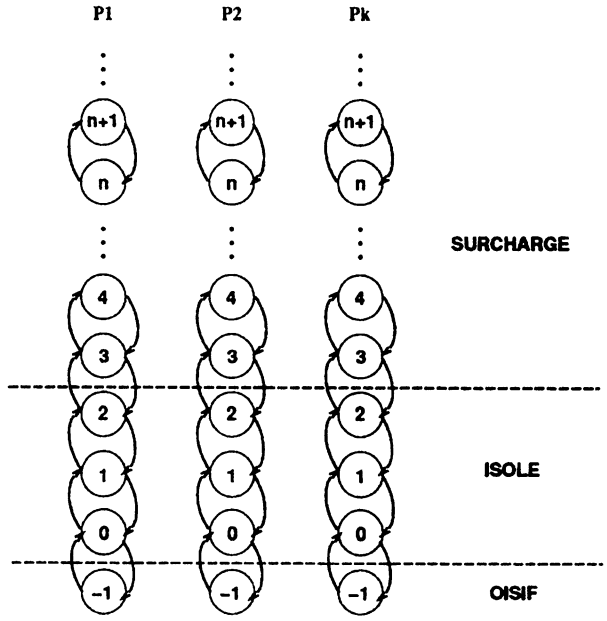


Figure 5.6 : Macro-états de charge

Les transitions d'états des unités de travail ne sont pas toutes significatives. Une partition de l'ensemble de ces valeurs d'états locaux est obtenue en définissant deux seuils : le nombre de points de choix en dessous duquel une unité de travail est considérée comme sous-chargée, et le nombre de points de choix en dessus duquel une unité de travail est considéré comme surchargée. Ces seuils sont des paramètres du système.

Chapitre 6

REALISATION D'UN PROTOTYPE OPERA - RESULTATS

6.1 Description de l'architecture cible: le Supernode

Pour avoir une idée plus précise de l'opportunité réelle du parallélisme dans l'évaluation de Prolog sur des machines multi-processeurs à mémoire distribuée, nous avons choisi pour notre prototype:

- Un processeur bien adapté au parallélisme: le Transputer [1]. Ce processeur possède une gestion de processus très efficace et permet de créer facilement des réseaux de processeurs de taille importante. La réalisation matérielle des primitives de communication (rendez-vous synchrone et échange de messages) est un autre atout de ce processeur.

- Une machine facilement extensible: le Supernode. Les Transputers y constituent des unités de travail faiblement couplées (chacun a sa propre mémoire locale). Une caractéristique essentielle de cette machine est la possibilité de reconfigurer dynamiquement le réseau d'interconnexion des processeurs et de permettre ainsi de réduire le coût des transferts entre unités de travail. Cette reconfiguration est assurée par une structure de contrôle global constituée par un réseau de Transputer.

[1] Au tout début du projet Opéra nous avions pensé utiliser une machine multiprocesseur à base de 68000 connectés sur un bus VME. J'avais alors écrit un premier compilateur Prolog générant du code 68000 mais une évaluation de cette maquette nous à fait nous orienter vers les machines à base de Transputer.

Les choix d'implantation du modèle OPERA ont été faits de manière à prendre en compte l'architecture des configurations Tnode et Meganode de la gamme Supernode [Harp86].

6.1.1 Le Transputer

Le nom de Transputer désigne en fait une famille de micro-processeurs de la société INMOS. Il s'agit de micro-processeurs 32 bits (T400,T422,T414, T425) ou 16 bits (T212) intégrant soit des unités de calcul en virgule flottante (T800, T801 ou T805) soit des contrôleurs de disques (M212), et d'une famille de composants d'interconnexion tel que des commutateur de circuits "crossbar 32x32" (C004) ou des interfaces lien série/bus parallèle (C011,C012). Cette grande variété illustre le fait que le Transputer est issu d'une philosophie de conception modulaire d'architecture parallèle. Nous ne parlerons ici que des caractéristiques communes aux micro-processeurs T414 et T800 car les autres composants utilisés dans l'architecture cible de notre prototype sont de moindre importance.

Le Transputer T800 est un microprocesseur 32 bits pouvant fonctionner avec une horloge à 15, 20 ou même 30 Méga-Hertz. Sur un seul circuit sont regroupés une unité arithmétique et logique 32 bits, une unité flottante 64 bits aux normes IEEE754, 4 processeurs d'entrées/sorties, 4 Kilo-octets de RAM [2] permettant des accès en un seul cycle, et une interface de gestion de mémoire externe qui permet d'utiliser une grande variété de types de mémoire.

Sur le T800, l'interface mémoire est réalisée de manière classique par un bus externe qui multiplexe adresse et données sur 32 bits. Un T800 peut adresser jusqu'à 4 Giga-octets de mémoire externe. Avec une horloge à 20 MégaHertz on peut obtenir un taux de transfert de données de 26 Méga-octets par seconde. Le Transputer n'a pas été conçu pour partager de la mémoire entre plusieurs processeurs, les processeurs sont donc faiblement couplés.

Sur les T800/T414, la communication entre plusieurs modules Transputer + mémoire locale est rendue possible par la présence de 4 liens série permettant une communication bidirectionnelle. La liaison entre deux Transputers s'effectue sans logique extérieure, ce qui facilite beaucoup la conception de machine à réseaux de Transputers [3]. La vitesse de transmission théorique est de 10 Méga-bits par seconde et par lien, ce qui correspond à un débit effectif d'environ 970 Kilo-octets par seconde. Moyennant des limitations plus contraignantes (en particulier sur la distance), la vitesse de transmission des liens peut être configurée à 20 Mbits/sec. Les 4 processeurs d'entrées/sorties intégrés sur le T800 sont indépendants et peuvent émettre ou recevoir par les liens séries en accédant directement à la mémoire (DMA) ce qui permet au processeur d'exécuter un programme en parallèle avec les transferts (au maximum 8 transferts simultanés par

[2] pas d'unité en virgule flottante et RAM de 2 Ko seulement dans le cas du T414.

[3] Il existe un grand nombre de sociétés qui produisent chacunes plusieurs cartes et plusieurs configurations de machines à base de Transputer. Une machine à 1024 processeurs a même été construite en Angleterre.

T800 en utilisant les deux directions sur chacun des 4 liens). Cette capacité a été mise à profit dans OPERA pour effectuer du calcul pendant les entrées/sorties.

Du point de vue programmation, le Transputer est tout à fait original. Son jeu d'instructions a été spécialement conçu pour faciliter l'implantation du langage OCCAM. La multi-programmation est très aisée car le processeur comporte un ordonnanceur de processus cablé, une gestion du temps (échéancier) ainsi que des primitives de communication par rendez vous. Les performances du Transputer pour la commutation de processus sont impressionnantes : en dessous de la micro-seconde. Ces capacités de multi-programmation sont surtout utilisées dans notre noyau de système; notre unité de travail Prolog, quant à elle, n'est constituée que d'un nombre limité de processus.

6.1.2 Caractéristiques du Tnode

Le Supernode est une famille de machines multi-processeurs reconfigurable dont le processeur de base est le Transputer. La gamme des machines Supernode s'étend de 16 à 1024 processeurs. Ces machines ont deux caractéristiques importantes qui les différencient d'autres machines à base de Transputers : le réseau d'interconnexion est reconfigurable dynamiquement (en cours d'exécution) à un coût raisonnable. Cette reconfiguration est assurée par des Transputers spécifiques et une voie de contrôle qui permet la communication entre ces processeurs de contrôle et les autres. Cette machine est le résultat du projet ESPRIT P1085 auquel nous avons participé.

La machine utilisée pour notre prototype est un Tnode. Cette configuration constitue le début de la gamme Supernode. Elle peut comporter jusqu'à 32 processeurs T800 dans sa version de base et jusqu'à 64 dans sa version tandem. Son contrôle est simplifié car le réseau d'interconnexion est constitué d'un simple crossbar et le contrôle ne comprend qu'un processeur.

Ainsi, notre Tnode comporte 16 Transputers de travail T800 avec 2 Méga-octets de mémoire locale pour chacun d'eux, et 1 Transputer de contrôle T414 comportant 512 Kilo-octets de mémoire.

Ces processeurs peuvent communiquer entre eux par deux réseaux distincts.

- 1 réseau reconfigurable réalisé au moyens des 4 liens bidirectionnels de chaque Transputer. La configuration qui peut être modifiée dynamiquement est contrôlée par le T414.
- 1 bus de contrôle de type maître-esclave, disposant d'une bande passante de 100 Ko/s, qui permet aux 16 T800 de communiquer avec le processeur T414 de contrôlé.

a) Le réseau d'interconnexion des liens

Le réseau d'interconnexion des Supernodes est totalement reconfigurable : on peut configurer le Supernode de manière à obtenir n'importe quel graphe dont les sommets sont de degré 4. Les 4 liens "séries" des Transputers sont configurés (au démarrage) à 10

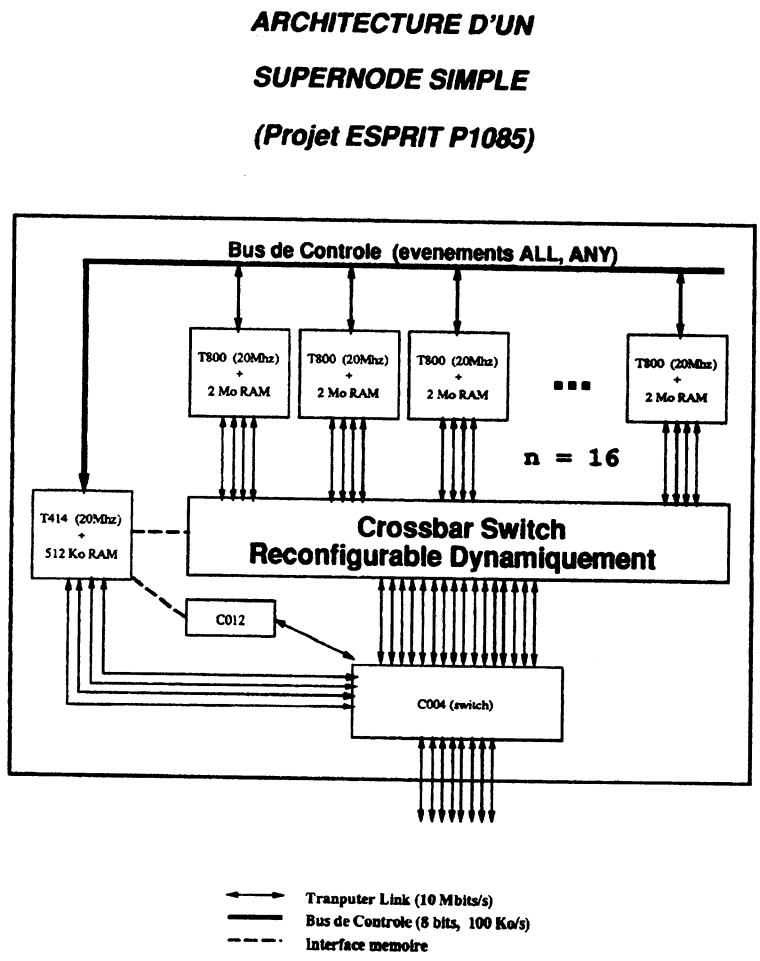


Figure 6.1 : Configuration de Tnode utilisée pour OPERA

Mbits/sec pour être compatible avec la carte d'interface de la machine hôte [4] mise à notre disposition.

b) La voie de contrôle

Le processeur contrôleur est relié à tous les processeurs de travail par l'intermédiaire d'une voie de contrôle qui permet entre autre de gérer les signaux de base du Transputer (Reset, Analyse, Erreur). Cette voie de contrôle comporte également un bus 8 bits que ses concepteurs ont destiné à la mise au point, aux synchronisations inter-processeurs et à la reconfiguration du réseau d'interconnexions. Les faibles performances de ce bus (de l'ordre de 100 Ko/s) restreignent son usage à des messages de petite taille. Cette voie de contrôle diffère d'un bus classique en ce sens qu'elle est un connecteur de N

[4] Une station de travail Apollo DN3000 équipé d'une carte Transputer T414 + 2 Mo de RAM accessible via un bus compatible PC-AT (très lent)

processus esclaves vers un maître qui permet

- au maître d'attendre un signal (et un octet) d'un esclave parmi un sous-ensemble quelconque de ses esclaves,
- au maître de diffuser un signal (et un octet) sur tout ou partie de ses esclaves.

Ces capacités sont suffisantes pour émuler sur ce bus de contrôle des liens virtuels qui rendent ainsi les communications via le bus de contrôle conforme au protocole standard de communication.

6.2 Placement de l'architecture logique sur l'architecture matérielle du Tnode

Cette section rappelle l'architecture du système OPERA et présente son placement sur l'architecture physique du Tnode. L'architecture logicielle du prototype a été conçue pour que cette projection soit "facile" c'est à dire qu'on trouve des équivalences naturelles entre processus et processeurs. Le processus est une abstraction de la notion de processeur et les interactions entre processus se font

- par l'intermédiaire du partage de mémoire dans le cas où les processus sont placés sur le même processeur et si la taille importante des données partagées (ex: piles de Prolog) rend impossible où trop coûteuse la copie d'un message;
- par des communications lorsque les processus sont placés sur des processeurs différents ou bien si la taille des messages est suffisament faible pour que l'on puisse en négliger le coût de copie.

6.2.1 Rappels sur l'architecture d'OPERA

OPERA est un système constituée d'une partie opérative et d'une partie contrôle.

a) Partie operative

La partie opérative est constituée d'un ensemble d'unités de travail faiblement couplées (interconnectées par un réseau de communication). Chaque unité de travail se décompose en une unité de traitement et une unité de transfert. L'unité de traitement est une machine virtuelle TWAM capable d'évaluer en séquentiel un programme Prolog indépendamment de son environnement. L'unité de transfert joue le rôle d'interface entre l'unité de traitement et son environnement. En particulier, elle assure le transfert des piles entre deux unités de travail. L'unité de transfert doit donc pouvoir accéder à toutes les structures de données de l'unité de traitement.

b) Partie contrôle

La partie contrôle est constituée d'une ou plusieurs unités de contrôle selon le nombre d'unité de travail à gèrer et d'un système d'échantillonnage de l'état de la partie opérative. Le rôle de la partie contrôle est d'assurer la régulation de charge. Pour ce faire, la partie contrôle maintient un état global approché de la partie opérative. Cet état est obtenu par des unités d'échantillonnage qui accèdent périodiquement à l'état local de chaque unité de travail. Ces états locaux sont transmis aux unités de contrôle via un réseau de contrôle. Au vu de l'état global approché des unités de travail dont elles ont la charge, les unités de contrôle prennent les décisions d'appariement entre unités de travail surchargées et unité de travail oisives. Ces décisions sont notifiées (via le réseau de contrôle) aux unités de travail qui alors entament un protocole d'échange de données. La partie contrôle est responsable de l'allocation des ressources globales de communication entre unités de travail. Lors de chaque appariement elle alloue une route logique entre les deux unités de travail concernées. Cette route peut être désallouée lorsque les deux unités de travail signalent à la partie contrôle la terminaison de leur protocole d'échange (cf figure 6.2).

6.2.2 Découpage en processus et placement

Dans cette section, les principaux processus de l'architecture logicielle sont décrits brièvement. La partie opérative (c.a.d. les unités de travail) est placée sur les processeurs de travail T800.

Le code du programme Prolog est dupliqué dans la mémoire locale de chaque T800. Une unité de travail est constitué du processus "*Calculateur*" qui est le moteur d'évaluation séquentiel de Prolog (exécute le programme Prolog) et de l'interface qui lui permet de dialoguer avec son environnement. L'environnement d'une unité de travail est constitué des unités de travail et de la partie contrôle (processus *Ordonnanceur*). A chaque type d'interaction correspond un processus spécialisé. Les événements qui peuvent se produire sont de trois types :

- Une unité de travail devenue inactive sollicite un nouveau travail de la part de son environnement.
- Une unité de travail surchargée est sollicitée par son environnement pour fournir du travail.
- L'échantillonnage de l'état d'une unité de travail pour la mise à jour de l'état global approché que maintient l'unité de contrôle.

L'interface est donc constituée d'une unité de transfert qui gère les échanges de portion de piles entre unités de travail, et d'une unité d'échantillonnage de l'état de l'unité de travail qui envoie des informations (charge du "*Calculateur*") à la partie contrôle.

Les processus du prototype d'OPERA sont les suivants :

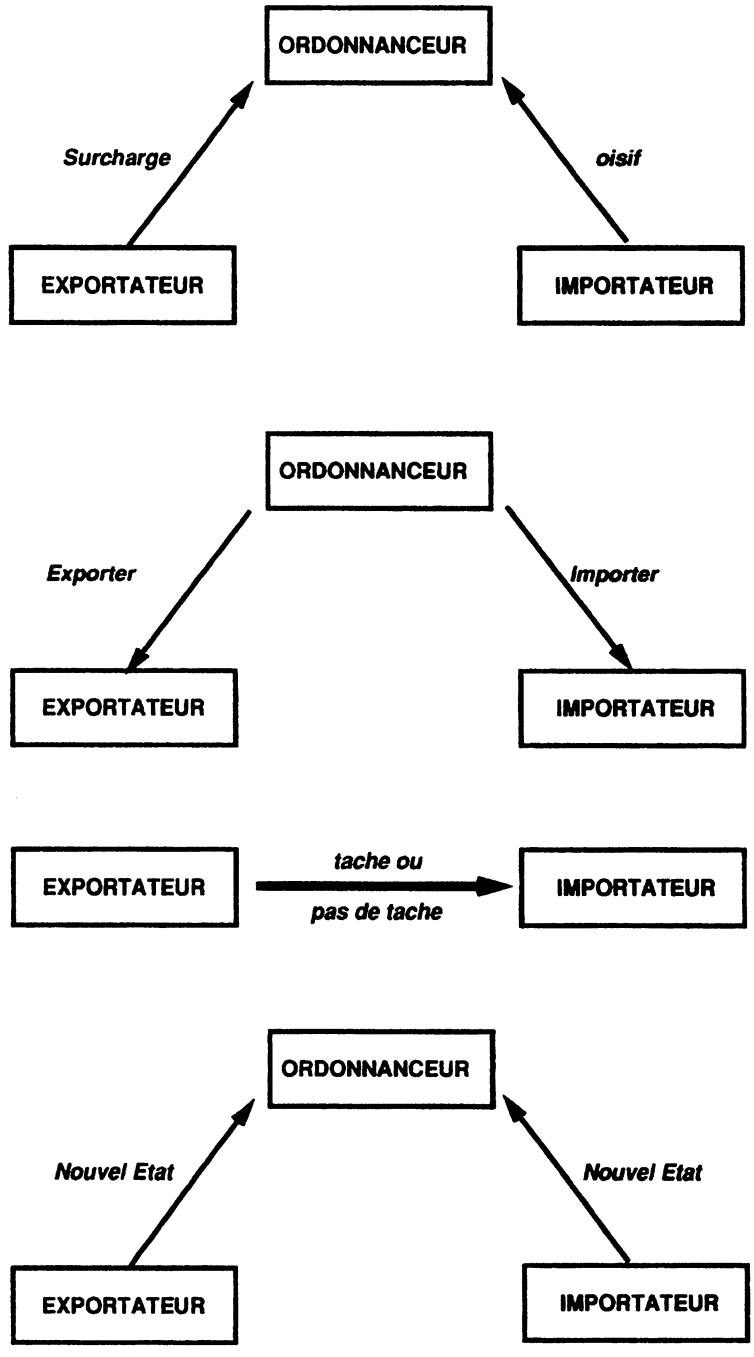


Figure 6.2 : protocole de synchronisation et d'allocation de routes

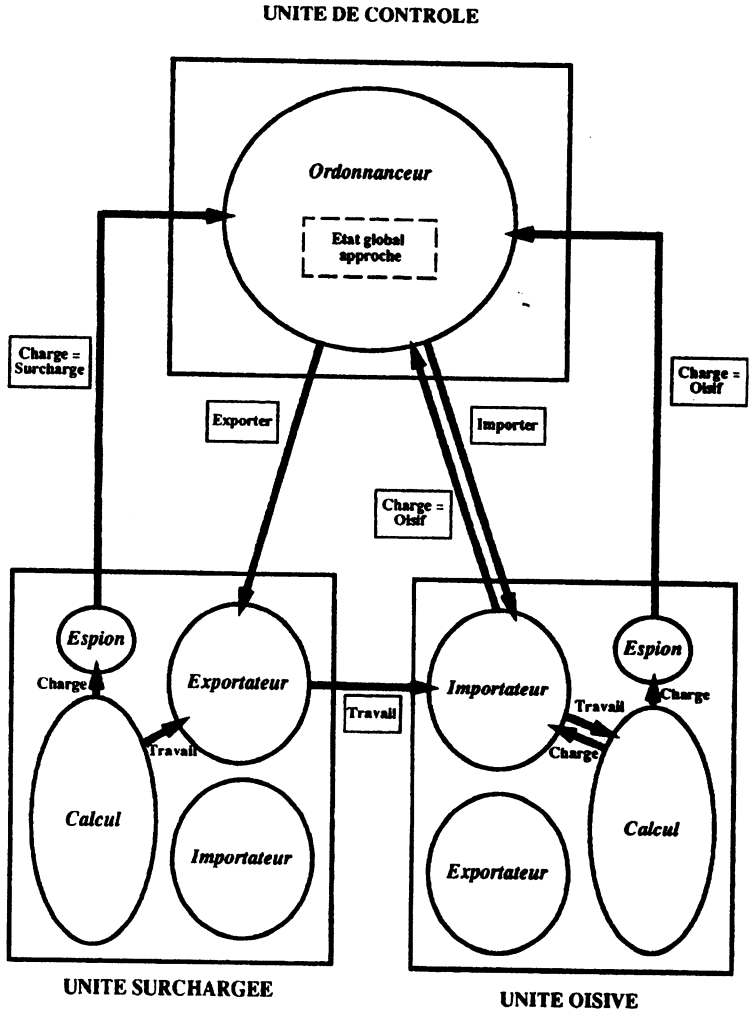


Figure 6.3 : les processus d'OPERA

a) L'unité de traitement : le "*Calculateur*"

Ce processus est chargé des fonctions suivantes :

- exécution de code de la TWAM. (programme Prolog)
- demande de travail à la partie contrôle (processus *Ordonnanceur*) puis réception de la réponse de l'Ordonnanceur (description d'un port par lequel une tâche peut être importée).
- contrôle de la réception des données d'une tâche importée
- restauration de l'état de la TWAM après la réception des données d'une tâche

Le code de ce processus est constitué du programme de l'utilisateur et de tout ce qui est nécessaire à son évaluation. Il est spécialisé dans l'exécution des instructions de la TWAM et se comporte comme s'il était seul à évaluer l'ensemble de l'arbre OU du programme. Tout au plus, il met à jour des données utiles à la redistribution du travail (datation des liaisons). Pour ce processus, tout se passe comme si l'environnement n'existait pas.

b) L'unité de transfert : processus *Exportateur* et *Importateur*

Son rôle essentiel est de dialoguer avec l'environnement de l'unité de travail pour décharger le *Calculateur* chaque fois que cela est possible et intéressant. La tâche principale du processus *Exportateur* est donc la communication. Il permet de désynchroniser le travail effectué par le "*Calculateur*" et le dialogue avec son environnement. En particulier les messages en provenance de cet environnement et qui n'ont pas de rapport direct avec l'évaluation en cours dans le *Calculateur* ne doivent pas perturber celui-ci. D'autre part, lors d'émission de messages vers l'environnement, on utilise l'asynchronisme du transfert par rapport au *Calculateur* pour minimiser le surcoût dû au parallélisme.

Exportateur : Ce processus est chargé des fonctions suivantes :

- réception d'une demande d'exportation transmise via la partie contrôle.
- choix du point de choix local à exporter
- contrôle de l'envoi des données : envoi de la description des données (partie exportée de la pile des point de choix). Si l'état de l'unité sollicitée pour exporter a évolué depuis l'échantillonnage fourni à la partie contrôle et ne correspond plus au macro-état surchargé, ce processus transmet une marque spéciale qui signale qu'il n'a aucun travail à fournir, sinon les portions de piles de la TWAM surchargée sont émises. Pour masquer les temps d'entrées-sorties par du calcul, la pile des variables est émise avant les autres piles de manière à ce que le CPU oisif puisse effectuer le plus tôt possible le traitement de la pile des variables pendant le transfert des autres piles.

- acquittement de l'exportation qui permet de libérer la connexion et de signaler le nouvel état après exportation.

Importateur : Ce processus est chargé des fonctions suivantes :

- réception d'une demande d'importation de la part du *Calculateur* (activation du processus d'importation)
- transmission de cette requête à l'*Ordonnanceur*.
- réception de la réponse de l'*Ordonnanceur* (qui signale l'établissement de la liaison).
- contrôle de la réception des données : réception de la description des piles à recevoir puis, le cas échéant, annulation du transfert.
- Si le transfert est possible, attente de la réception totale de la pile des variables
- déliaison des variables qui ont été liées après la création du point de choix transferé (liaisons conditionnelles). Défaire les liaisons pendant que les autres piles sont en cours de transfert
- attente de la réception totale des autres piles
- réactivation du *Calculateur* dès la fin de l'importation
- acquittement de l'importation (qui permet de libérer la connexion)

c) L'unité de contrôle : l'*Ordonnanceur* et son outil d'échantillonnage

Les unités de travail décrites ci-dessus communiquent leurs charges via le bus de contrôle au processus *Ordonnanceur*.

Espion : Ce processus, placé sur chacune des unités de travail, envoie, de temps en temps, la charge du *Calculateur* à l'*Ordonnanceur*. Ce processus, réveillé par l'échéancier, doit permettre de désynchroniser la partie opérative et la partie contrôle pour que l'évaluation séquentielle des branches OU ne soit pas ralentie par le contrôle global.

Le délai entre deux échantillonnages est un paramètre du système. Il a été introduit comme un moyen de règler le comportement du système et d'éviter le goulot d'étranglement que pourrait constituer le bus de contrôle. En outre, si une fréquence d'échantillonnage élevée permet à l'*Ordonnanceur* de maintenir un état global approché presque parfait qui limite le nombre de décisions d'appariement inappropriées, l'échantillonnage prend une proportion importante du temps d'une unité de travail. Sur notre Tnode, ce phénomène est accentué par le fait que l'espion ne peut accèder au bus de contrôle qu'en attente active d'émission du fait d'une erreur matérielle.

Les nombreuse micros-variations de la charge instantanée sont éliminées au moyen d'une fonction de lissage appliquée à chaque échantillonnage (réveil du processus *espion*). Cette fonction de lissage est actuellement une moyenne pondérée:

$$nouvelle\_charge = (charge\_courante + ancienne\_charge)/2$$

De même, le processus *espion* n'émet une nouvelle valeur de charge que si la variation de celle-ci dépasse une certaine valeur. Cet incrément est lui aussi un paramètre du système. La fonction de lissage n'est pas appliquée lorsque la charge courante est négative (-1) car il est important de détecter le plus tôt possible l'inactivité de l'unité de travail.

***Ordonnanceur*:** Ce processus est placé sur le processeur de contrôle. Il reçoit successivement les informations en provenance de chaque *espion* (état de charge) ou des processus *Importateur* et *Exportateur* (fin de transfert). Il classe les unités de travail en 3 catégories selon la charge:

- inactive
- isolée
- surchargée

et un état supplémentaire correspondant à un échange de travail entre deux unités. La transition d'isolée à surchargée correspond au franchissement d'un seuil de charge qui est un paramètre du système. Les transitions d'état entraînent les traitements suivants:

Lorsqu'une unité devient inactive

- L'unité de travail signale une charge négative (-1 = unité oisive) à l'*Ordonnanceur.*
- L'*Ordonnanceur* met à jour l'état approché et cherche l'unité de travail la plus chargée (il maintient une liste des unités de travail surchargées ordonnée par charge décroissante).
- Si aucune unité de travail n'est en surcharge, on mémorise simplement qu'une unité est en attente de travail.

  Si le nombre d'unités de travail inactives est égal au nombre d'unités de travail dépendantes de cet *Ordonnanceur* alors ce dernier détecte la fin (globale) du programme.
- S'il existe une unité en surcharge, l'*Ordonnanceur* alloue une route logique entre l'unité inactive et l'unité surchargée puis informe les processus *Importateur* et *Exportateur* de cet appariement. Ces messages permettent aux unités de travail de savoir à partir de quand la route est disponible.

- Les deux unités sont répertoriées en état de transfert pour éviter qu'elles ne soit resélectionnées aussitôt.

- On quitte l'état de transfert lorsque les deux unités appariées ont acquitté le transfert. Cet acquittement permet de désallouer la route logique utilisée. L'étiquettage des acquittements par les nouvelles charges permet d'éviter une trop grande dérive de l'état global approché par rapport aux états locaux réels. Ils permet également de mesurer le taux de décisions d'appariement inopportunes.

Lorsqu'une unité devient "isolée" Le processus *Ordonnanceur* est simplement informé de cet état de charge et ignore cette unité qui n'a pas suffisamment de travail pour en fournir à son environnement.

Lorsqu'une unité est en surcharge Le processus *Ordonnanceur* essaie d'apparier cette unité avec une unité inactive :

- L'*Ordonnanceur* met à jour l'état et cherche une unité de travail inactive.

- Si aucune unité de travail inactive n'est répertoriée on mémorise simplement qu'une unité est en surcharge.

- S'il existe une unité inactive le processeur de contrôle alloue une route logique entres les deux unités de travail et leur signale cet événement.

- Les deux unités sont répertoriées en état de transfert.

6.3 Mise en œuvre

6.3.1 Mode de communication entre processus

a) Interactions entre unités de travail et unité de controle

Chaque Transputer de la partie opérative (T800) communique directement avec le Transputer de contrôle (T414) par une liaison bi-point maître-esclave statique. Cette liaison sert de voie de service (arrêt , initialisation physique de Transputer) mais aussi de voie de contrôle pour véhiculer des informations sur les besoins de connexions et l'état de la partie opérative. La fonction de diffusion est utilisée par OPERA pour synchroniser ses unités de travail avant exécution d'un programme.

Ces interactions se font via le bus de contrôle à l'aide de la fonction de transfert d'octets et des signaux de "N vers 1" (ALL) et "1 parmi N" (ANY) (cf documentation technique du Supernode [Wail90]). L'émetteur et le récepteur sont obligés de se synchroniser à chaque octet transferé car il n'existe pas de mécanisme matériel de gestion de tampons (fifo).

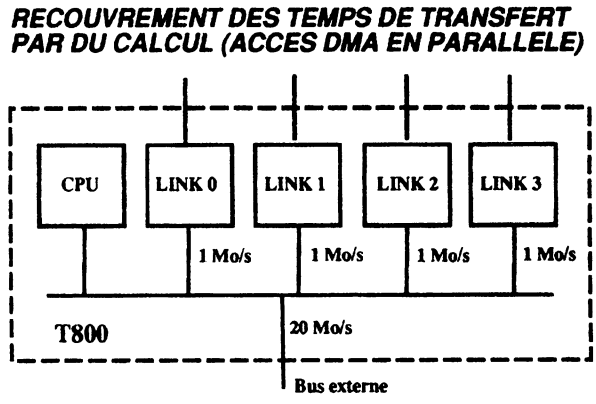


Figure 6.4 : recouvrement des E/S par du calcul

b) Interactions entre unités d'échanges

b.1) utilisation de réseau d'interconnexion du Tnode Dans notre prototype, nous nous intéressons à un mode d'utilisation très particulier du réseau d'interconnexion du Tnode : la reconfiguration dynamique systématique pour relier directement unité exportatrice et unité importatrice. Ce modèle de liaison point à point établie dynamiquement permet d'éviter le routage coûteux par la méthode "store and forward" que nécessite l'utilisation d'un réseau de Transputer à topologie fixe. Cette méthode de connexion bipoint est utilisable sur le Tnode car notre allocateur de liens prend en compte les contraintes matérielles imposées par la structure du réseau d'interconnexion en établissant toujours 2 liaisons entre deux processeurs (cycle élémentaire). L'architecture de ce réseau et les contraintes liés à la configuration sont décrites [Wail90]. L'allocateur de liens gère un ensemble de paires de processeurs. Le problème de l'interblocage lors des allocations des voies de communications est éliminé par la restriction de ne faire communiquer une unités de travail qu'avec une seule autre à un instant donné. Cette propriété n'est pas indispensable dans le modèle OPERA mais elle simplifie beaucoup la gestion de la dissymétrie des ports des processeurs du Tnode.

b.2) utilisation du parallélisme interne au Transputer Dans l'architecture Tnode, bien que la commutation de circuit physique soit très rapide, la gestion de la connexion dynamique coûte relativement cher et le coût d'initialisation devient important pour les petits messages (quelques octets). Ce coût élevé est avant tout dû aux performances dérisoires du bus de contrôle par lequel doivent transiter les requêtes et les acquittements de connexion dynamique. Par-contre la technique de reconfiguration dynamique devient très intéressante pour les grands messages car le Transputer permet de copier efficacement le contenu des blocs de mémoire. Cette capacité est largement mise à profit par le modèle OPERA et son implantation sur Transputer : le CPU du

Transputer continue à exécuter le code de la TWAM pendant que les processeurs de lien effectuent le transfert en DMA

Le Transputer oisif peut copier un bloc de mémoire du Transputer actif sans interrompre le calcul de ce dernier si ce n'est pour initialiser le Transfert: Le surcoût du transfert effectif se borne aux vols de cycle mémoire sur le bus externe du Transputer: les accès à la mémoire par les unités de DMA intégrée dans le Transputer sont entrelacés avec ceux de CPU.

b.3) réalisation du protocole d'échange entre unités de travail La structure du Tnode nous a fait choisir la réalisation des routes logiques par l'allocation dynamique de liens. La technique de préallocation (statique) de routes a été rejetée sur ce type d'architecture à cause du surcoût important induit par les techniques de routage. Le bus de contrôle du Supernode est utilisé comme voie de transmission des messages de contrôle (voir figure 6.2) tels que l'état de charge des différentes unités de travail et les messages signalant la disponibilité d'une route logique. L'allocation d'une route logique lors d'une décision d'appariement correspond à l'établissement de deux connexions physiques (liens) entre les 2 Transputers T800 concernés. Ces connexions sont établies par le processeur de contrôle sur lequel est placé l'*Ordonnanceur* et le gestionnaire de connexions.

Dans le but d'optimiser les performances, principalement celles du transfert des données d'une tâche exportée, plusieurs processus ont été ajoutés dans l'unité de transfert:

***EmetPile$_i$*:** Les processus *EmetPile$_i$*, au nombre de $2(i = 1, 2)$, ont été crées pour pouvoir transférer les données d'une tâche en utilisant au maximum la bande passante des 2 liens du Transputer. Ils sont activés par le processus *Exportateur*. Chaque *EmetPile$_i$* utilise un lien différent et chacun envoie une partie égale des données. En multiplexant les messages du système et ceux de l'application (OPERA dans notre cas), on devrait pouvoir aller jusqu'à 4 processus de ce type au prix d'une plus grande complexité du système.

RecoisPile$_i$ Ces 2 processus sont les correspondants respectifs, du coté de l'unité de travail importatrice, des processus *EmetPile$_i$*. Ils sont chargés de la réception des données de la tâche. Ces processus sont activés par le processus *Importateur*. Chacun d'eux doit signaler la fin de réception de sa partie de la pile de variables (transférée en premier lieu) à ce processus *Importateur*, alors inactif, pour que celui-ci applique l'opération "undo", puis terminer la réception des autres piles et en informe l'*Importateur*.

Une route logique est effectivement libérée lorsque ses deux extrémités (ports) ont été signalées libres[5]. En réalité cette libération n'est que logique: les voies physiques

[5]En fait, du point de vue de la réalisation, on pourrait économiser un transfert sur le bus de contrôle en éliminant soit le message de libération du port de l'émetteur soit le message de libération du port du récepteur car il y a forcément une relation de précédence entre les deux événements. Seul le dernier nécessite d'être acquitté. Cette relation de précédence étant

correspondantes ne sont pas interrompues aussitôt car la tâche "connexion_manager" qui pilote réseau de commutateur, maintient un cache de connexion qui évite de réaliser une liaison déjà existante et ne supprime une liaison marquée libre que lorsque cela devient impératif [6].

c) Interactions entre unité de calcul et unité d'échange

Le couplage entre le processus *Calculateur* et le processus *Exportateur* d'une même unité de travail doit être minimisé pour que l'environnement ne perturbe pas trop le comportement du *Calculateur*. Cependant le processus *Exportateur* étant chargé d'envoyer toutes les données nécessaires à l'évaluation des points de choix exportés, il doit pouvoir y accèder donc les partager avec le *Calculateur*. Rappelons quelles sont les synchronisations entre *Calculateur* et *Exportateur* qui garantissent la cohérence des données transférées en parallèle de leur modifications par le *Calculateur*: il s'agit

- du règlement du conflit d'accès à la file des point de choix, et en particulier de l'inhibition du retour arrière sur des tâches en cours d'exportation.
- de la cohérence des données transférées relativement aux opérations de liaison/déliaison.

  Les accès effectués par le processus *Exportateur* ont la propriété suivante : toutes les données sont accédées uniquement en lecture (lors de leur exportation) à l'exception du retrait d'un point de choix de la file des points de choix.

Les opérations d'insertion ou de retrait dans la file de points de choix, se font

- du côté du *Calculateur* par les instructions "try" et "trust" qui modifient ces données (création et destruction de points de choix);
- du côté du processus *Exportateur*, seulement au moment où il retire le point de choix à exporter de la file de points de choix.

Du fait du travail en parallèle du processus *Calculateur* et du processus *Exportateur*, on doit garantir la cohérence du transfert et des piles transmises. Les seules opérations modifiant les piles Prolog sont: les opérations d'empilement / désempilement et les opérations de liaison/déliaison de variables.

- Les opérations d'empilement ne posent aucun problème puisque seules les portions basses des piles sont transférées (les opérations d'empilement sont postérieures aux points de choix exportés donc n'accèdent pas à ces portions de piles).

---

fortement dépendante du protocole de communication utilisé pour le transfert de pile nous avons préféré conserver les deux messages de manière a pouvoir changer ce protocole sans remettre en cause la partie contrôle.

[6] Le circuit crossbar switch développé pour le projet ESPRIT ne permet pas la déconnexion, aussi, on compose une déconnexion et une connexion pour former une permutation de connexions. Notre serveur de connexion gère le fait que certaines liaisons matérielles sont disponibles avant que les clients ne les demandent.

- Les opérations de désempilement n'affectent la zone transferée que si le *Calculateur* effectue un retour arrière jusqu'aux points de choix exportées. Dans ce cas deux solutions sont envisageables :

  - Le *Calculateur* est suspendu jusqu'à la fin du transfert puis se retrouve dans l'état inactif (c'est la solution retenue).

  - Le transfert est interrompu et annulé, la TWAM exportatrice continue normalement la résolution et l'importateur doit rechercher un autre travail. Cette solution plus efficace n'a pas été retenue pour la raison suivante : elle nécessite un mécanisme spécialisé entre une unité de travail et son environnement permettant d'annuler un transfert en cours et les Transputer n'offrent pas la possibilité d'interrompre un transfert en DMA en laissant le processeur de lien dans un état cohérent. Une solution aurait été de partitionner les messages en paquets pour permettre le test périodique de la condition de validité du transfert. Le surcoût de cette solution est payé même pour les transferts valides, nous avons donc décidé de repousser la réalisation d'un tel mécanisme après l'obtention de données quantitatives permettant d'en déterminer l'opportunité.

- Les opérations de liaison/déliaison de variables peuvent modifier la pile des variables dans la zone transférée. Comme nous l'avons vu dans la description de la TWAM, la pile des variables est composée de doublets (valeur , date de liaison). La machine fournit une synchronisation, très élémentaire, entre unité de traitement et unité de transfert : l'accès au mot mémoire. La date elle même doit avoir une taille suffisante pour éviter les problème de débordements. Nous avons choisi de l'implémenter sur un mot. Les cellules représentant les variables sont des structures comportant plusieurs mots.

  Pour garantir la cohérence du transfert, il suffit de choisir de coder les variables libres avec une date égale à $+\infty$ [7]. En effet, si l'on prend cette convention, toute modification d'une cellule de la partie importée, qu'il s'agisse de liaison ou de déliaison, effectuée par le *Calculateur* ne peut être effectuée qu'à une date postérieure à la date du point de choix transféré le plus récent. Cela laisse donc invariant le fait que la date de la cellule considérée est strictement supérieure à la valeur du champ CLOCK du point de choix exporté le plus récent.

  Le travailleur importateur retrouve une pile des variables cohérente après application de l'opération "undo" (remise à libre) sur toutes les variables dont la date est supérieure à la date de création du point de choix transferé. En effet, même si, momentanément, le champ valeur des cellules variables modifiées pendant leur exportation a pu avoir une valeur indéterminée [8], le champ date de ces variables

[7] En fait, n'importe quelle valeur strictement supérieure à la date maximale que peut atteindre le compteur CLOCK : typiquement on choisit l'entier maximum 2^{32}-1.

[8] Le champ valeur pourrait avoir une taille quelconque et être affecté de manière totalement asynchrone à sa lecture (pas d'exclusion mutuelle).

[9] reçues reste toujours supérieur à la date d'exportation ce qui permet de réinitialiser systématiquement ces variables à (+∞,libre) quelque soit le doublet (date,valeur) reçu.

Moyennant ces quelques précautions l'exportation peut être réalisée très efficacement par des transferts en "DMA" de gros blocs sans requérir de synchronisation avec l'unité centrale utilisée par le *Calculateur*.

6.3.2 Environnement système

a) Les différents outils

Au début du projet OPERA, le Tnode n'était qu'à l'état de conception et nous avons dû développer les premiers modules d'OPERA sur des cartes mono-Transputer accessibles à partir de micro-ordinateur PC. A la réception de la première machine Tnode (prototype), il n'y avait que très peu de logiciel disponible sur Tnode et, en particulier, aucun système d'exploitation. Le consortium du projet ESPRIT 1085 s'était restreint au langage OCCAM et à son environnement très fermé TDS. Le langage OCCAM n'est pas très bien adapté à l'implantation de Prolog. La carence de logiciel fonctionnant sur la Tnode nous a conduit à développer nos propres outils. La chaîne de développement utilisée pour notre prototype a entièrement été développée par notre équipe dans le cadre de ce projet. Cette chaîne comporte :

- un compilateur Prolog (syntaxe C-Prolog) écrit en C-Prolog générant le code intermédiaire de la TWAM. Ce compilateur admet la compilation séparée.
- une phase de macro-génération du programme écrit dans le langage de la machine virtuelle (produit par le compilateur Prolog) en assembleur Transputer.
- un assembleur Transputer, un éditeur de liens et gestionnaire de librairies
- un compilateur C à la norme ANSI pour lequel nous avons ajouté quelques extensions parallèles au langage (canaux typés, alternative...) ainsi que son environnement (processus légers, E/S, noyau de communication...). Ce compilateur nous a permis de créer tout l'environnement système indispensable sur le Supernode pour réaliser notre prototype.
- un petit noyau de système d'exploitation assurant le chargement des programmes sur les réseaux de Transputer, le service fichier "à la Unix" accessible à partir de tous les sites de manière transparente, la gestion multi-fenêtres des E/S des processus légers (en cours de portage sous X-windows), la mise au point, la gestion simplifiée de tâches, l'utilisation de protocole clients/serveur et l'appel de procédure à distance.

[9] Le champ date est représenté sur un seul mot mémoire; la sérialisation des accès mémoire suffit à garantir l'exclusion mutuelle entre les modifications qui sont apportées par le *Calculateur* et l'accès en lecture qui est effectué par l'unité de transfert.

- un outil de permettant d'exploiter (post-mortem) les traces d'exécution pour la mise au point et la prise de mesures. Cet outil permet une ré-exécution partielle du programme de régulation de charge sur la machine hôte et la visualisation graphique de toutes les connexions établies dynamiquement. Cette trace temporelle permet également de visualiser l'état de charge d'une unité de travail ainsi que la durée des échanges entre les unités de travail Prolog.

- deux noyaux de communications multi-protocoles utilisant la technique de routage. Après leur évaluation, nous avons choisi d'utiliser la technique de configuration dynamique pour Prolog et de conserver la technique de routage mise au point pour le système (pour faciliter la mise au point). L'un de ces noyau de communication a servi de base de travail dans le projet ESPRIT Supernode 2 qui vise à la conception d'un système d'exploitation pour les machines de la gamme Supernode.

Nous ne détaillerons pas plus cet environnement bien qu'il ait nécessité un effort de conception, de développement et de validation considérablement plus important que celui d'OPERA, car la conception des logiciels "système" indispensables au fonctionnement de toute machine parallèle, dépasse largement du cadre de la programmation logique parallèle. Le lecteur intéressé trouvera dans l'annexe B, une description sommaire de l'environnement système et du C parallèle que nous avons développé et utilisé à cette occasion.

b) difficultés rencontrées

Les difficultés rencontrées dans la réalisation de notre prototype ont été très nombreuses. Il s'agit principalement de difficultés de mise au point :

- L'asynchronisme et l'absence de temps global rend complexe la détermination de l'ordre exact des événements.

- Le non déterminisme rend certaines erreurs difficilement reproductibles, (en particulier les erreurs de synchronisation) ;

- L'absence de mémoire commune rend très difficile le maintien d'un état global exact observable (méthode inexploitable sur Supernode car trop inefficace).

- Les faibles performances des entrées-sorties (de notre station "hote") devant les capacités de calcul du Tnode rendent l'observation "en ligne" quasiment impossible car elles perturbent beaucoup trop le comportement du programme à observer. De plus la latence des communications introduit une déformation de l'image du système. La mise au point a essentiellement été réalisée "post-mortem" en effectuant une réexécution "symbolique" du programme sur la machine hôte à partir des traces de comportement obtenues par l'exécution réelle.

- l'absence de support matériel sur les Transputer est un handicap pour la mise au point de programmes séquentiels : il n'y a pas d'instruction de point d'arrêt, pas de protection mémoire, l'état courant du Transputer n'est pas observable dans sa totalité!. Dans un programme parallèle le problème est accru car une petite erreur localisé peut très facilement provoquer une réaction en chaîne qui bloque tout le système et rend extrêmement difficile son repérage.

- difficultés liées à la nature des machines Supernode[10].

6.4 Premiers résultats

Les résultats présentés dans cette section résument une première campagne de mesures. Ces mesures ont été effectuées sur des programmes n'effectuant aucunes entrée-sortie. Le "cut" séquentiel classique a été implanté dans notre prototype mais nous n'avons pas encore réalisé l'implantation du "cut" parallèle ni les prédicats à effet de bord. Nous disposons cependant du minimum de prédicats de sortie nous permettant d'afficher dans des fenêtres différentes, les groupes de solutions trouvées par chaque processeur.

6.4.1 Performances du Transputer

Avant de donner des mesures précises de durées d'exécution de Prolog, examinons les performances du processeur cible. Le Transputer est un processeur 32 bits habituellement crédité d'une puissance de 10 MIPS (selon INMOS). En pratique, cette puissance théorique est difficilement comparable à celle des processeurs classiques comme ceux de la famille 680x0 ou 80x86 pour les raisons suivantes :

- Il faut un grand nombre d'instructions Transputer pour réaliser l'équivalent d'une instruction 68000 car les instruction sont extrêmement simple (elles ont au plus un argument explicite). D'une certaine manière, le Transputer peut être considéré comme un RISC (seulement 13 instructions les plus utilisées, sont exécutables en un seul cycle grâce à un codage sur un seul octet). Son jeu d'instruction ne comporte qu'un minimum de mode d'adressage : l'adressage immédiat (le système de préfixe rend très coûteux le chargement des constantes codées sur 32 bits) et l'adressage indirect indexé.

- Il n'existe qu'un seul registre de base (W) ce qui est très insuffisant pour implémenter efficacement les langages classiques (la machine TWAM comporte un grand nombre de registres de bases).

[10]Il s'est agit de travailler sur machines en cours de conception et de mise au point, avec des documentations très imprécises, voire inexistantes. Cela nous a fait perdre beaucoup de temps et nous a réduit à mettre au point les programmes pilotes des organes essentiels du Supernode (bus de contrôle et crossbar switch) par des phases d'essai-erreur

- Le Transputer ne possède pas de registres aux sens classique : une pile interne utilisée pour stocker un par un les opérandes d'un opérateur se limite à 3 places. La commutation de processus ne sauvegarde/restaure que le compteur ordinal et le registre W. Elle est très efficace mais la priorité donnée à l'accélération de la commutation de contexte qui est un avantage indéniable pour la multiprogrammation du processeur, est un sévère handicap pour l'implantation de certains langages comme Prolog.

- Le Transputer ne possédant pas un mécanisme efficace de décalage cablé, la gestion de champs de bits est très coûteuse. L'accès à l'octet est également coûteux (6 cycles). La gestion d'entier sur 16 bits doit être réalisée par logiciel et coûte jusqu'à 30 cycles dans certains cas ! Toutes ces considérations font qu'il est difficile de gérer des structures compactes (pointeurs taggés par exemple). L'alternative est de privilégier l'accès au mot de 32 bits au prix d'une augmentation du nombre d'accès mémoire (déjà très élevé dans cette machine "sans registres").

- l'interface d'accès à la mémoire externe est du type multiplexé et nécessite au moins 3 cycles (typiquement $\simeq$ 150 ns). Aucune utilisation de cache n'est prévue.

- Pour limiter cette insuffisance, une mémoire d'accès rapide (1 cycle) est intégré sur le chip. Sa taille varie de 2 Ko/s sur les T414 à 4 Ko/s sur les T800. Des mesures ont montré que le placement d'un programme dans cette mémoire permet une accélération de 1,8 à 3 par rapport à un placement en mémoire externe.

- La performance moyenne du Transputer est très inférieure à la performance en crête qu'il serait possible d'atteindre si tout le programme et ses données pouvait résider dans les 4 Ko de mémoire interne et si le programme ne comportait que des instructions exécutables en un seul cycle [11]. En réalité, la vitesse d'exécution est extrêmement liée aux performances de la mémoire externe et elle est, par conséquent, difficilement comparable aux autres processeurs récents qui sont presque tous avantagés par la gestion de cache. Ceci devrait évoluer avec l'arrivée de la prochaine génération de Transputers (T9000) pour lesquels cette mémoire interne à une taille de 16 Ko/s et est gérée en cache.

La plupart des compilateurs du commerce ne tirent pas parti de la mémoire interne du Transputer car il est difficile pour un compilateur d'exploiter efficacement cette mémoire hétérogène. Une solution possible pour un compilateur est d'utiliser la mémoire interne pour émuler des registres et pour contenir des routines fréquemment appelées. Ainsi, cette mémoire peut contenir les registres de la TWAM et les routines d'unifications. Notre compilateur C// y émule des registres de bases et y place les variables de la classe "register".

[11] il y en a seulement 13 sur près d'une centaine qui vérifient cette propriété.

Program	T_1		T_2		T_3	
	ms	Klips	Ms	Klips	Ms	Klips
fib (15)	209	23	143	34	143	34
deriv	1353	7	1285	8	813	13
hamilton	35593	13	34116	14	20335	24
nrev (30) *100	39979	19	26953	21	23123	34
queen (4)	284	24	269	25	179	38
query	155	14	148	15	104	22
quickdiff	1933	15	1872	16	1139	26
quicksort	346	17	327	18	204	29
farmer *100	3729	8	3542	8	2112	14
map	1795	13	1693	13	1009	22
queens1 (8)	5787	18	5440	19	3663	28

Tableau 6.1 : Performances séquentielles selon l'usage de la RAM interne au T800

6.4.2 Efficacité du calcul séquentiel

La version 0.0 d'OPERA est une version préliminaire qui ne comporte aucune optimisation et offre déjà une bonne efficacité. Les mesures effectuées sur un premier prototype séquentiel ont montrés qu'il était possible d'atteindre une vitesse raisonnable : 143 ms pour le calcul de la fonction de fibonacci (fib(15)) soit 34 Klips (KiLo Inférences par Secondes) sur un T800 fonctionnant à 20 Mhz en plaçant les registres de la TWAM et une partie du code dans la mémoire interne. Les performances tombent a 23 Klips pour fib(15) si l'on n'utilise pas cette mémoire rapide.

Le code de la TWAM généré par le compilateur est expansé en code assembleur plutôt que d'être émulé comme dans la plupart des implémentations Prolog existantes. Si l'efficacité de la partie séquentielle de Prolog dépend beaucoup de ce codage, il n'en est pas de même pour la partie parallèle. L'efficacité de l'exécution parallèle doit beaucoup á celle de l'environnement de Prolog et celle du système qui sont, tout deux, écrits en C//.

A notre connaissance, la TWAM est la plus efficace des implantations Prolog existant à ce jour sur Transputer.

Dans la table 6.1, T_1 est une version de TWAM où tous les registres et tout le code sont placés en mémoire externe. Dans la version T_2, le code Prolog compilé est chargé en RAM interne. Dans la version T_3, les registres de la TWAM et la partie du "run-time" la plus utilisée sont placés dans la RAM interne Les unités de mesures sont la milliseconde pour le temps d'exécution absolu et le millier d'inférences par secondes (Klips) pour la vitesse.

L'optimisation T_3 n'a pas encore été réalisée dans la version parallèle car le système qui la supporte, ne gére pas l'allocation de la mémoire interne du Transputer. Sur cette version, l'exécution séquentielle du programme sur un seul processeur *naive-reverse*

atteint 21 KLIPS.

6.4.3 Programme de test du parallélisme

Plusieurs petits programmes de test, principalement ceux utilisés par l'ECRC, ont été utilisés pour mesurer l'efficacité de l'exécution parallèle d'OPERA. L'absence de méta-prédicat du type "one-solution" ou de la coupure parallèle dans le prototype OPERA nous a contraint à modifier légèrement certaines parties de ces programmes. Le texte source de quelques programmes testés est fourni dans l'annexe A. La figure 6.5 et la table 6.2 se rapportent aux programmes:

hamilton : le problème consiste à trouver un cycle hamiltonien dans un graphe. Le graphe utilisé pour ce test comporte 20 sommets et 60 arcs. Toutes les solutions sont calculées (soit environ 460 000 inférences).

map : calcule toutes les solutions au problème classique de coloriage d'une petite carte avec 4 couleurs (soit environ 22 800 inférences).

queens1 : Il s'agit du problème du placement de 8 reines sur un échiquier 8 x 8 de telle manière qu'aucune d'entre elles ne soit en prise par une autre. Le programme de test correspond à la solution qui consiste a répertorier les lignes et les colonnes déjà obtenues par calcul de solutions partielles de manière a éviter de les réutiliser ultérieurement (cf annexe A); cela permet de réduire la taille de l'espace de recherche. Toutes les solutions (soit 92 pour le problème des 8 reines) sont calculées ; ce qui représente approximativement 103 000 inférences.

queens2 : ce programme correspond à une solution plus "naturelle" au problème des reines: chaque ligne est testée pour chaque colonne. Les 92 solutions du problème des 8 reines sont calculées ce qui représente approximativement 233 000 inférences.

Contrairement aux mesures effectuées pour d'autres systèmes Prolog OU-parallèle [Chas89a] [Szer89] [Ali90] [Bosc90], les programmes de tests ne contiennent que du Prolog sans aucune déclaration additionnelle concernant l'opportunité du parallelisme. Il n'y a donc pas de restriction explicite quant au point de choix à utiliser pour créer une nouvelle tâche. Il est intéressant de remarquer que, malgré cela, le prototype OPERA permet un gain de vitesse effectif par rapport à l'exécution séquentielle.

6.4.4 Résultat des mesures de test parallèle

Comme pour les systèmes multi-séquentiels à mémoire commune, le gain de performance de l'exécution parallèle d'OPERA par rapport à l'exécution séquentielle dépend de la taille du problème à résoudre. Ce gain de performance est linéaire pour les problèmes de grande taille. La courbe du gain de performance en fonction du nombre de processeurs de travail utilisés décroît pour les programmes les plus "petits" (voir table 6.2).

Program	1	2	4	8	12	16	Quintus Sun3/50	Quintus Sun4/60
ham	31271	16937	8122	4320	3260	2791	27100	6000
	1	1.85	3.85	7.24	9.59	11.2		
map	1568	893	515	378			1350	300
	1	1.76	3.04	4.15				
queens1-8	3890	2060	1112	682	592	540	4350	930
	1	1.89	3.50	5.70	6.57	7.20		
queens1-10	90926	45628	23029	11663	8275	6433	103300	21900
	1	1.99	3.94	7.79	10.98	14.13		
queens2-8	8571	4397	2380	1319	1085	933	8650	1867
	1	1.95	3.60	6.50	7.90	9.18		

Tableau 6.2: Temps d'exécution des programmes de tests exécutés en parallèle

La première ligne donne le temps d'exécution, mesuré en millisecondes. La seconde ligne donne le gain de l'exécution parallèle par rapport à l'exécution séquentielle (sur une seule unité de travail) du système parallèle. Les mesures d'OPERA sont effectuée sur un Tnode à 16 Transputers de travail et un Transputer de contrôle (non compté comme Transputer de travail dans le nombre de processeur cité dans les tables et figures). Ces mesures sont comparées à celles obtenues par exécution sur un SUN3/50 (crédité de 1.5 MIPS) et sur un SUN4/60 (12.5 MIPS). Le système Prolog utilisé pour ces mesures est l'un des Prolog commerciaux les plus performant il s'agit de Quintus-Prolog version 2.4.2.

Lorsque le nombre de processeurs est trop important devant la taille du problème à résoudre, leur compétition introduit un surcoût prépondérant sur le gain obtenu par la parallélisation ; dans ces conditions le choix de paralléliser cause une diminution de la vitesse d'exécution.

Les mesures données pour le système parallèle correspondent à une version où la totalité des piles est copiée. Dans cette version l'*Ordonnanceur* sélectionne toujours un seul point de choix et celui-ci est le point de choix non évalué le plus proche de la racine. En d'autres termes, on utilise la stratégie d'ordonnancement qui minimise les transferts et augmente la granularité des tâches. Dans cette version les processus d'échanges entre unités de travail n'utilisent qu'un seul lien configuré à 10 Mbits/s.

6.4.5 Optimisations

Les gains de performances donnés sont obtenus en copiant l'intégralité de toutes les piles à chaque transfert d'une tâche. Si l'on considère que la taille de ces piles augmente régulierement d'une unité entre chaque transfert le temps total passé dans le transfert est de la forme

$$1 + 2 + 3 + ...n = O(n^2)$$

où n est le nombre total des transferts effectués lors de l'exécution d'un programme. Sous la même hypothèse, la copie incrémentale donnerait un temps de l'ordre de

$$1 + 1 + ... + 1 = O(n)$$

Outre les optimisations du modèle OPERA, il existe d'autres sources potentielles d'optimisations qui sont liées à l'architecture matérielle du Supernode : la version utilisé pour les mesures n'utilise qu'un lien à 10 Mbits/s pour effectuer les transferts, ce qui offre une bande passante d'environ 970 Ko/s. Il est possible de multiplier par 8 cette bande passante si l'on utilise simultanément les 4 liens et si l'on configure les liens des Transputers du supernode à 20 Mbits. On peut également envisager d'utiliser les quatre liens pour diffuser plus rapidement le travail : un processeur exportant simultanément des travaux différents vers 4 autres processeurs au lieu d'un seul actuellement. Cette optimisation est importante surtout pour les petits programmes pour lesquels les processeurs oisifs attendent longtemps un job : la faible vitesse de diffusion fait que le système est globalement sous-chargé sur une période relativement importante devant le temps total d'exécution. L'outil d'analyse de la charge que nous avons développé pour OPERA nous a montré que c'était le cas pendant l'initialisation et la terminaison des programmes, et que ces périodes transitoires représentent une proportion importante du temps total d'exécution des petits benchmarks classiques (ex : farmer, queen(4)...). Ces optimisations architecturales ne seront mise en oeuvre qu'après modification de l'environnement système que nous avons développé sur supernode ; en particulier il est nécessaire de changer le noyau de communication qui achemine les appels système pour que la technique de connexion dynamique soit le mécanisme universel de notre système.

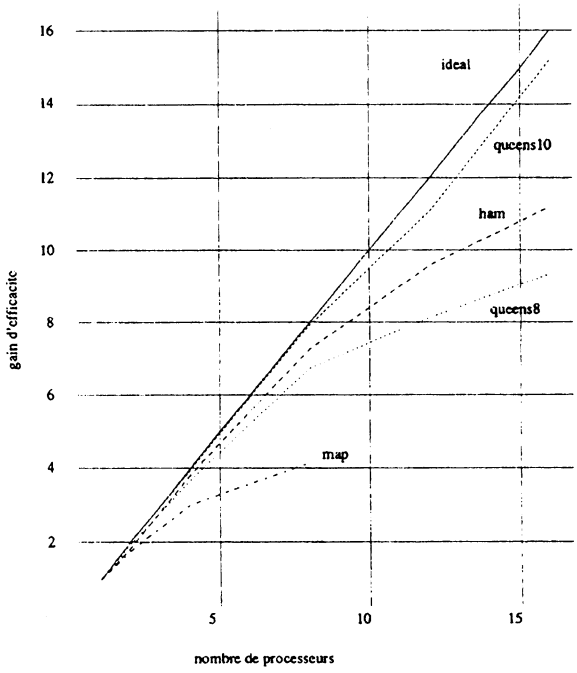


Figure 6.5 : gain d'efficacité en parallèle

Chapitre 7

CONCLUSION

7.1 Conclusions sur le travail effectué

L'objectif principal de la réalisation d'un prototype d'OPERA sur Supernode était l'obtention de données quantitatives pour évaluer l'opportunité de parallélisation de Prolog sur des machines massivement parallèles à mémoire distribuée.

7.1.1 De bonnes performances pour OPERA

La faisabilité d'un tel projet a pu être démontrée puisque notre prototype est aujourd'hui opérationnel sur des machines "à usage général" et qu'il permet déjà d'obtenir des performances comparables aux meilleurs systèmes Prolog parallèles [Lusk88] [Ali90] qui fonctionnent sur machines à mémoire commune. De nombreuses optimisations sont encore possibles, en particulier la technique de copie incrémentale doit permettre de diminuer le volume des données transférées.

Avec seulement 16 processeurs peu adaptés à l'évaluation de Prolog (les T800), le prototype fournit déjà un accroissement de performances important par rapport aux meilleurs systèmes séquentiels commerciaux (cf comparaison avec Quintus Prolog sur Sun4). La prochaine génération de Transputer (T9000) est telle que le rapport puissance de calcul sur bande passante d'E/S est conservé: ces deux coefficients ont été décuplés d'une génération à l'autre. Compte tenu des performances obtenues avec notre petit prototype, on peut raisonnablement espérer obtenir une performance de plusieurs millions d'inférences par secondes (en moyenne) avec seulement une dizaine de T9000. Les machines dont la construction est prévue dans le projet ESPRIT GPMIMD devraient comporter plusieurs milliers de T9000...

7.1.2 Le parallélisme implicite n'est pas une utopie

Il est important de rappeler que ces performances sont obtenues sans que le programmeur Prolog n'ait spécifié les opportunités de parallélisation. Aucune indication relative

à l'architecture cible (nombre de processeur, topologie, etc...) n'est fournie par le programmeur. Les programmes compilés pour être exécutés sur un seul processeur (en séquentiel) n'ont pas besoin d'être modifiés ni même recompilés pour être exécutés sur un plus grand nombre de processeurs (le fichier binaire exécutable est exactement le même dans les deux cas). Ceci est possible car le parallélisme n'a pas été introduit au niveau du compilateur mais "en dessous" du niveau de la machine abstraite cible de ce compilateur. Cela permet de suivre l'évolution des techniques de compilation (en séquentiel) sans remettre en cause les principes fondamentaux du modèle OPERA. Ce dernier atteint donc l'objectif de portabilité et d'extensibilité que nous nous étions fixés. La copie de données autorise la conversion de format et il est possible d'envisager l'utilisation de notre modèle dans un environnement hétérogène (ex: réseau local de stations de travail et machine parallèle). Ce dernier point conforte l'aspect portabilité.

Un point d'ombre cependant : à problème constant le modèle OPERA permet d'obtenir un gain de performances temporelles quasi-linéaire du nombre de processeurs mais le coût en taille mémoire augmente également linéairement en fonction du nombre de processeurs. Ceci est dû à la technique de duplication physique de données logiquement partagée (le code du programme est également dupliqué sur chaque nœud). Il est clair que ceci est un inconvénient du modèle: la parallélisation ne permet pas de résoudre un problème nécessitant plus de mémoire qu'il n'y en a de disponible sur un nœud alors que, en pratique, l'utilisation de machines parallèles correspond souvent à un volume de données à traiter plus important que ce qu'il est possible de traiter efficacement avec une machine séquentielle. On peut cependant augmenter la taille du problème si cette augmentation correspond à un plus grand nombre de branches d'évaluation (cas de problèmes combinatoires) car dans ce cas, la recupération mémoire lors du retour arrière permet de fonctionner avec une mémoire de moindre taille et le gain de performances temporelles est plus facile à obtenir.

7.1.3 Carence d'environnements de programmation et d'exécution

Prolog peut fournir un moyen d'exploiter le parallélisme implicitement (vis a vis du programmeur) mais les problèmes de gestion du parallélisme ne disparaissent pas pour autant. Ils sont repoussés au niveau du concepteur du système parallèle. En particulier, la gestion des différentes ressources matérielles (parallélisme réel) implique l'utilisation d'un langage à parallélisme explicite.

L'expérience acquise lors du développement des nombreux outils mis en œuvre pour réaliser notre prototype fait ressortir le manque de logiciel de base et de système pour ce genre de machine : peu de chose existe à l'heure actuelle et l'avenir de telles machines nécessitera un effort de recherche considérable. On peut citer deux raisons à cela :

- La nouveauté du processeur utilisé et l'originalité de son architecture n'a pas permis de recupérer des logiciels existant sur les machines séquentielle classiques. En ce sens, les utilisateurs des Transputers sont très défavorisés par rapport à

ceux qui utilisent des processeurs plus "classiques" pour lesquels il existent déjà un grand nombre d'outils et de compilateurs.

- Les machines parallèles à mémoire distribuée font apparaître beaucoup plus de problème liés à l'exploitation du parallélisme que les machines parallèles à mémoire partagée. Pour ces dernières, le matériel fournit "implicitement" des solutions à la plupart des problèmes de synchronisation (l'accès à la mémoire partagée est l'outil élémentaire de synchronisation). Bien que de nombreux résultats aient été obtenu en algorithmique parallèle, peu de solutions efficaces ont été mises en œuvre sur les machines commerciales. Par exemple, bien que de nombreux langages permettent de définir des processus légers/coroutines et des outils de synchronisation/communication, aucun compilateur actuel de ces langages (OC-CAM, C//, ADA, etc...) ne permet de s'abstraire de la frontière qu'il y a entre les nœuds d'un réseau de processeur sans mémoire commune. Actuellement, ces générateurs de code ne produisent que du code pseudo-parallèle (multiprogrammation) pour un seul processeur et l'on doit recourir à d'autres outils comme par exemple, des langages de configuration pour décrire le placement du graphe de processus sur le graphe de processeurs. Chaque utilisateur se pose le problème complexe du plaçement/routage pour chacun de ses programmes et pour chaque configuration de machine utilisée : avec les outils actuels il ne peut pas faire abstraction des caractéristiques de la machine cible ! Cela rend complexe la programmation des machines à mémoire distribuée.

7.2 Perspectives

Les résultats pratiques du projet OPERA sont

- une implantation du modèle qui fournit une bonne base de travail pour affiner l'étude de la parallélisation implicite de Prolog.
- de nombreux outils, sous-produits intermédiaires du projet, qui ont permis de mettre en évidence les problèmes que pose l'exploitation du parallélisme réel (parallélisme physique) et qui apportent quelques éléments de solution.

Le succès du projet OPERA permet d'envisager deux types de continuation :

7.2.1 Amélioration de la parallélisation de Prolog

a) Extension du parallélisme

L'extension du parallélisme peut porter soit sur l'augmentation du parallélisme physique (implantation du modèle OPERA sur un nombre plus important de processeurs : 128 sur le Meganode de l'IMAG) soit sur l'augmentation des opportunités du parallélisme logique (exploitation d'une forme de parallélisme ET, calcul spéculatif [Haus89]). Ces deux aspects complémentaires nécessitent un travail technique important sur le

prototype courant pour lui adjoindre toutes les bibliothèques de prédicats prédéfinis nécessaires à l'augmentation de taille et à la diversification des programmes Prolog testés.

b) Modélisation/Simulation

La lourdeur de la mise au point sur ce type de machines parallèles nous conduit à nous orienter vers la modélisation/simulation pour évaluer différentes stratégies de régulation de charge. L'évaluation de la charge d'un processeur pourrait être basée sur un calcul statistique simple portant sur des mesures et permettant de "prévoir" le futur à partir du passé (cela suppose que l'exécution de programmes Prolog a des périodes de comportement stationnaire). Le prototype existant donne une idée des concepts et mécanismes à modéliser. Il peut également servir d'outil de mesures permettant de confirmer ou d'infirmer les résultats obtenus par modélisation.

c) Extension du langage

La contrainte de la taille mémoire est importante car les machines à mémoire distribuée n'offrent pas sur chaque nœud autant de mémoire qu'en nécessitent les gros programmes Prolog. Il ne nous semble pas raisonnable de conserver le modèle de parallélisme implicite avec duplication de données pour les gros programmes. Pour ceux-ci, le découpage du problème en sous-problèmes pourrait être réalisé par le programmeur. Cela éviterait de dupliquer le code sur tous les sites. Le langage Prolog devra alors être étendu pour pouvoir exprimer un programme comme un ensemble de systèmes coopérants.

7.2.2 Aspects architecturaux

Une autre orientation possible est motivée par le manque de support matériel pour réaliser efficacement un contrôle global du parallélisme. Les études à mener dans ce sens relèvent de la conception d'architectures parallèle. Par exemple, un support maériel pour la diffusion serait très utile à la régulation de charge.

Un autre aspect, apparu lors de notre étude, consiste à examiner l'utilisation de la pagination pour accèder à une mémoire commune en conservant notre modèle de copie. L'espace d'adressage (virtuel) de chaque *travailleur* serait placé dans une mémoire globale (mémoire secondaire) partagée entre tous les processeurs. La mémoire physique locale à chaque processeur jouerait alors le rôle de cache de cet espace d'adressage. L'opération de copie de piles de notre modèle pourrait être fusionnnée avec le mécanisme de pagination pour effectuer des copies "paresseuses" et diminuer ainsi le coût de parallélisation. Le partage logique de parties communes "en lecture seule" permettrait d'éviter la duplication de données dans l'espace secondaire même s'il subsisterait une duplication progressive de ces données dans les caches (mémoires locales). De même, la taille du code ne serait plus un point bloquant. Le mécanisme de datation permettrait de réaliser traitement des variables "au vol" pendant le chargement d'une page en mémoire locale. Cette gestion de mémoire virtuelle permettrait également de multiprogrammer chaque processeur (placer plusieurs *travailleur* sur un seul processeur) afin

de masquer les temps d'entrées/sorties par du calcul. Un espace mémoire secondaire de grande capacité (un disque) permettrait de relâcher les contraintes sur taille des programmes exécutables.

Un autre point important est à prendre en considération pour l'avenir: on sait actuellement réaliser des processeurs spécialisés dans l'exécution de la WAM qui sont très efficaces mais dont les performances sont limités principalement par la bande passante de l'accès à la mémoire. L'usage de tels processeurs dans les futures machines parallèles semble donc avantager plus les machines à forte bande passante mémoire (ce qui est le cas des machines à mémoire distribuée) que les machines à mémoire partagée dont la limite est justement liée au phénomène de surcharge du bus accèdant à cette mémoire. Bien qu'étant beaucoup plus simple à gérer, les machines à mémoire commune nous semble avoir des limitations beaucoup plus strictes que les machines basées sur des mémoires distribuées.

Annexe A

PROGRAMMES D'ESSAI

A.1 Coloriage d'une carte avec 4 couleurs

```
% Programme: map.pro
% Fonction:  colorer des pays avec des couleurs differentes
% repeter l'operation 200 fois.

module(map, entry(maingoal/0)).


%coloriage   200 fois

map(N):-map_loop(N).

map_loop(0).
map_loop(X):- map_top,sub(X,1,Y),map_loop(Y).

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

map_top:-
        el(X1,[b]),
        el(X2,[r]),
        el(X7,[g]),
        el(X13,[w]),
        el(X3,[b,r,g,w]),
        not(X2=X3),
        not(X3=X13),
        el(X4,[b,r,g,w]),
        not(X2=X4),
```

```
        not(X7=X4),
        not(X3=X4),
        el(X5,[b,r,g,w]),
        not(X13=X5),
        not(X3=X5),
        not(X4=X5),
        el(X6,[b,r,g,w]),
        not(X13=X6),
        not(X5=X6),
        el(X8,[b,r,g,w]),
        not(X7=X8),
        not(X13=X8),
        el(X9,[b,r,g,w]),
        not(X13=X9),
        not(X4=X9),
        not(X8=X9),
        el(X10,[b,r,g,w]),
        not(X4=X10),
        not(X5=X10),
        not(X6=X10),
        not(X9=X10),
        el(X11,[b,r,g,w]),
        not(X11=X13),
        not(X11=X10),
        not(X11=X6),
        el(X12,[b,r,g,w]),
        not(X12=X13),
        not(X12=X11),
        not(X12=X9).

maingoal:- map(200),!.
```

A.2 Placement des 8 reines sur un échiquier

```
% d'apres le benchmarch d'ECRC Gmbh

% pour restraindre l'espace de recherche, une liste de
% valeur possible est passee a la procedure de generation
% si une ligne est selectionnee dans cette liste pour une
% colonne donnee, on retire cette ligne de la liste.
```

```
module(queenEcrc1, entry(maingoal/0)).

maingoal:-
        go(8, Soln), fail.

go( Size, Solution ):-
        poss_rows( Size, PossList ),
        solve( PossList, 0, [], Solution ).

solve( [Poss1|RestPoss], Col, Current, Final ):-
        add(Col,1,Col1),
        delete( [Poss1|RestPoss], Row, NewPossRows ),
        safe( Col1, Row, Current ),
        solve(NewPossRows,Col1,[s(Col1,Row)|Current],Final).
solve( [], _, Final, Final ).

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

safe( X, Y, [s( I, J )|Rest] ):-
        notthreaten( X, Y, I, J),
        safe( X, Y, Rest ).
safe( X, Y, [] ).

notthreaten( X, Y, I, J ):-
        add(X,Y,A),
        add(I,J,A1),
        notequal(A,A1),
        sub(X,Y,S),
        sub(I,J,S1),
        notequal(S,S1).

poss_rows( N, R ) :- poss_rows( N, [], R ).

poss_rows( 0, L, L ).
poss_rows( N, K, L ):-
        notequal(N,0),
        sub(N,1,N1),
        poss_rows( N1, [N|K], L ).
```

Annexe B

L'ENVIRONNEMENT SYSTEME ET C//

B.1 La communication en C//

Il existe de nombreuses versions de compilateur C pour des machines parallèles. Pour la plupart le parallélisme n'y est pas exprimable dans le langage et ces environnement de programmation se contentent d'offrir des primitives de bas niveau par rapport aux constructions disponibles dans les langages parallèles modernes. Si l'utilisation de bibliothèques "parallèles" offre l'avantage de pouvoir être enrichie à volonté pour un faible coût, elle implique de gros inconvénients : manque de clarté, absence de sémantique précise, absence de vérifications de cohérence dès la compilation, absence d'analyse possible par le compilateur donc pas d'optimisation automatique, etc...

Il faut intégrer le parallélisme dans le langage. Pour notre première version nous n'avons inclus directement que deux nouveaux concepts (dérivés de CSP [Hoar78]) : les canaux pour le rendez-vous synchrone et la construction alternative pour intégrer le non-déterminisme. Nous avons décidé de ne pas inclure la notion de processus dans le langage avant d'avoir examiné en détail les problèmes de fond posés dans un langage comme C. Toutefois le concept de processus qui est au coeur de la programmation parallèle est concrétisé par un ensemble de bibliothèques et de structures permettant de prendre en compte l'aspect de réseau de processus dynamique.

B.1.1 Les canaux

Il est indispensable d'introduire deux nouveaux types permettant de manipuler des objets offerts par le Transputer :

- les canaux
- les horloges.

Ces deux types seront bien entendu prédéfinis et pourront être regroupés comme tout autre type C en tableaux, structures et unions.

L'objet canal

C'est un moyen de transmission entre processus suffisamment abstrait pour masquer la nature du médium de communication et suffisamment simple pour être implémenté efficacement. Pour ce faire, nous avons introduit le canal comme objet du langage. Un canal est un connecteur permettant le rendez-vous et l'échange unidirectionnel de message entre deux processus. Ses principales caractéristiques sont :

- un canal possède un nom ; les règles de visibilité de ce nom sont les même que celles des variables C.
- un canal possède des attributs. L'attribut principal d'un canal est le type des objets C qu'il peut transmettre. Cet attribut est utilisé par le compilateur pour vérifier la cohérence des opérateurs sur les canaux.
- le canal est un objet abstrait dont la structure interne est inaccessible au programmeur.
- la manière d'implémenter les canaux n'est pas figée par le compilateur mais relève de l'environnement d'éxécution. Elle n'est pas unique : un canal C peut être un des canaux internes du Transputer, un canal virtuel (distribué) géré par le noyau de communication, une liaison ethernet... en fait n'importe quel médium de communication ayant les propriétés de "fiabilité" et de "transfert en un temps fini".
- comme tous les objet C, un canal peut être alloué dynamiquement. Cette dernière caractéristique s'avère très importante dans certaines applications mais restreint le nombre de vérifications que peut effectuer le compilateur.

La syntaxe de déclaration de canaux est la suivante :

```
[attribut] chan [type_c] identificateur_canal~;
```

déclare un canal `identificateur_canal` qui permet de transmettre des objets C de type `type_c`. La signification de **`type_c`** et d'attribut est la suivante :

- En l'absence de spécification de type, le canal transmet des valeurs de type int. Un canal est un objet qui ne peut être affecté ni lu (donc copié) car il n'a pas de valeur : on ne peut que lui appliquer les opérateurs spécifiques de communication. Tous les types C permettant de déclarer une variable sont acceptés. Notons que le mot clé **chan** se comporte de manière syntaxique comme les attributs de type standard. Ceci implique l'utilisation préalable de **typedef** lors de la déclaration de canaux de types construits comme le montrent les exemples qui suivent. Le

type `void` permet de déclarer des canaux spéciaux qui transmettent des signaux sans valeur : ceci permet d'écrire des synchronisations par rendez-vous sans avoir besoin de transmettre une valeur.

- attribut (qui est optionnel) permet de limiter le type des références que l'on peut faire à `identificateur_canal`. Cet attribut peut prendre deux valeurs (exclusives) : `const` ou `volatile`. Pour ne pas alourdir la syntaxe, nous avons choisi de réutiliser les mots clés des attributs introduits par la nouvelle norme ANSI en étendant leur usage aux canaux.

La syntaxe exacte est décrite en annexe. Les exemples suivants illustrent différentes déclaration possibles :

```
chan Canal ;
```

déclare un canal pour transfert d'entier.

```
extern const chan char Canal ;
```

déclare un d'un canal transmettant des caractères. L'opération démission ne peut suivre cette déclaration.

```
chan int Tab_Canal [ 5 ] ;
```

déclare un tableau de 5 canaux transmettant des entiers.

```
typedef int Tab [ 5 ] ; chan Tab Canal ;
```

déclare un canal transmettant des tableaux de 5 entiers.

```
chan signed short * Canal ;
```

déclare un pointeur sur un canal transmettant des entiers courts signés.

```
typedef unsigned long * Pointeur ;
chan Pointeur Canal ;
```

déclare un canal transmettant des pointeurs sur des entiers longs non signés.

Remarque: Ceci n'a de sens que si les processus qui s'échangent des pointeurs partagent la mémoire contenant les éléments pointés.

Les canaux tels que nous les avons définis permettent donc de transmettre des objets C structurés et même d'inclure des canaux dans des types structurés (qui ne peuvent pas être transférés eux-même par des canaux pour cette raison). Ces caractéristiques permettent une grande souplesse de programmation. Toutefois l'usage des canaux de C// est, sur un autre point, plus restrictif que ce qui existe en OCCAM 2.

Ce langage permet d'utiliser ce que son auteur (la société INMOS) appelle "Protocole". Il s'agit en fait de protocoles extrêmement simples puisque basés sur la notion de rendez-vous. OCCAM permet la définition un langage permettant de décrire la structure du flot de valeurs transférées après établissement du rendez-vous. Un tel langage est défini par une grammaire dont les éléments terminaux sont les types de base du langage OCCAM. Cette notion de grammaire de langage n'est pas présente clairement dans l'implémentation actuelle d'OCCAM. N'y sont présentes que les notions de sérialisation et de "protocole variant" (une marque permet d'aiguiller de manière univoque le processus récepteur dans l'arbre des possibilités pour reconnaître le flot reçu).

On pourrait imaginer de généraliser ce principe : à chaque canal serait attaché non plus un type mais une grammaire hors contexte permettant de définir le langage L associé au flot de valeurs transitant par ce canal. Le processus récepteur comporterait alors un analyseur de cette grammaire (outil du type yacc) pour permettre la réception (à l'exécution) et le processus émetteur utiliserait une séquence d'envoi de valeurs compatibles avec le langage L. Le compilateur devrait pouvoir vérifier que les éléments de mémorisations utilisés en réception sont compatibles avec L ce qui pourrait poser un problème d'allocation dynamique de ressources mémoire en cas de règle grammaticale récursive et d'impossibilité de connaître dès la compilation la séquence émise. L'étude de tels mécanismes et de la classe de grammaire intéressante dépasse le cadre de ce document, aussi nous ne préciseront pas plus cet aspect des canaux.

Dans C// nous n'avons pas voulu éliminer la possibilité d'utiliser un même canal pour transmettre des objets hétérogènes. La notion de séquence hétérogène d'objet tel quelle existe en OCCAM 2 (rendue nécessaire par l'absence de structure dans ce langage) peut être assimilée au transfert de type structure C.

Pour ce qui est des "protocoles variants" d'OCCAM 2, le transfert d'union d'agrégat peut être utilisé mais il n'est pas optimal dans la mesure où la taille transférée correspond toujours au cas de l'objet le plus volumineux. De plus cela oblige les processus émetteur et récepteur à réserver cet espace maximal avec le problème suivant : il n'est pas possible d'exprimer (en C) le choix d'une valeur correspondant à un cas quelconque de l'union. Une autre solution à ce problème consiste à déclarer une union de canaux transférant des types hétérogènes comme le montre l'exemple suivant :

```
union {
  chan int canal_entier;
  chan char canal_octet;
} canal_heterogene;
```

Comme il a été précisé plus haut, un canal ne peut pas être copié ; il ne peut qu'être référencé. Par conséquent, le passage d'un canal en paramètre d'une fonction se passe de la même manière que pour un tableau : passage par adresse. Exemple :

```
/* prototype  de la fonction: */
void f ( chan int c ) ;

/* definition de la fonction: */
void f ( c )
chan int c ; /* c est une reference */
{
  chan int d ;     /* d est un canal */
  ...
}

/* appel de la fonction: */
chan c_global ;
...
f(c_global); /* operateur & implicite */
```

Remarque : il est illégal de passer en paramètre un type complexe contenant un canal (le compilateur actuel ne le vérifie pas) ou alors il faudrait convenir que dans ce cas le passage se ferait systématiquement par référence.

Opérateurs sur les canaux

En ce qui concerne la manipulation de ces objets, elle se fait par l'intermédiaire d'opérateurs prédéfinis. Nous avons choisi d'utiliser une syntaxe fonctionnelle (similaire à l'appel d'une fonction) pour des raisons de clarté ; l'opération n'est pas traduite en un appel de routine mais expansé en ligne partout où cela est possible.

Les opérateurs sur les canaux sont de trois types : opérateurs de réception, opérateurs d'émission, opérateurs de contrôle. Ces opérateurs peuvent être vus comme des fonctions génériques : il s'applique à des canaux de type différents. Voici les "prototypes génériques" de ces opérateurs ainsi que leur sémantique :

```
<Type_item> in ( chan <Type_item> Canal ) ;
```

permet de recevoir une valeur de type `Type_item` par le canal `Canal`. La syntaxe fonctionnelle de l'opérateur `in` permet son utilisation comme opérande dans une expression C. Cette dernière possibilité doit être utilisée avec précaution car elle implique une bonne connaissance de l'ordre d'évaluation des expressions. Le compilateur vérifie que le contexte de l'expression est compatible avec le type du canal. On ne peut pas utiliser `in` pour un canal ayant l'attribut `volatile`. On ne doit pas utiliser cet opérateur dans un contexte d'instruction (in doit trouver un tampon pour y stocker la valeur reçue).

```
void out ( chan <Type_item> Canal , <Type_item> Valeur ) ;
```

permet d'emettre une valeur **Valeur** de type **Type_item** par le canal **Canal**. Le compilateur vérifie que le le type de la valeur est compatible avec celui du canal. On ne peut pas utiliser out pour un canal ayant l'attribut **const**.

```
void out ( chan void Canal ) ;
```

permet d'emettre un signal par le canal **Canal** qui doit être de type **void**. On ne peut pas utiliser cette forme pour un canal ayant l'attribut **const**.

```
void in ( chan void Canal );
```

permet de recevoir un signal par le canal **Canal** qui doit être de type **void**. On ne peut pas utiliser cette forme pour un canal ayant l'attribut **volatile**.

```
void inblock ( chan <Type_item> Canal ,
               <Type_item> Adresse_items ,
               unsigned int Nb_items
             ) ;
```

permet de recevoir une série de **Nb_items** valeurs de type **Type_item** par le canal **Canal** et de la stocker dans le tampon adressé par **Adresse_items**. L'ordre des adresses reflète l'ordre de réception des items. Le compilateur vérifie que le type pointé par **Adresse_items** est le même que celui du canal. On ne peut pas utiliser **inblock** pour un canal ayant l'attribut **volatile**.

```
void outblock( chan <Type_item> Canal ,
               <Type_item> Adresse_items ,
               unsigned int Nb_items
             ) ;
```

permet d'emettre une série de **Nb_items** items pointée par **Adresse_items** sur le canal **Canal**. L'ordre des adresses reflète l'ordre de d'émission des items. Le compilateur vérifie que le type pointé par **Adresse_items** est le même que celui du canal (T). On ne peut pas utiliser **outblock** pour un canal ayant l'attribut **const**.

```
<c_process> resetchan ( chan <Type> Canal ) ;
```

permet d'initialiser ou de réinitialiser un canal à la valeur **noprocess** (pas de processus au rendez-vous). Dans le cas d'une réinitialisation **resetchan** renvoi l'identificateur du processus qui était déjà au rendez-vous avant réinitialisation, ou la valeur **noprocess** dans le cas ou le processus interlocuteur n'était pas encore au rendez-vous. Dans le cas d'une première initialisation **resetchan** ne renvoie aucune valeur significative.

Gestion du temps - Horloges

Les horloges sont des dispositifs abstraits (physiques ou logiques) sur lesquels s'appliquent les opérateurs suivants :

after (clock horloge , heure) attente d'une certaine heure

delay (clock horloge , délai) attente d'un délai

unsigned int clock_resol (clock horloge) ; retourne la résolution de l'horloge en micro-secondes.

B.1.2 L'alternative

L'alternative est un constructeur du langage C// qui permet d'introduire une forme de non déterminisme dans le langage. Cette construction correspond à la notion d'alternative définie dans CSP. L'alternative de C// permet de gérer les événements de types suivants :

1. canal prêt à recevoir
2. événement de l'échéancier
3. condition simple (expression C)
4. combinaison des événements 1 et 3 ou de 2 et 3

Notons qu'il n'est pas possible d'effectuer une alternative sur un événement d'émission : nous avons choisi d'éliminer ce cas car l'aspect dynamique du langage C ne permet pas au compilateur de déterminer (statiquement) si un même canal apparaît simultanément dans deux alternatives. Ce cas implique l'élection de deux gardes (une garde pour l'alternative qui teste l'émission et une garde pour l'alternative qui teste la réception). Ce mécanisme est très lourd à mettre en oeuvre, même sur les Transputers ; nous avons pris la même restriction que le langage Occam de manière à ne pas payer un coût trop élevé dans les cas d'alternative simple (qui sont beaucoup plus fréquentes).

Les sections qui suivent présentent la sémantique et la syntaxe du constructeur **alt** ainsi que quelques exemples d'utilisation.

Sémantique

Une instruction `alt` est constituée par un ensemble de gardes et d'alternatives :

- une garde est un couple de conditions qui doivent être satisfaites pour que l'alternative associée soit executée,
- une alternative est une suite d'instructions C directement liée à une ou plusieurs gardes.

L'instruction `alt` correspond au transfert du contrôle d'exécution vers l'une des alternatives qui composent le `alt`, alternative qui est choisie en fonction de la valeur des gardes de celui-ci. Pour être choisie, une garde doit :

- Satisfaire les deux éléments qui la composent : une expression booléenne (ou condition) et un rendez-vous sur un canal ou horloge.
- Avoir la plus haute priorité parmi les gardes dont la condition et le rendez-vous ont été satisfaits. Cette notion de priorité est précisée un peu plus loin.

Il faut indiquer que l'évaluation des gardes est faite en les considérant sur un même pied d'égalité. C'est uniquement après l'évaluation de toutes les conditions et la mise en attente d'un rendez-vous sur tous les canaux et horloges dont la condition est vraie, que nous faisons intervenir la notion de priorité. Ce mécanisme de choix d'une garde sera appelé dans ce qui suit, le mécanisme d'élection d'une alternative.

L'élection d'une alternative se fait de la façon suivante :

1. On évalue les conditions de toutes les gardes qui constituent le alt. Les gardes sans condition sont considérées comme ayant une condition toujours vraie ; les gardes sans rendez-vous sont par ailleurs considérées comme ayant déjà eu ce rendez-vous et la garde `default` comme satisfaisant les deux contraintes (condition et rendez-vous). Les gardes itératives sont dépliées en un ensemble de gardes simples ; pour ce faire, nous effectuons l'itération spécifiée par le constructeur for (sémantique similaire à celle de l'instruction C for) et déterminons la garde correspondant à chaque pas de l'itération. A chaque garde sera associée la valeur de l'inducteur à ce moment-là, ce qui permettra à l'utilisateur d'identifier l'alternative élue.
2. Tous les canaux et horloges désignés par les gardes dont la condition est vraie, sont activés en attente de leur rendez-vous. Ces objets seront désignés dans ce qui suit par le terme dispositifs activés
3. Dès qu'un rendez-vous est obtenu (plusieurs peuvent l'être en même temps), nous désactivons tous les dispositifs activés.
4. On parcourt alors l'ensemble de dispositifs activés et on s'arrête sur le premier des dispositifs dont nous constatons le rendez-vous. L'ordre dans lequel est effectué le parcours détermine la priorité des alternatives. Dans notre cas, cet ordre est celui de l'écriture des alternatives de l'instruction alt : la première alternative a la priorité la plus forte et la dernière la moins forte.
5. L'alternative élue est celle dont la garde correspond au dispositif choisi.

Il serait possible de demander au compilateur d'effectuer un choix non-déterministe (ou d'autres choix, qui sont à discuter), grâce à un pragma du langage. Il faut cependant indiquer qu'un tel mécanisme ne pourrait être implanté qu'à un coût sensiblement supérieur.

Une fois effectuée l'élection, le contrôle est transféré au début de l'alternative choisie. Si la garde de celle-ci est de type itératif, la valeur de l'inducteur, correspondant au pas d'itération qui a donné lieu à l'élection, est restituée préalablement au transfert.

Les instructions de l'alternative sont exécutées séquentiellement. Le comportement est identique à celui d'un switch C après choix de l'étiquette (case ou default) : l'exécution d'une alternative continue sur l'alternative suivante (les alternatives sont compilées les unes après les autres), si aucune instruction de branchement (**break**, **goto** ou éventuellement `continue`) n'est rencontrée.

remarque : L'inducteur d'une garde itérative n'est réinitialisé que si l'une de ces instances est choisie ; dans tous les autres cas, il contient la dernière valeur qui lui a été affectée (en général, celle qui a provoqué la sortie de la boucle de la garde itérative)

Exemples

L'exemple qui suit illustre l'utilisation de l'instruction alt :

```
alt {
  guard c1 :
    y = in( c1 ) + 1 ;
    z = 2 * y ;
    break ;
  guard c2 :
    out( c2 , message ) ;
    break ;
  guard nb > 5 , c3 :
  guard c4 :
    out( sortie, t++ ) ;
    break ;
  default :
    nobody = 1 ;
}
```

Les gardes itératives peuvent se révéler très utiles quand on désire exprimer un rendez-vous sur des canaux calculés dynamiquement : tables de canaux ou horloges, listes de ces dispositifs, etc. Les exemples qui suivent illustrent l'utilisation du alt dans ce cas :

```
alt {
  guard attente[i] , canal[i]
    for ( i = 0 ; i < 10 ; i++ ) :
      recu[i] = in( canal[i] ) ;
      attente[i] = TRUE ;
```

```
}
```

```
alt {
  guard c->actif, c->canal
    for ( c = tete; c != NULL; c = c->suivant ) :
      message = in( c->canal );
      recu = TRUE;
      break;
  default :
    recu = FALSE;
}
```

Il faut enfin indiquer que l'instruction alt peut être utilisée pour écrire une forme de switch dynamique, similaire au cond du langage Lisp :

```
alt {
  guard i < 0 :
    printf("Indice negatif ... \n");
    break;
  guard default :
    exit( -1 );
  guard i = 0 :
    fin = TRUE;
  guard i > 0 && i < 256 :
    out( sortie, i );
    break;
}
```

On peut observer qu'aucune garde ne fait appel à un rendez-vous sur canal ou horloge ; il s'agit en effet d'une utilisation détournée du alt mais parfaitement licite dans notre langage.

Syntaxe de la construction ALT

La syntaxe suivante est un extrait de la grammaire de C//.

```
/* Definition de l'instruction alt */
```

```
instruction_alt :
    alt corps_alt ;

corps_alt :
    instr_gardee
  |
    { liste_instr_gardees } ;

liste_instr_gardees :
    instr_gardee
  |
    liste_instr_gardees instr_gardee ;

instr_gardee :
    garde : liste_instrC_opt  ;

liste_instrC_opt :
    /* vide */
  |
    statement /* instr. C classiques */ ;


/* Definition des gardes. */


garde :
    garde_simple
  |
    garde_iterative
  |
    default ;

garde_simple :
    guard corps_garde_simple ;

corps_garde_simple :
    condition
  |
    condition_opt canal_ou_horloge ;

canal_ou_horloge :
    expression /* expr. de type canal ou horloge */ ;

condition_opt :
```

```
    /* vide */
  |
    condition , ;

condition :
    expr_affect /* sans op. virgule */ ;

garde_iterative :
    guard corps_garde_simple
    for( inducteur;expression_opt;expression_opt) ;

inducteur :
    expr_unaire
  |
    expr_unaire op_affectation expr_affect ;
```

Contraintes contextuelles

La construction alt doit respecter un certain nombre de contraintes contextuelles :

- Les conditions des gardes doivent être de type scalaire. Ces conditions sont des expressions d'une catégorie syntaxique dans laquelle on ne peut pas faire usage de l'opérateur virgule (à moins de parenthéser). Nous avons voulu empêcher par ce moyen des conflits avec la virgule qui sépare la condition du canal/horloge d'une garde. Il faut bien remarquer que les conditions ne sont pas nécessairement des affectations, en dépit de leur nom : il s'agit d'une catégorie de la grammaire C qui permet de dériver tout autre type d'expression, à l'exception bien entendu des expressions virgule.

- La deuxième partie d'une garde est une expression qui doit avoir le type soit canal, soit horloge. Une expression horloge est nécessairement constituée par une opération after ou une opération delay.

- Aucun canal ou horloge ne doit apparaître dans deux gardes d'un même alt. Une telle utilisation peut provoquer des incohérences sémantiques : plusieurs alternatives valides pour un même canal, plusieurs réveils pour une même horloge. Ces incohérences n'affecteront en rien l'exécution Anormale du programme mais peuvent donner des résultats inattendus.

- Une garde peut ne pas être suivie d'instructions, ce qui permet de partager ces instructions avec d'autres gardes (à la manière des instructions switch de C).

- L'inducteur d'une garde itérative est un objet qui sera employé pour effectuer l'itération. Cet objet est désigné par l'expression unaire et doit être une variable

modifiable. Il faut noter que l'on ne peut pas utiliser l'opérateur virgule dans la partie inducteur et que, par conséquent, on ne peut exprimer au plus qu'une initialisation d'une variable d'induction.

bilan caractéristiques de l'alternative en C//

Les principales caractéristiques de l'alternative en C// sont :

- l'alternative est un constructeur du langage. Le générateur de code ne fait aucune hypothèse sur l'implémentation des canaux : l'alternative peut donc utiliser des canaux distribués et un temps "logique" (échéancier géré par logiciel).

- aspect dynamique : le nombre de gardes d'une même alternative n'est déterminé qu'à l'exécution ; il peut donc varier d'une exécution à l'autre (ce qui n'est pas possible en Occam).

- Il y a une séparation très nette entre l'opération de construction de l'ensemble des gardes éligibles et l'élection proprement dite : ceci permet d'appliquer différentes politiques d'élection d'une garde sans avoir à modifier le générateur de code. En fait il suffit que l'utilisateur effectue une édition de lien avec sa propre routine d'élection : on peut ainsi introduire l'utilisation d'un générateur de nombre aléatoire par exemple.

- contrairement à Occam l'élection d'un événement de type canal ne provoque pas le réveil du processus élu. Ce réveil est explicite (utilisation de l'opérateur in dans le processus gardé et non dans la garde). Cela apporte beaucoup plus de souplesse dans la programmation en particulier dans les applications systèmes pour l'écriture de serveurs.

- les gardes ne sont évaluées qu'une seule fois ce qui est compatible avec les "effets de bord" dans l'évaluation de la garde.

B.2 Les processus

Cette section décrit le support à l'exécution du langage C// ainsi qu'une partie des bibliothèques.

B.2.1 Quel type de processus ?

Nous avons choisi d'offrir deux niveaux bien distincts
qui correspondent à deux grains de parallélisme très différents :

Parallélisme à gros grain : La nature des objets utilisés par ce type de processus est telle que leurs coûts d'accès est important. Nous avons choisi le concept de tâche pour ce type de processus. Les mécanismes de synchronisation et de communication entre tâches sont coûteux car gérés globalement par le système. Les objets gérés par ces processus sont de taille importante (fichiers, segments de mémoire). Ce sont tous des objets du niveau système (ressources globales). Ce grain important est sans doute le plus intéressant pour exploiter le parallélisme réel compte-tenu des possibilités machines actuelles.

Parallélisme a grain fin : Par opposition, on regroupe sous le terme de processus légers toutes les activités parallèles qui ont des durées faibles et qui agissent sur des objets beaucoup plus simples. Pour ce type de processus on ne peut pas avoir les même coûts de gestion que pour des processus système. A ce niveau, il est possible de faire partager par les processus une partie de leur mémoire et d'utiliser cela pour effectuer des communications très efficacement (la gestion de ces transferts est assurée à la compilation du langage de programmation). Ce grain fin est tout a fait indiqué pour la multi-programmation d'un monoprocesseur (partage de mémoire entre les processus légers).

Les abstractions proposées par notre système et décrites ci-après sont basées sur cette classification. Elle sont fortement liées au problème de granularité (degré de parallélisme versus coût de communication).

B.2.2 Les tâches

Le développement de notre environnement système a été réalisé dans le but de supporter notre système Prolog parallèle et différents outils nécessaires à son développement. Nous n'avons pas eu le temps d'étudier tous les problèmes systèmes et de définir un modèle de tâche très élaboré. Nous nous sommes contentés de réaliser le "minimum vital". Cette section donne un aperçu des fonctionnalités offertes par notre environnement.

Notion de tâche

Une tâches représente une unité d'exécution qui possède un nom que gère le système. Une tâche est un module exécutable. Elle est chargée en mémoire pour exécution. Elle ne peut ni migrer dans le réseau ni migrer dans la mémoire ou sur disque (pas de swapping). Ces restrictions sont dues essentiellement à l'absence de mécanisme de translation d'adresse sur les transputers actuels.

L'impossibilité de migration pourrait être relaxée par une modification de notre générateur de code générant des données (pointeurs) relogeables. Le choix du modèle d'exécution et de la génération de code a été fait de manière à faciliter cette modification.

Dans la version actuelle, une tâche ne peut pas être répartie sur plusieurs processeur. Une solution est d'offrir un mécanisme de regroupement de tâches élémentaires en tâche parallèle (cf Task Force sous Helios [Garn87]).

Création - destruction

Une tâche peut être créée par une autre tâche à l'aide des primitives de type **exec**. Dans la version actuelle, ces primitives sont partiellement implémentées (contrairement à UNIX, les primitives exec empilent le contexte de la tâche créatrice et activent la tâche créée qui rend le contrôle à la tâche créatrice lors de sa disparition). Les primitives spawn permettent de créer une tâche qui est activée en parallèle de la tâche créatrice. La primitive `exit` permet à une tâche de terminer.

B.2.3 Les processus légers ("Thread").

La notion de processus légers de type "thread" a été introduite dans notre système pour permettre une exploitation efficace de la programmation concurrente au sein d'une tâche. La notion de tâche telle quel définie précédemment implique un important coût de communication inter-tâches car les tâches ne partagent pas d'espace de mémoire à faible coût d'accès. D'autre-part, la gestion des tâches est effectuée par le système (de part les ressources accessibles par une tâche) et coûte cher. Le coût de la commutation de contexte entre deux tâche est en général très important, ce qui en rend trop chère, l'utilisation dans un bon nombre d'applications.

Il est nécessaire de disposer de processus peu coûteux en création/destruction qui partagent de la mémoire afin d'éviter le transfert de données entre processus et permettre des synchronisations très peu coûteuses. De plus, il est rarement nécessaire que ces processus utilisent des ressources du niveaux système: dans la grande majorité des cas le rôle de ces processus est de permettre une écriture plus simple de l'enchaînement d'actions qui serait difficile (et moins naturel) d'exprimer sous forme d'un flot de contrôle unique (création d'un automate d'état fini complexe).

Ainsi une nouvelle organisation d'exécution est bâtie sur la dualité Tâche / "Thread". Une tâche représente l'ensemble des ressources nécessaire à l'exécution d'un programme. C'est en particulier la ressource mémoire et son organisation. La tâche est le site d'exécution d'un nombre quelconque de "threads". La tâche classique devient dans ce schéma une tâche munie d'une seule "thread".

Notion de "thread"

La gestion de ces processus ne fait pas intervenir le système et est effectuée de manière interne à la tâche. Dans notre cas, les "threads" sont des processus autonomes qui se partagent de la mémoire (les variables globales de C) et ont une partie privée (pile des environnements C). Ces processus partagent l'interface système de la tâche à laquelle ils appartiennent.

Pour obtenir une meilleure efficacité, le contexte du processus est "allégé": une commutation de contexte entre deux processus légers d'une même collection se borne alors à sauvegarder une partie très réduite de ce contexte (en général, le compteur ordinal et le sommet de pile).

L'implémentation de ce modèle sur le Transputer s'appuie sur le noyau de processus de la machine et nous permet ainsi:

- d'exploiter au mieux les possibilités de la machine,
- de simplifier la réalisation en raison de la similitude entre processus Transputer et processus légers)
- d'atteindre une plus grande efficacité.

Les mécanismes simples et efficaces se bornent à :

- la manipulation de processus (création, destruction, attente de la fin d'un processus), et
- la synchronisation (par rendez-vous, exclusion mutuelle, sémaphores...)

Ils sont proposés en tant que fonctions d'une bibliothèque C et ne changent en rien les programmes qui ne font pas usage de cette bibliothèque.

Tous les processus légers d'une même collection s'exécutent sur le même processeur, pour des raisons simplificatrices. Ils peuvent alors partager sans problème la mémoire correspondant :

- au code du programme,
- aux données globales, et
- au tas (allocation dynamique de mémoire).

Chaque processus léger a sa propre pile.

Le contrôle de l'accès à la mémoire partagée reste sous la responsabilité de l'utilisateur : si deux processus légers utilisent une même variable ou un groupe de variables, les instructions qui y accèdent en exclusion mutuelle. La communication entre processus légers peut alors se faire au moyen de variables communes.

Les "thread" sont donc utilisées pour la multi-programmation fine du Transputer (concurrence d'accès au CPU) alors que les tâches peuvent être utilisées pour exploiter le parallélisme réel des machines multi-processeurs. Ces deux aspects sont complémentaires dans la mesure où une commutation de contexte peu coûteuse permet un grain de parallélisme plus fin et, par la même, une possibilité de compenser les temps d'attentes dûs aux communications et aux relations d'inter-dépendances entres processus distants (recouvrement du temps de communication par l'évaluation d'une ou plusieurs autres "thread"). La répartition temporelle du processeur entre les "thread" des tâches allouées dans un même espace d'adressage physique se fait directement par le "scheduler" du Transputer ; de ce fait, l'implémentation s'accommode des contraintes de celui-ci.

Identification d'une "thread"

Dans la plupart des cas on a simplement besoin d'identifier (pour différentier) les "thread" d'une même tâche ce qui implique une désignation locale à une tâche par contre cette désignation locale doit être complétée lorsque l'on veut différencier les "thread" de plusieurs tâches.

Une thread possède donc une identification unique tant locale que globale. L'identification globale d'une "thread" est rendue nécessaire par la possibilité offerte à chaque "thread" d'accéder en parallèle aux ressources systèmes (accessibles via des serveurs: serveur de fichiers, serveur de fenêtres, ...) depuis n'importe quel noeud (processeur) du réseau. Pour distinguer deux threads sur des processeurs différents, on complète l'identification locale de la thread par l'identification (univoque) du noeud dans le réseau

Compte tenu des spécifications des "thread" et de leur grand nombre, la gestion des identificateurs de "thread" doit être la moins coûteuse possible et être compatible avec les possibilités offertes par le transputer pour en tirer le maximum de performances. Une solution possible est d'utiliser l'adresse du descripteur de "thread" en mémoire (bloc dans lequel est sauvé/restauré l'état du processeur. Ce choix a largement influencé tout notre modèle d'exécution ainsi que le choix "original" de génération de code de notre compilateur C//. Les processus gérés par le microcode du transputer sont identifiés (localement à un processeur) par l'adresse (physique) de leur espace de travail. Le (seul) registre de base du transputer: W (Workspace pointer) désigne la base des données et le registres I l'instruction en cours Le couple (W,I) constitue le seul contexte à changer d'où les performances du "scheduler" micro-codé du Transputer (commutation de contexte de l'ordre de la micro-seconde!...). Tout le jeu d'instruction du transputer est influencé par ces caractéristiques; en particulier, les canaux internes sont implémentés par un seul mot mémoire contenant l'identificateur (W) du processus qui arrive le premier au rendez-vous, les files (une par priorité de processus) des processus activables ainsi que celles des échéanciers utilisent cet identificateur. Par conséquent il est très intéressant de garder une identification de "thread" compatible avec celle utilisée par toutes les mécanismes micro-codés du transputer.

A un instant donné, l'adresse de l'espace de travail d'une "thread" l'identifie (de manière univoque) localement au transputer sur lequel réside la tâche. Cet identificateur permet de différencier les "thread" de deux tâches différentes résidentes sur le même processeur à un instant donné. Le choix d'une adresse physique comme identificateur implique une dépendance vis a vis de l'allocation mémoire: une "thread" ne peut pas être déplacée après avoir été allouée, cela limite également la possibilité de migration de processus.

L'unicité d'identification dans le temps d'une "thread" nécessite d'adjonction de sa date (locale au processeur) de création de manière à pouvoir différencier deux "thread" occupant successivement la même place en mémoire. Ces deux processus ne peuvent évidemment pas coexister puis qu'il n'y a pas de chevauchement de leurs durées de vie toutefois des références à cette identification peuvent subsister après la mort du processus; c'est pourquoi la date de création est indispensable. C'est le cas par exemple

d'objets persistants crées par un processus, ou des messages en transit dans le noyau de communication. En pratique la date n'est vraiment utile que lors du traitement d'exceptions (et suppression d'un processus) et pour la mise au point.

En résumé, la complexité de l'identificateur de "thread" varie selon le contexte de sa référence. On peut décrire cette identification de la manière suivante :

```
Identification_Temporelle :
    Identification_Globale Date_Creation

Identification_Globale :
    Identification_Noeud Identification_Locale

Identification_Noeud :
    Numero_Processeur

Identification_Locale :
    Adresse_Espace_Travail
```

Opérateur de composition parallèle - grappe de process

Un programme parallèle C// décrit un réseau de thread interconnectés par des canaux. La description d'un tel réseau doit être la moins restrictive possible et doit prendre en compte l'aspect dynamique (réseau évoluant à l'exécution). Le langage OCCAM ne permet pas d'offrir un mécanisme de description de réseau indépendant de l'activation des processus du réseau. Nous avons choisi de dissocier ces deux mécanismes.

L'instanciation d'un réseau de processus nécessite:

1. de créer des instances de processus identifiables mais non actifs
2. d'utiliser ces identificateurs pour décrire la relation d'interconnexion des processus
3. d'activer l'ensemble des processus ainsi connectés.

Le schéma précédent introduit la notion de grappe de processus crées dynamiquement, activés en même temps et terminant lorsque l'ensemble des processus qui la composent a terminé leur exécution. On retrouve alors la notion de composition parallèle d'OCCAM avec une différence importante : l'aspect dynamique qui confère à notre outil une très grande souplesse d'utilisation. La contrepartie de l'aspect dynamique de l'opérateur de composition parallèle est la limitation des possibilités de vérifications à la compilation. Actuellement cet opérateur est implanté par des bibliothèques et n'est pas intégré au langage.

L'opération de composition parallèle pose le problème de la relation entre processus créateur (exécutant l'opération) et les processus crées. Cette relation peut être synchrone, c'est à dire que le créateur attend la fin de l'exécution des processus crées ou asynchrone : dans ce cas il n'y a pas de contraintes de synchronisation entre processus créateur et processus crées.

filiation synchrone : Cette relation de filiation des processus légers peut être représentée par un arbre dont les noeuds sont les "thread"

Cette relation permet de définir des arbres de processus (noeuds = fils directs d'un même père). Elle peut être utilisé comme mécanisme d'abstraction (pour masquer la structure interne de la grappe). Elle permet également de préciser le contrôle du réseau en améliorant notablement la clarté des programmes. L'imbriquation des durées de vie qui correspond à la projection de l'arbre de filiation est compatible avec la notion d'abstraction (regroupement temporel des processus) : la durée de vie du processus père englobe celle des fils, le père étant inactif pendant l'exécution de ses fils (comme pour le constructeur PAR d'OCCAM).

opérateurs : Une primitive de création d'une instance de processus identifiable (sans son activation) est nécessaire. La primitive `p_new` réalise cette fonction. Sa description exacte ainsi que celles de toutes les primitives de gestion des processus légers, seront précisées dans les sections suivantes.

La définition de la grappe de processus est assurée par un mécanisme de "parenthèsage" des créations d'instance de processus. Ces créations étant dynamiques (impossible à déterminer dès la compilation), nous n'avons pas jugé bon de surcharger la grammaire du langage C// par un constructeur assurant ce parenthèsage puisqu'il n'aurait permis aucun contrôle sémantique. Le parenthèsage est d'ailleurs un parenthèsage relatif à la trace temporelle d'un processus et non à la structure linéaire du texte du programme (parenthèsage syntaxique). Ce mécanisme est donc assuré par deux primitives de la bibliothèque parallèle : `p_begin_par` et `p_end_par` qui définissent respectivement le début de construction de la grappe des fils et la fin de cette construction. Ces deux primitives sont exécutées par le processus père. Tous les `p_new` exécutés (par le père) entre le `p_begin_par` et le `p_end_par` correspondent à une instance d'un fils. Lors de l'appel de `p_end_par`, le processus père active tous ses fils puis se bloque. Il sera reveillé (reprise en séquence après `p_end_par`) par le dernier de ses fils à terminer. Signalons une contrainte évidente : il ne doit pas y avoir de `p_begin_par` sans `p_end_par`, de plus il n'est pas possible de les imbriquer dans un même processus. La "syntaxe" des traces temporelles légales d'un processus est la suivante :

```
trace_processus :
    trace_vide
  |
    element_trace trace_processus
  ;
```

```
element_trace :
    instruction_C_normale
  |
    p_begin_par creation_grappe p_end_par
  ;

creation_grappe :
    /* vide */
  |
    element_creation creation_grappe
  ;

element_creation :
    instruction_C_normale
  |
    p_new
  ;
```

La syntaxe précédente doit être respectée. Aucune vérification n'étant possible à la compilation le bon fonctionnement du système dépend du respect de ces règles par l'utilisateur. La version actuelle n'effectue aucune vérification à l'exécution car nous avons jugé cela trop coûteux.

filiation asynchrone : Il arrive que l'on veuille créer un processus n'ayant aucune contrainte de durée de vie et d'activation vis a vis de son créateur. Ceci est contradictoire avec une création par **p_begin_par**, **p_end_par** mais pas avec la relation de filiation. En effet, deux fils d'un même père ont des durées de vie complètement indépendantes ; il suffit donc d'offrir un mécanisme permettant à un processus de créer un propre frère. Plus exactement, il suffit de permettre l'activation d'une instance de processus (créée par **p_new**) comme fils du père du créateur. Ceci est assuré par la primitive **p_spawn**.

En résumé, on dispose donc de deux opérations bien distinctes sur l'arbre définissant le contrôle d'exécution des processus :

1. création d'un fils

2. création d'un frère

Tous les noeuds internes de l'arbre sont des processus inactifs (en attente de terminaison du sous-arbre dont ils sont racines). Ce dernier point est important car il permet de charger le processus père de la libération de toutes les ressources allouées pour l'exécution de ses fils après que ceux-ci aient terminés toute activité.

Primitives de gestion des thread

On distingue la création d'une thread de son activation pour permettre la création dynamique (à l'exécution) de grappes de threads dont la structure peut être quelconque. Par exemple, supposons que l'on veuille définir un arbre de processus fonctionnant tous en parallèle tel que les noeuds de l'arbre sont des threads et tel que deux nœud voisins sont reliés par un canal. Cette grappe de processus peut être définie récursivement par la création des processus du réseau et des canaux les interconnectant.

Exemple:

```
PRIVATE PROCESS producteur ( chan int vers_racine )
{
  int i;
  for(i=1;i<100;i++)
    out( vers_racine , i );
}

PRIVATE PROCESS feuille ( chan int de_racine )
{
  for(;;)
    printf( "recu %d\n" , in( de_racine ) );
}

PRIVATE PROCESS noeud_interne ( chan int de_racine ,
                                chan int vers_gauche ,
                                chan int vers_droit
                              )
{
  if ( ... )
    out ( vers_droit , in( de_racine ) );
  else
    out ( vers_gauche , in( de_racine ) );
}

void creer_arbre( int profondeur , chan int de_racine )
{
  if { --profondeur == 0 }
  {
    p_new( ... , feuille ,
           sizeof(chan int) ,
           de_racine
         );
  }
```

```
  else
  {
    chan int vers_droit ;
    chan int vers_gauche;
    resetchan( vers_droit  );
    resetchan( vers_hauche );
    p_new( ... , noeud_interne ,
           3*sizeof(chan int) ,
           de_racine , vers_gauche , vers_droit
         );
    creer_arbre( profondeur , vers_gauche );
    creer_arbre( profondeur , vers_droite );
  }
}

chan int racine = noprocess;

main()
{
  ....
  p_begin_par();
    p_new( ... , producteur , sizeof(racine) , racine );
    creer_arbre( 10 , racine );
  p_end_par();

}
```

Il est nécessaire de pouvoir créer des processus sans les activer aussitôt car toutes les préconditions de l'activation ne sont pas valides dès l'allocation de ressources (création de la thread).

Les fonctions de création-destruction d'une thread sont :

p_new :

```
#include <process.h>
extern t_c_process
       p_new( size_t stack_size ,
              char * pad_name ,
              t_process_code process_code,
              size_t args_size ,
              ... /* args_values */
            );
```

p_new crée un nouveau processus sur le même processeur que le processus courant. p_new alloue un contexte pour le nouveau processus ainsi une pile de taille **stack_size** pour l'évaluation de la fonction **process_code**. p_new met également à jour les informations concernant les relations de filiation entre processus créateur et processus crée (arbre). Le processus ainsi crée n'est pas activé et sa priorité n'est pas encore fixée. La fonction p_new retourne l'identificateur de la thread créée ou NULL s'il n'y a plus assez de mémoire pour créer ce processus.

L'activation d'un tel processus pid = p_new(...) peut se faire soit explicitement, par p_run(pid) p_runh(pid) ou p_runl(pid), soit implicitement, lors d'un p_end_par() si le p_new est exécuté après un p_begin_par(). Les ressources allouées par le processus p lors de l'exécution pid = p_new(...) peuvent être explicitement libérées par le processus p qui doit, pour ce faire, exécuter p_free(pid) sauf dans le cas de p_end_par() qui libère implicitement ces ressources lorsque tous les processus activés en parallèle ont terminé.

p_free :

```
#include <process.h>
extern void p_free( t_c_process pid ) ;
```

p_end :

```
#include <process.h>
extern void p_end( void ) ;
```

p_end termine le processus courant. Si tous les frères de ce processus sont terminés alors p_end réveille le père. L'appel à p_end peut être omis car il est exécuté implicitement lors du retour de la fonction qui correspond au processus.

p_info :

```
#include <process.h>
extern t_thread_info * p_info( void ) ;
```

p_info() permet d'accéder aux informations caractérisant le processus C. Il est vivement conseillé de ne pas modifier ces informations.

p_myself:

```
#include <process.h>
extern t_c_process p_myself( void ) ;
```

p_myself() permet au processus courant de s'identifier de manière non ambiguë relativement aux autres processus qui existent (actif ou pas) au même instant sur le même processeur. L'adresse renvoyée par p_myself() est celle de son workspace (sans le bit de priorité).

p_myself_prio :

```
#include <process.h>
extern t_c_process p_myself_pri( void ) ;
```

`p_myself_pri()` permet au processus courant de s'identifier de manière non ambiguë relativement aux autres processus qui existent au même instant sur le même processeur. L'adresse renvoyée par `p_myself_pri()` est celle de son workspace avec le bit de priorité.

B.2.4 Communication/synchronisation inter-thread

La communication inter-thread peut être réalisée de deux manières :

par des canaux : Ce type de communication est similaire à celui rencontré dans le langage Occam. Il est traité au niveau langage ce qui permet les vérifications de types. Il est vivement recommandé d'utiliser ce type de communication en accord avec la contrainte d'absence de données communes entre les threads. Ceci garantit la validité du programme dans le cas d'une distribution des threads sur des processeurs sans mémoire commune. L'exemple suivant illustre ce mode de communication.

```
#include <process.h>
#include <stdio.h>
#include <chan.h>

chan int c = noprocess ;

PRIVATE PROCESS emetteur ( int max )
{
  register int i ;
  out( c , max ) ;
  for ( i = 0 ; i < max ; i++ )
    out( c , i ) ;
}

PRIVATE PROCESS recepteur ( void )
{
  register int i ;
  for ( i = in( c ) ; i > 0 ; i-- )
    printf( "%d\n" , in( c ) + 1 ) ;
}

PUBLIC main ()
{
  p_begin_par() ;
    p_new( 10000 , "send" , emetteur , sizeof(int) , 9 ) ;
```

```
    p_new( 10000 , "recv" , recepteur , 0 ) ;
    printf("avant le par\n");
  p_end_par() ;
  printf("apres le par\n");
}
```

par mémoire commune : Dans le cas où les messages sont de taille importante et où les "thread" sont sur le même processeur, on peut tirer parti de l'espace d'adressage commun pour éviter la copie des messages inhérentes à la solution des canaux. Les thread utilisent alors des moyens de synchronisation traditionnels tels que les sémaphores. Les primitives de synchronisation de notre système sont les suivantes :

```
void sem_init( t_semaphore * sem_ptr , int initial_count ) ;
```

Inititialise le compteur du sémaphore pointé par `sem_ptr` à la valeur positive ou nulle `initial_count`. Dans la plupart des cas on utilise `initial_count = 1` pour assurer l'exclusion mutuelle.

```
void sem_P( t_semaphore * sem_ptr ) ;
void wait( t_semaphore * sem_ptr ) ;
```

décrémente le compteur du sémaphore pointé par `sem_ptr` d'une. Le processus qui exécute `sem_P` est bloqué (mis en queue la file du sémaphore) si le compteur est négatif.

```
void sem_V( t_semaphore * sem_ptr ) ;
void signal( t_semaphore * sem_ptr ) ;
```

incrémente le compteur du sémaphore pointé par `sem_ptr`. Si ce compteur est négatif ou nul, le processus qui exécute `sem_V` active le processus qui est en tête dans la file des processus en attente.

B.2.5 implantation de C// gestion des ressources mémoires

Contrairement à Occam, le langage C permet un usage libéral des pointeurs et les bibliothèques standards de C offrent un ensemble de fonctions permettant de gérer un tas. Traditionnellement, on partage l'espace d'adressage d'un processus en deux zones : la pile des environnement et le tas dont le sommet croît vers le sommet de la pile d'évaluation.

Dans notre système nous avons choisi de permettre à chaque "thread" de pouvoir effectuer toutes les opérations autorisées par un C séquentiel y compris l'allocation dynamique de mémoire. Le Transputer ne possédant pas de mécanisme de gestion mémoire virtuelle, l'espace d'adressage possible se réduit à la mémoire physique présente sur le noeud. Cette limitation physique ne permet pas d'envisager raisonnablement une partition statique de la mémoire du noeud entre les différentes "thread" qui peuvent être très nombreuses ; cela entraînerait un trop grand nombre de petits espaces libres. Nous avons donc choisi de factoriser les espaces des tas de chaque "thread" en une seule zone commune. La solution retenue offre l'avantage de pouvoir partager l'espace libre pour les tas (l'union de tas est un tas) donc de permettre à un plus grand nombre de "thread" d'utiliser l'allocation dynamique de mémoire. Cette solution est compatible avec la notion de tâche vu comme une collection de ressources partagées par les "threads".

Le partage du tas par les "threads" nécessite une synchronisation à chaque allocation et libération (exclusion mutuelle). Cette synchronisation bien que réalisée par un simple sémaphore peut devenir trop coûteuse dans le cas d'allocation, libération fréquentes d'un grand nombre de petits blocs. Pour éviter ce type de dégradation nous avons organisé la gestion du tas d'une tâche en deux niveaux : au niveau de la "thread", on gère des petits blocs sans synchronisation (la liste des blocs est privée). Lorsqu'il n'y a plus de blocs privés on demande un nouveau gros bloc au tas commun.

Note : Le non-déterminisme dû au parallélisme interdit toute hypothèse sur la valeur du pointeur rendu par la fonction malloc ; cela signifie qu'un même malloc dans un même programme dupliqué sur deux noeuds distincts ne renvoie pas nécessairement la même valeur !!!

En ce qui concerne la gestion des piles d'évaluation, la solution de factorisation des tas ne diminue pas la taille mémoire nécessaire car la relation de filiation des "thread" nécessite un arbre de piles. Contrairement à Occam, l'arbre des appels n'est pas fini (à cause de la récursivité) et ne peut pas être déterminé statiquement (de part la récursivité et l'aspect dynamique de notre parenthèsage `p_begin_par()`, `p_end_par()`). Par conséquent, il n'est pas possible de résoudre le problème de l'allocation mémoire des piles d'évaluation dès la compilation.

Notre solution consiste à allouer dynamiquement une zone de mémoire, lors de la création d'une thread, pour y mettre la pile d'évaluation de ce nouveau processus. Cette zone est retirée du tas commun à toutes les thread. Cette solution comporte plusieurs inconvénients :

- il n'y a pas de relation entre les adresses des processus fils et celle du processus père comme en Occam. Cela aurait facilité le traitement des exceptions.
- il est nécessaire que l'utilisateur précise une taille de pile suffisante pour l'évaluation de la "thread" (premier paramètre de p_new) ce qui peut être assez difficile à déterminer précisément. Une sur-évaluation entraîne une surconsommation de mémoire sans possibilité simple de récupération (correction de cette borne en cas de saturation globale). Une sous-évaluation entraîne un débordement de pile dont la détection systématique coûte cher.

Nous avons cependant retenu cette solution car elle offre une efficacité optimale du point de vue durée de l'exécution séquentielle (gestion en pile classique). Dans le cas général il n'est pas possible de faire évaluer cette borne par le compilateur C à cause (entre autre) de la récursivité. Les cas où ce serait possible (arbre des appels finis) sont courants mais parfois difficiles à détecter (par exemple dans le cas de pointeur sur une fonction).

Une autre alternative beaucoup plus propre (mais moins efficace en temps) est possible : plutôt que d'essayer de calculer la taille de la pile d'évaluation, on peut remettre en cause cette structure de données ou plus exactement la relation de contiguïté existant traditionnellement entre le contexte de la fonction appelante et celui de la fonction appelée. En effet, si l'on considère que le contexte de chaque procédure est alloué dynamiquement, il n'est plus nécessaire de préciser (ou de calculer) la taille mémoire requise par l'évaluation d'un processus. On a alors un mécanisme d'allocation de l'environ appelé (en séquentiel) identique à celui d'allocation d'un environ de la nouvelle thread (en parallèle). Cette homogénéité n'offre pas que des avantages : on paie le même coût en séquentiel qu'en parallèle. Ce coût est celui d'une allocation dynamique (très chère) et de l'héritage des données de liaisons (en particulier des paramètres d'appel). Pour transmettre les paramètres entre appelant et appelé on peut envisager deux possibilités :

- recopier les paramètres réels empilés par l'appelant dans son environ sur le sommet de l'environ ce qui coûte en temps et en espace.
- augmenter les données standards de liaisons par un pointeur sur les paramètres. Cela alourdit dans une moindre mesure le coût d'un appel mais augmente la complexité du générateur de code.

Les tailles d'environ étant hétérogènes, l'allocation dynamique d'environnement reste très chère et il nous est paru néfaste de faire payer ce coût même dans le cas séquentiel simple. Une solution serait de mixer les deux techniques : lorsque l'analyse du graphe des appels permet de synthétiser une taille de ressources nécessaire, compiler l'allocation en gestion de pile et dans les autres cas utiliser l'allocation dynamique. Cette dernière solution offrirait à l'utilisateur l'avantage de s'abstraire de cette borne lors de la programmation sans payer systématiquement le prix fort lors de l'exécution. Toutefois sa difficulté de mise en oeuvre nous a poussé à remettre à plus tard son exploitation.

B.2.6 Environnement d'exécution

Nous avons décider de faire partager l'interface entre une tâche et le système par toutes les "thread" de cette tâche. Il en résulte que l'interface d'entrées/sorties (dont l'initialisation coûte cher) est partagée entre ces processus légers. L'utilisation d'un même fichier par plusieurs "thread" peut poser des problèmes de cohérence. La sémantique du partage de ressources systèmes est du ressort de l'utilisateur mais notre système garantit tout de même l'atomicité des opérations élémentaires d'entrées/sorties : par exemple notre système garantit que les caractères envoyés par la fonction `fprintf` arriveront au serveur de fichier dans l'ordre et en un seul bloc.

Pour des raisons de mise au point, nous avons choisi de traiter différemment les flots standards (stdin, stdout et stderr) lorsqu'il n'y a pas de redirection. En effet, en mode interactif, il est très gênant de voir se mélanger les traces en provenance de plusieurs sources (surtout lorsque ces messages sont identiques ou, plus généralement, lorsque leur contenu ne permet pas d'identifier leurs sources respectives). Le problème est encore plus important dans le cas du flot d'entrée lorsqu'il s'agit de fournir une réponse à la source (inconnue) qui a émis la requête de lecture.

Pour résoudre ces problèmes, nous avons étiqueté tous les messages en provenance d'une "thread" par l'identification de la source ainsi que de sa date d'émission (celle-ci permet la reconstitution de la trace temporelle d'un processeur dans l'éventualité de son exploitation par le serveur à qui sont destinés ces messages). L'usage de ces informations permet au destinataire de faire les traitements appropriés en différenciant (s'il le souhaite) les requêtes par leur source. C'est en particulier, le cas de notre serveur de fichiers qui redirige les requêtes correspondant aux flots standards vers le serveur d'écran qui peut alors les démultiplexer vers les fenêtres (terminaux virtuels) correspondantes. L'utilisateur dispose donc d'autant de fenêtres que de "thread" effectuant une entrée/sortie sur un des flots standards.

Remarque : Le partage de l'interface d'entrées/sorties entre les thread nous a amener à définir certains accès aux données de la tâche comme des appels de fonctions. Ainsi stdin, stdout, stderr et errno ne sont pas des variables globales mais des fonctions (par conséquent on ne peut pas les utiliser pour une initialisation statique).

Bibliographie

[AitK90] H. Ait Kaci. The WAM: A (Real) Tutorial. PRL Research Report No.5, Digital Paris Research Laboratory, 1990.

[Ali88] K. A. M. Ali Or-parallel Execution of Prolog on BC-machine In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, p.1531-1545, 1988.

[Ali90] K. A. M. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the NACLP'90*, p.757-776, MIT Press, Austin, 1990.

[Ambr90] V. Ambriola, P.Ciancarini, M.Danelutto Design and Distributed Implementation of the Parallel Logic Language Shared Prolog In ACM vol 3, pages 40-49, 1990

[Baro88a] U.C. Baron, B. Ing, M. Ratcliffe, and P. Robert. A distributed architecture for the PEPSys parallel logic programming system. In *Proceedings ICPP'88*, International Conference on Parallel Processing, Chicago, August 1988.

[Baro88b] U. Baron et al. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Tokyo, pp.841-850, 1988.

[Beau90a] A. Beaumont,S. Muthu Raman, and P. Szeredi. Scheduling or parallelism in aurora with the bristol scheduler. Technical Report TR-90-04, University of Bristol, March 1990.

[Bekk86] Bekkers, Y., Canet, B., Ridoux, O. and Ungaro, L. MALI: A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages. In *Proceedings of 3th International Symposium on Logic programming*, Salt Lake City, pp.258-264, 1986.

[Benk89] H. Benker, J.M. Beacco, A S. Bescos, A M. Dorochevsky, T Jeffré, A. Polmann, J. Noyé, B. Poterie, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, G. Watzlawik a knowledge krunching machine 16th Annual International Symposium on Computer Architecture, p186-194, Jerusalem, may 1989

[Bode89] J.P. Bodeveix LOGARITHM : Un modèle de Prolog parallèle. Son implémentation sur transputer Thèse Docteur es Sciences, Université Paris Sud, 1989

[Bosc90] P.G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and functional programming on distributed memory architectures. In *Proceedings of the 6th International Conference on Logic Programming*, pages 325–339, Jerusalem, June 1990.

[Bott89] T. Botteri-Corso. et al Un Système Unix-like pour une Station de Travail à Base de Transputers In Actes des Journées Franco-Brésiliennes sur les Systèmes Répartis, Florianopolis, 1989

[Bran88] M. Braner. Trollius Manuals. Cornel Theory Center, 1988.

[Bran88] P. Brand. and J. Almgen Wave-front Model for Scheduling in Or-Parallel Prolog. Technical Report Gigalips Project, SICS, 1988.

[Butl86] R. Butler, E. Lusk, R. Olson and R. Overbeek ANLWAM - A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, Argonne, 1986.

[Butl88] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling or-parallelism: An Argonne perspective. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seattle, August 1988.

[Cald89] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora. In *Proceedings of the 6th International Conference on Logic Programming*, p419-435, MIT press, Lisboa, June 1989.

[Carl88] M. Carlsson and J. Widen. SICStus Prolog user manual. Research report, SICS, 1988.

[Carl90] M. Carlsson. Design and Implementation of an OR-Parallel Prolog Engine Thesis, Royal Institute of Technology, Stockholm, March 1990

[Chas89a] J. Chassin de Kergommeaux. Measures of the PEPSys implementation on the MX500. Technical Report CA-44, ECRC, January 1989.

[Chas89b] J.Chassin, P. Codognet, P. Robert, et J.-C. Syre, Programmation Logique Parallele Technique et Science Informatique (TSI), Vol. 8, No 3 et 4, 1989.

[Chas89c] J. Chassin de Kergommeaux. Implémentation et évaluation d'un système logique parallèle Thèse de doctorat de l'université Joseph Fourier, GRENOBLE I, 1989

[Chik90] T. Chikayama. Current status of research and development of parallel inference systems in fifth generation computer systems. Pre-conference Workshop on Parallel Logic Programming, 7^{th} International Conference on Logic Programming, ICLP'90, Eilat, june 1990.

[Clar88] K.L. Clark Logic Programming Schemes Proceedings of the International Conference On Fifth Generation Computer Systems, p120-139, ICOT, 1988.

[Cloc88] W.F. Clocksin and H. Alshawi A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. New Generation Computing, Vol.5, pp.361-376, 1988.

[Clos53] C. Clos. A study of non blocking switching networks. In Chuan-Lin Wu and Tse-Yun Fen, editors, *Tutorial Interconnexion Networks for parallel and Distribued Processsing.* IEEE Computer Society Press, 1984. republished from The Bell System Technical Journal, pp406-424, March 1953.

[Colm90] A. Colmerauer An introduction to Prolog III Communication of ACM, 33(7):69-90, 1990

[Cone83] J.S. Conery The AND/OR model for Parallel Interpretation of logic Programs Ph.D thesis, Dept. Information and Computer Science, Univ. Calif., Irvine, 1983

[Cone85] J.S. Conery and D.F. Kibler AND Parallelism and Nondeterminism in Logic Programs. New Generation Computing, Vol.3, pp.43-70, Springer-Verlag, 1985.

[DeGr84] D. DeGroot Restricted And-Parallelism. In Proceedings of the International Conference on Fifth Generation Computer Systems 1984. ICOT, Tokyo, pp.471-478, 1984.

[DeReg88] J.P. Derycke, P. Regache. Le Transputer. Rapport d'Année Spéciale ENSIMAG, Juin 1988, Grenoble.

[Desp85] A.M. Despain and Y.N. Patt Aquarius - A High Performance Computing Sytem for Symbolic/Numeric Applications p376-382, IEEE 1985

[Dinc84] M. Dincbas and J.P. Lepape Metacontrol of logic programs in METALOG; in [ICFGCS 84]

[Dinc88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier The Constraint Logic Programming Language CHIP. in *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, vol. 1,pages 693-702, Tokyo, Dec. 1988.

[Dswa84] D.S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the International Symposium on Logic Programming*, pages 198-202, Atlantic City, New Jersey, 1984. IEEE.

[Dwor84] C. Dwork, P. Kanellakis and J. Mitchell, On the Sequential Nature of Unification. Journal of Logic Programming, Vol.1, pp.35-50, 1984.

[FaGra90] C. Fabre, F. Grardel. Contrôle de la communication pour machines massivement parallèle. Rapport de 3eme année ENSIMAG, Grenoble, Juin 1990.

[Favr89a] J. Briat, M. Favre, Y. Langué, et al PARX: a parallel operating system for transputer-based machine. In *Proceedings 10th. Occam User Group*, IOS, Springfield, Amsterdam 1989.

[Favr91] J. Briat, M. Favre, C. Geyer, J. Chassin de Kergommeaux. *Scheduling of OR-parallel Prolog on a Scalable, Reconfigurable, Distributed Memory Multiprocessor.* in PARLE 1991.

[Furu90] M .Furuichi, K. Taki, N. Ichiyoshi A Multi-Level Load Balancing Scheme for OR Parallel Exhaustive Search Programs on the Multi-PSI in ACM pp.50-59, 1990

[Garn87] N.H. Garnett. Helios: an Operating System for the Transputer. Proceedings of OUG-7, IOS, Springfield, 1987.

[Geye91] C. Geyer Une contribution à l'étude du parallélisme OU en Prolog sur des machines sans mémoire commune These de Doctorat de l'université Joseph Fourier, GRENOBLE, 1991

[Ghos90] D. Ghosal, G. Serazzi and S. K. Tripathi. Processor Working Set and Its Use in Scheduling Multiprocessor Systems. Institute for Advanced Computer Studies, Computer Science Dept., Univ. of Maryland, College Park, (UMIACS-TR), Jan. 1990.

[Ghosal90] Dipak Ghosal and al. Characterizing Parallel Program Behavior: An Experimental Study. Institute for Advanced Computer Studies, Computer Science Dept., Univ. of Maryland, College Park, (UMIACS-TR), 1990.

[Harp86] J.G. Harp, C.R. Jesshope, T. Muntean, and C. Whitby-Stevens. The development and application of a low cost high performance multiprocessor machine. In *Proceedings ESPRIT'86: Results and Achievements.* Elsevier Science Publishers, 1986.

[Haus87] B. Hausman, A. Ciepielewsky, S. Haridi OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors In IEEE, p69-79, 1987.

[Haus89] B. Hausman. Pruning and scheduling speculative work in or-parallel system. In *PARLE89*, Eindhoven, june 1989.

[Herm90] M. Hermenegildo S. K. Debray, N-W Lin. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN'90 Conference on*

Programming Language Design and Implementation, White Plains, New York, June 20-22 1990.

[Hoar78] C.A.R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, n 8, Aug. 1978.

[Inmos84] David May. occam Definition. Inmos Notice, 1984.

[Inmos87] Inmos Ltd. IMS T800 transputer.

[Inmos88] Inmos Ltd. Transputer Development System. Prentice Hall, 1988.

[Kacs90] P. Kacsuk, I. Futo and Sz. Ferenczi Implementing CS-Prolog on a communicating process architecture Journal of Microcomputer Applications, vol 13, p19-41

[Kern78] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice Hall 1978.

[Kowa74] R. Kowalski Predicate Logic as a Programming Language. Information Processing 74, IFIP Congress, pp.569-574, 1974.

[Kowa79] R. Kowalski Algorithm = Logic + Control. Communications of the ACM, Vol.22, pp.424-436, 1979.

[Kumo86] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh and Y.Sohma Kabu-Wake: A New Parallel Inference Method and Its Evaluation ICOT Technical Report TR-150, 1986

[Lloy88] J.-W. Lloyd Fondements de la Programmation Logique. Editions Eyrolles, Paris, 1988.

[Lusk88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B Haussman. The Aurora Or-Parallel Prolog System In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, November 1988

[Muda89] S. Mudambi Performance of Aurora on a Switch-Based Multiprocessor. In *Proceedings of North American Conference on Logic Programming*, Cleveland, pp.697-712, 1989.

[Mart82] A. Martelli and U. Montanari An Efficient Unification Algorithm *ACM Transactions on Programming Languages and Systems*, 1982

[Masu86a] H. Masuzawa and et al. Kabu wake parallel inference mechanism and its evaluation. In *1986 FJCC*, pages 955–962. IEEE, November 1986.

[Memm82] G Memmi and Y. Raillard Some New Results About the (d,k) Graph Problem IEEE Transactions on Computers, Volume C-31, Number 8, 1982

[More91] E. Morel Implantation des Predicats à effets de bord pour Prolog Parallèle Rapport de DEA, INPG, Grenoble, 1991

[Munt88] Traian Muntean. Les Super-calculateurs à Transputers La Recherche n 204, Vol. 19, Novembre 1988

[Pere84] L.M. Pereira, R. Nasr Delta-Prolog: a distributed logic programming language in Procedings of FGCS, Tokyo, 1984

[Robi65] J. Robinson A Machine-Oriented Logic Based on Resolution Principle. Journal of the ACM, Vol.12, pp.23-41, 1965.

[Shap86] E. Shapiro. Concurrent Prolog: A progress report. Computer 19, 8 Aug. 1986, p44–58.

[Shap89] E. Shapiro. The family of concurrent logic programming languages. *ACM computing surveys*, 21(3), september 1989.

[Shea89] D. Shea et al. The IBM Victor Multiprocessor Project. Proc. of the 4th International Conference on Hypercubes, April 1989.

[Shen87] D. Shen A Simulation Study of the Argonne Model for parallel execution of Prolog IEEE p54-68, 1987

[Szer89] Peter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceeding of the North American Conference on Logic Programming NACLP'89*, Cleveland, October 1989.

[Tane87] A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[Tayl90] A Taylor LIPS on a MIPS: Results from a Prolog compiler for a RISC Seventh International Conference on logic Programming, p174-185, Jerusalem, Jun 1990

[Thib90] O Thibault, J Noye, H Benker KCM: une machine Prolog 2eme Symposiun Architectures Nouvelles de Machines, p311-332, Toulouse, Sep 1990

[Tick88] P.A. Tinker Compile-time granularity analysis for parallel logic programming languages Proceedings of the International Conference On Fifth Generation Computer Systems, pp.994-1000, ICOT, 1988.

[Tink87] P.A. Tinker and G. Lindstrom, A Performance-oriented Design for Or-Parallel Logic Programming. In Proceedings of 4th International Conference on Logic Programming, Melbourne, pp.601-615, 1987.

[Tink88] P.A. Tinker Performance of an Or-Parallel Logic Programming System In International Journal of Parallel Programming, Vol 17, no 1, pp.59-92, 1988.

[Touz90a] A. Touzene and B. Plateau. Mesures de performance des communications du meganode à 128 transputers. Technical report, LGI-IMAG, projet CMaP, 46, avenue Félix Viallet, 38031 Grenoble Cédex, France, 1990.

[VanR84] P. Van Roy A Prolog Compiler for the PLM Report No. UCB/CSD 84/203 November 1984 Computer Science Division (EECS) University of California, Berkeley, California 94720

[Wail90] Ph. Waille. *Introduction à L'architecture des Machines Supernode.* Rapport Technique n 56, LGI-IMAG, Février 1990.

[Warr77] D.H.D. Warren Implementing Prolog - Compiling Predicate Logic Programs. Research Reports No. 39, 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[Warr83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report tn309, SRI, October 1983.

[Warr87a] D.H.D. Warren. Or-parallel execution models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, pp 243–259, Springer-Verlag, March 1987

[Warr87b] D.H.D. Warren. The SRI model for or-parallel execution of Prolog. abstract design and implementation issues. In 4^{th} *Symposium on Logic Programming*, pages 92–102. San Fransisco, Sept. 1987.

[West87a] H. Westphal, P. Robert, J. Chassin, and J.-C. Syre. The PEPSys model: Combining backtracking, and- and or-parallelism. In 4^{th} *Symposium on Logic Programming*, pages 436–448, San Fransisco, Sept. 1987.

Table des matières

Table des figures

Table des tableaux