GONCALVES

N° d'ordre : 1211

UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir

LE TITRE DE DOCTEUR DE 3ème CYCLE

Daniel ROCACHER

ETUDE D'UNE ARCHITECTURE DIRIGEE PAR LES DONNEES, APPLIQUEE AU LANGAGE ADA.

Thèse soutenue le 19 octobre 1984, devant la Commission d'Examen

Membres du Jury :	C. CARREZ	Président
	V. CORDONNIER	Rapporteur
	P. DOUSPIS	Examinateur
	A. RECOQUE	Examinateur
	C. RENVOISE	Examinateur

P R O F E S S E U R S CLASSE EXCEPTIONNELLE

M.	CONSTANT Eugène	I.E.E.A.
M.	FOURET René	Physique
M.	GABILLARD Robert	I.E.E.A.
M.	MONTREUIL Jean	Biologie
M.	PARREAU Michel	Mathématiques
M.	TRIDOT Gabriel	Chimie
M.	VIVIER Emile	Biologie
M.	WERTHEIMER Raymond	Physique

P R O F E S S E U R S 1ère classe

M.	BACCHUS Pierre	Mathématiques
M.	BEAUFILS Jean-Pierre (dét.)	Chimie
M.	BIAYS Pierre	G.A.S.
M.	BILLARD Jean (dét.)	Physique
M.	BOILLY Bénoni	Biologie
M.	BOIS Pierre	Mathématiques
M.	BONNELLE Jean-Pierre	Chimie
M.	BOUGHON Pierre	Mathématiques
M.	BOURIQUET Robert	Biologie
M.	BREZINSKI Claude	I.E.E.A.
M.	CELET Paul	Sciences de la Terre
M.	CHAMLEY Hervé	Biologie
M.	COEURE Gérard	Mathématiques
M.	CORDONNIER Vincent	I.E.E.A.
M.	DEBOURSE Jean-Pierre	S.E.S.
M.	DYMENT Arthur	Mathématiques

PROFESSEURS 1ère classe (suite)

M. ESCAIG Bertrand	Physique
M. FAURE Robert	Mathématiques
M. FOCT Jacques	Chimie
M. GRANELLE Jean-Jacques	S.E.S.
M. GRUSON Laurent	Mathématiques
M. GUILLAUME Jean	Biologie
M. HECTOR Joseph	Mathématiques
M. LABLACHE COMBIER Alain	Chimie
M. LACOSTE Louis	Biologie
M. LAVEINE Jean Pierre	Sciences de la Terre
M. LEHMANN Daniel	Mathématiques
Mme LENOBLE Jacqueline	Physique
M. LHOMME Jean	Chimie
M. LOMBARD Jacques	S.E.S.
M. LOUCHEUX Claude	Chimie
M. LUCQUIN Michel	Chimie
M. MIGEON Michel Recteur à Grenoble	E.U.D.I.L.
M. MIGNOT Fulbert (dét.)	Mathématiques
M. PAQUET Jacques	Sciences de la Terre
M. PROUVOST Jean	Sciences de la Terre
M. ROUSSEAU Jean-Paul	Biologie
M. SALMER Georges	I.E.E.A.
M. SEGUIER Guy	I.E.E.A.
M. SIMON Michel	S.E.S.
M. STANKIEWICZ François	S.E.S.
M. TILLIEU Jacques	Physique
M. VIDAL Pierre	I.E.E.A.
M. ZEYTOUNIAN Radyadour	Mathématiques

P R O F E S S E U R S 2ème classe

M. ANTOINE Philippe	Mathématiques (Calais)
M. BART André	Biologie
Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mathématiques
M. BELLET Jean	Physique
M. BERZIN Robert	Mathématiques
M. BKOUCHE Rudolphe	Mathématiques
M. BODARD Marcel	Biologie
M. BOSQ Denis	Mathématiques
M. BRASSELET Jean-Paul	Mathématiques
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie
M. CARREZ Christian	I.E.E.A.
M. CAYATTE Jean-Louis	S.E.S.
M. CHAPOTON Alain	C.U.E.E.P.
M. COQUERY Jean-Marie	Biologie
Mme CORSIN Paule	Sciences de la Terre
M. CORTOIS Jean	Physique
M. COUTURIER Daniel	Chimie
M. CROSNIER Yves	I.E.E.A.
M. CURGY Jean-Jacques	Biologie
Mle DACHARRY Monique	Géographie
M. DAUCHET Max	I.E.E.A.
M. DEBRABANT Pierre	E.U.D.I.L.
M. DEGAUQUE Pierre	I.E.E.A.
M. DELORME Pierre	Biologie
M. DELORME Robert	S.E.S.
M. DE MASSON D'AUTUME Antoine	S.E.S.
M. DEMUNTER Paul	C.U.E.E.P.

PROFESSEURS 2ème classe (Suite 1)

M. DENEL Jacques	I.E.E.A.
M. DE PARIS Jean-Claude	Mathématiques (Calais)
Mle DESSAUX Odile	Chimie
M. DEVRAINNE Pierre	Chimie
M. DHAINAUT André	Biologie
Mme DHAINAUT Nicole	Biologie
M. DORMARD Serge	S.E.S.
M. DOUKHAN Jean-Claude	E.U.D.I.L.
M. DUBOIS Henri	Physique
M. DUBRULLE Alain	Physique (Calais)
M. DUBUS Jean-Paul	I.E.E.A.
M. FAKIR Sabah	Mathématiques
M. FONTAINE Hubert	Physique
M. FOUQUART Yves	Physique
M. FRONTIER Serge	Biologie
M. GAMBLIN André	G.A.S.
M. GLORIEUX Pierre	Physique
M. GOBLOT Rémi	Mathématiques
M. GOSSELIN Gabriel (dét.)	S.E.S.
M. GOUDMAND Pierre	Chimie
M. GREGORY Pierre	I.P.A.
M. GREMY Jean-Paul	S.E.S.
M. GREVET Patrice	S.E.S.
M. GUILBAULT Pierre	Biologie
M. HENRY Jean-Pierre	E.U.D.I.L.
M. HERMAN Maurice	Physique
M. JACOB Gérard	I.E.E.A.
M. JACOB Pierre	Mathématiques
M. JEAN Raymond	Biologie
M. JOFFRE Patrick	I.P.A.

PROFESSEURS 2ème classe (suite 2)

M. JOURNEL Gérard	E.U.D.I.L.
M. KREMBEL Jean	Biologie
M. LANGRAND Claude	Mathématiques
M. LATTEUX Michel	I.E.E.A.
Mme LECLERCQ Ginette	Chimie
M. LEFEVRE Christian	Sciences de la Terre
Mle LEGRAND Denise	Mathématiques
Mle LEGRAND Solange	Mathématiques (Calais)
Mme LEHMANN Josiane	Mathématiques
M. LEMAIRE Jean	Physique
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique
M. LOSFELD Joseph	C.U.E.E.P.
M. LOUAGE Francis (dét.)	E.U.D.I.L.
M. MACKE Bruno	Physique
M. MAIZIERES Christian	I.E.E.A.
M. MESSELYN Jean	Physique
M. MESSERLIN Patrick	S.E.S.
M. MONTEL Marc	Physique
Mme MOUNIER Yvonne	Biologie
M. PARSY Fernand	Mathématiques
Mle PAUPARDIN Colette	Biologie
M. PERROT Pierre	Chimie
M. PERTUZON Emile	Biologie
M. PONSOLLE Louis	Chimie
M. PORCHET Maurice	Biologie
M. POVY Lucien	E.U.D.I.L.
M. RACZY Ladislas	I.E.E.A.
M. RAOULT Jean François	Sciences de la Terre
M. RICHARD Alain	Biologie

PROFESSEURS 2ème Classe (suite 3)

M. RIETSCH François	E.U.D.I.L.
M. ROBINET Jean-Claude	E.U.D.I.L.
M. ROGALSKI Marc	Mathématiques
M. ROY Jean-Claude	Biologie
M. SCHAMPS Joël	Physique
Mme SCHWARZBACH Yvette	Mathématiques
M. SLIWA Henri	Chimie
M. SOMME Jean	G.A.S.
Mle SPIK Geneviève	Biologie
M. STAROSWIECKI Marcel	E.U.D.I.L.
M. STERBOUL François	E.U.D.I.L.
M. TAILLIEZ Roger	Institut Agricole
Mme TJOTTA Jacqueline (dét.)	Mathématiques
M. TOULOTTE Jean-Marc	I.E.E.A.
M. TURRELL Georges	Chimie
M. VANDORPE Bernard	E.U.D.I.L.
M. VAST Pierre	Chimie
M. VERBERT André	Biologie
M. VERNET Philippe	Biologie
M. WALLART Francis	Chimie
M. WARTEL Michel	Chimie
M. WATERLOT Michel	Sciences de la Terre
Mme ZINN JUSTIN Nicole	Mathématiques

CHARGES DE COURS

M. ADAM Michel	S.E.S.

CHARGES DE CONFERENCES

M. BAFCOP Joël	I.P.A.
M. DUVEAU Jacques	S.E.S.
M. HOFLACK Jean	I.P.A.
M. LATOUCHE Serge	S.E.S.
M. MALAUSSENA DE PERNO Jean-Louis	S.E.S.
M. NAVARRE Christian	I.P.A.
M. OPIGEZ Philippe	S.E.S.

Je tiens à remercier

Monsieur CARREZ, Professeur à l'U.S.T.L., de me faire l'honneur de présider ce jury.

Monsieur CORDONNIER, Professeur à l'U.S.T.L., qui a accepté de me prodiguer conseils et encouragements dans la réalisation de cette thèse.

Monsieur RENVOISE, de la Société "La Mondiale", qui en tant que Chef de service au Centre de Recherche de la Compagnie des Machines BULL, m'a accueilli avec bienveillance et sympathie dans son service et a donné à ce travail son orientation et sa consistance.

Monsieur DOUSPIS, Ingénieur de la Société ALSYS, qui a accepté d'examiner ce travail avec beaucoup de patience et de gentillesse. Ses conseils, les nombreuses discussions que nous avons eues et son amitié m'ont été précieux.

Madame RECOQUE, Déléguée Scientifique au Groupe BULL, d'avoir bien voulu participer au Jury.

Je remercie également

Monsieur ROHMER et tous les membres de son équipe qui m'ont cordialement aidés par leur bonne humeur, leur optimisme et leur joïe de vivre m'apportant ainsi un soutien moral si important.

Madame PHILIPPE qui a dactylographié ce rapport avec beaucoup de gentillesse, efficacité et compétence.

Madame DEBOCK qui a assuré avec soin et diligence le tirage de ce document.

TABLE DES MATIERES

0. INTRODUCTION

1. EXPLOITATION DU PARALLELISME

1.1. Concepts de base

1.1.1. Types de dépendances

1.1.2. Diverses formes de parallélisme

1.2. Les modes de fonctionnement

1.2.1. Fonctionnement centré sur les instructions

1.2.2. Fonctionnement centré sur les données

1.2.2.1. Travaux du M.I.T.

1.2.2.2. Travaux du C.E.R.T.

1.2.2.3. Travaux de l'Université d'Irvine

1.2.2.4. Travaux de l'Université de Lille

1.2.2.5. Autres projets

1.3. Critiques des machines dirigées par les données

1.3.1. Point de vue logiciel

1.3.2. Point de vue matériel

1.3.3. Point de vue technique

1.4. Conclusion

2. LE MODELE DATA FLOW MACROSCOPIQUE

2.1. Le modèle statique

2.1.1. Concept d'atome

2.1.2. Traitement effectif

2.1.3. Mécanisme d'enchainement d'exécution des atomes

2.1.4. Traitement de contrôle-Mot d'état

2.2. Le modèle dynamique

2.2.1. Nécessité d'un modèle dynamique

2.2.2. Concept de molécule

2.3. Traitement des constructions classiques des langages de programmation

2.3.1. Prise en compte des instructions de choix (IF, CASE)

2.3.2. Prise en compte des boucles

2.3.3. Prise en compte des sous-programmes

2.4. Conclusion

3. LE DATA FLOW MACROSCOPIQUE APPLIQUE AU LANGAGE ADA

3.1. Parallélisation des programmes Ada

3.1.1. Ordonnancement des points de rendez-vous

3.1.2. Parallélisme et traitement d'exception

3.2. Data Flow Macroscopique appliqué aux tâches

3.2.1. Prise en compte des mécanismes de "rendez-vous"

3.2.1.1. Prise en compte des instructions "accept" et appel d'entrée

3.2.1.2. Prise en compte des appels conditionnels d'entrée

3.2.1.3. Prise en compte des appels d'entrée avec attente limitée

3.2.1.4. Prise en compte de l'instruction "select"

3.2.3. Prise en compte des mécanismes d'exception

3.3. Conclusion

4. ELEMENTS D'UNE ARCHITECTURE MULTI-MICROPROCESSEURS

4.1. MLI, un processeur de base

4.1.1. La fonction mémoire

4.1.2. L'adressage lexical

4.1.3. Mécanismes de gestion des tâches

4.1.3.1. Mécanismes d'activation d'une tâche

4.1.3.2. Mécanismes de rendez-vous

4.1.3.3. Mécanismes de traitements d'exception

4.1.4. Architecture matérielle de MLI

4.2. Les réseaux d'interconnexion

4.2.1. Quelques architectures

4.2.1.1. Réseaux Crossbars

4.2.1.2. Réseaux multi-étages

4.2.1.3. Autres réseaux

4.2.2. Principes de commandes

4.2.3. Propriétés

4.2.3.1. Modularité et extensibilité

4.2.3.2. Partitionnement

4.2.3.3. Tolérance aux Pannes

4.2.3.4. Performances

4.2.4. Conclusion

4.3. Eléments d'une architecture multi-processeurs

4.3.1. Caractéristiques d'une architecture adaptée au modèle

4.3.2. Eléments d'architecture matérielle de AMOS

4.3.2.1. Le réseau d'interconnexion

4.3.2.2. La mémoire principale

4.3.2.3. Les processeurs d'exécution

5. MISE EN OEUVRE DU DATA FLOW MACROSCOPIQUE SUR MLI

5.1. Data Flow Macroscopique et Unité de Compilation

5.1.1. Exploitation du parallélisme dans une unité de compilation

5.1.2. Exécution parallèle d'unités de programmation non-imbriquées

5.1.3. Exécution parallèle d'unités de programmation imbriquées

5.1.4. Exécution parallèle des "unités de programmation"

5.1.5. Exécution parallèle des boucles dynamiques

5.2. Data Flow Macroscopique et mécanismes de rendez-vous

5.3. Mise en oeuvre des exceptions

5.4. Sur la taille des processus parallélisés

6. CONCLUSION

CHAPITRE O

I N T R O D U C T I O N

1. INTRODUCTION

Dans la course aux machines hautes performances la nouvelle génération d'ordinateurs, appelée 5ème Génération, se démarquera fortement du passé. De nombreuses architectures parallèles ont été étudiées dans les centres de recherche du monde entier et de grands projets ont été lancés, en particulier au Japon avec le projet national FGCS (Fifth Generation Computer System), afin de définir les ordinateurs des années 1990 /Mot 83/.

Quatre générations d'ordinateurs se sont succédées en moins de quarante ans reposant sur une évolution de la technologie : tube à vide, transistors, circuits intégrés, LSI (Large Scale Integrated Circuits). Ces générations ont augmenté les performances de sept ordres de magnitude tout en faisant chuter de façon importante les prix.

Cependant le principe de fonctionnement de ces machines n'a pas changé fondamentalement durant ces années et se caractérise par le principe défini par Von Neuman explicitant la forme de séquencement par le cadencement des instructions.

Ces architectures classiques sont essentiellement monoprocesseur exécutant des instructions les unes après les autres, dans l'ordre défini par le programme, grâce à un compteur ordinal.

La 5ème Génération est marquée par une nouvelle étape de la technologie le VLSI (Very Large Scale Integrated Circuits) dont les capacités d'intégration permettent de disposer sur un boitier d'un véritable mini-ordinateur (tel le HP 9000 qui comporte 450 K transistors). Les prévisions permettent de penser que dans les années 90 plus d'un million de transistors pourront être intégrés sur une puce.

Cependant la technologie existante ou prévisible à court terme possède des limites qui ne pourront pas satisfaire les besoins importants en puissance de calcul. Selon Wilson /Wil 82/, ils devraient être de l'ordre d'une centaine de magnétude supérieure aux gros calculateurs actuels, dans la prochaine décade.

La simulation des grands systèmes physiques rencontrés en dynamique des fluides et des solides, la construction aéronautique et spatiale, la physique des plasmas, les prévisions météorologique, la géophysique, la ballistique necessitent une telle capacité de calcul. De nombreuses autres disciplines sont concernées, ainsi l'Intelligence Artificielle dont les systèmes experts consultent, à l'aide d'un parcours d'arbre, des bases de connaissance de plus en plus importantes pour appliquer des des analyses déductives, le traitement de la parole, la reconnaissance des formes et le traitement des images sont grands consommateurs de MIPS (Million d'instructions par seconde).

Ainsi l'enjeu est important et vital pour l'armée mais aussi pour le monde médical et industriel.

La simultanéité est la clé de la haute performance.

Les chercheurs sont trés nombreux à penser que la réponse naturelle aux besoins en puissance de calcul passe par l'intermédiaire de l'exploitation du parallélisme inhérent aux programmes grâce à des architectures parallèles : les multiprocesseurs.

Si jusqu'à présent la conception de ces machines se heurtait à une barrière technologique, désormais, l'évolution des VLSI permet d'envisager la construction de système regroupant plusieurs centaines de ces circuits.

Les arguments en faveur des multiprocesseurs justifient l'intérêt énorme qu'ils provoquent ces dernières années.

- les performances grâce à l'exploitation du parallélisme des programmes ;

- la fiabilité grâce aux possibilités de redondance et de fonctionnement en mode dégradé ;

- le rapport performance/prix : l'intérêt économique d'un système répétitif construit à partir d'un petit nombre de plaques, minimisant le coût de développement et d'industrialisation est certain ;

- la modularité que présente toute structure répétitive permet soit d'augmenter la puissance d'incréments relativement faibles, soit de pouvoir établir une gamme de produits ;

- la disponibilité de ces systèmes dû à leur possibilité de reconfiguration et de partitionnement permet de développer des techniques de maintenance, de test sous système, voire d'auto-tests ;

- la facilité d'utilisation grâce à la spécialisation de tout ou partie de ces machines.

Cependant, malgré ces avantages nombreux, la machine performante, fiable et peu chère n'est probablement pas encore construite car de nombreux problèmes ne sont pas encore bien maitrisés :

- les communications entre les différentes unités fonctionnelles ;

- leur synchronisation ;
- la parallélisation et la distribution des tâches ;
- les entrées - sorties ;
- les arrangements mémoire ;
- les bloquages et les problèmes d'exclusion mutuelle .

Les objectifs de cette thèse

Une alternative à ces problèmes du parallélisme et de son exploitation est de faire sauter les carcans des principes de Von Neuman à l'aide d'un shéma d'exécution dirigée par les données. Dans une machine dirigée par les données, l'exécution d'un programme n'est plus cadencée par le flot des instructions comme dans les machines classiques, mais par le flot des données. Ces travaux ont conduit à l'étude de langages spécifiques, dits les langages "data flow" et à l'étude d'architectures de machines adaptées à leur exécution. Mais ils n'ont, à ce jour, abouti au mieux qu'à la réalisation de prototype.

Un tel parallélisme est exploité en général au niveau des instructions du programme ce qui engendre des fonctions de cadencement d'un coût élevé relativement au parallélisme dégagé. Aussi, dans cette thèse nous rappelons un principe d'exploitation du parallélisme à un niveau plus global dit macroscopique. Ce mode d'expression du parallélisme présente de nombreux avantages, tant au niveau efficacité, dans la mesure où il permet d'adapter la taille des processus parallélisés à leur fonction de cadencement, qu'au niveau utilisation car il est transparent à l'utilisateur et au langage de programmation qu'il utilise. En outre, ce parallélisme peut conduire à la définition d'architectures parallèles relativement classiques utilisant des composants du commerce.

Notre propos est, outre de faire comprendre les intérêts du modèle d'expression du parallélisme présenté, de montrer ses qualités d'adaptation non seulement, à des structures classiques des langages de programmation, mais aussi à des structures plus complexes à mettre en oeuvre telles celles proposées par le langage Ada : tâches, rendez-vous, exception.

Notre étude a consisté d'autre part à proposer des éléments d'architecture permettant la réalisation du data flow macroscopique pour les structures classiques et complexes des langages de programmation. Cette étude montre qu'une telle architecture apparaftêtre une très bonne orientation, grâce aux solutions technologiques actuellement disponibles, pour obtenir un maximum de performances.

Le plan de cette thèse

Après un examen de nombreuses machines parallèles, des réseaux d'interconnexion et d'une machine langage intermédiaire, une architecture est proposée pour l'application du modèle data flow macroscopique au langage Ada.

Le chapitre 1 est une étude bibliographique des systèmes parallèles existants. Nous y avons rappelé quelques principes de base de la parallélisation et décrit le mode de fonctionnement de différentes machines parallèles afin d'en faire une analyse critique.

Le chapitre 2 rappelle les principes du modèle data flow macroscopique et ses applications aux structures classiques des langages de programmation.

Le chapitre 3 montre comment le data flow macroscopique permet de prendre en compte des mécanismes support du temps réel et de récupération d'erreur. Ainsi, après avoir examiner les problèmes de sémantique posés par la parallélisation des tâches Ada, les prises en compte des instructions "accept", appel d'entrée, appel d'entrée conditionnel, appel d'entrée avec attente limitée, "select" et des mécanismes d'exception du langage Ada, ont été étudiées.

Le chapitre 4 rappelle quelques caractéristiques d'un processeur langage intermédiaire dont la connaissance se révèle nécessaire à la compréhension de la mise en oeuvre du modèle effectuée au chapitre suivant. De plus quelques éléments d'architectures de réseaux d'interconnexion et d'une machine multi-microprocesseurs sont présentés.

Le chapitre 5 montre une mise en oeuvre possible du modèle data flow macroscopique et traite, en particulier, des problèmes d'adressage lors de l'exécution parallèle de différentes unités d'un programme Ada, ainsi que la mise en oeuvre des solutions proposées au chapitre 3 pour la prise en compte des mécanismes de rendez-vous et d'exception. Il faut remarquer que pour, la clarté de notre exposé, nous avons toujours tenté de séparer les problèmes d'expression du parallélisme grâce au data flow macroscopique (chapitre 3) et les problèmes de mise en oeuvre (chapitre 5), bien que ceux-ci soient parfois très liés et difficilement dissociables.

Le 6ème chapitre qui expose les enseignements tirés de cette étude ainsi que ses évolutions possibles, forme la conclusion de ce rapport.

CHAPITRE I

EXPLOITATION DU PARALLELISME

1. EXPLOITATION DU PARALLELISME

1.0 INTRODUCTION

Dans ce chapitre les principes de base de la détection et de l'expression du parallélisme sont d'abord présentés. Puis, après avoir rapidement examiné les machines à fonctionnement centré sur les instructions qui ont été étudiées par ailleurs de façon complète dans (DR 83), quelques systèmes à fonctionnement centrés sur les données sont présentés. Par la suite, une analyse critique de ces machines est effectuée afin de mieux comprendre les choix techniques adoptés pour le modèle d'expression du parallélisme et l'architecture présentés dans cette thèse.

1.1. CONCEPTS DE BASE

Les problèmes de détections du parallélisme inhérent aux programmes et de leur exploitation ont fait l'objet de nombreuses recherches. Dans ce chapitre les différents moyens d'expression et d'exploitation du parallélisme sont passés en revue.

1.1.1. Types de dépendances

Pour déterminer le parallélisme exploitable il y a trois types de dépendance à détecter : celle sur les données, celles sur les contrôles, et celles sur les ressources. Les deux premières dépendent des programmes et la troisième des machines.

- la dépendance des données

L'étude théorique de la parallélisation des programmes montre que la mise en parallèle d'instructions différentes repose sur la recherche de relations de dépendances basées sur la production et consommation des variables du programme.

Considérant deux instructions I1, I2, séquentielles dans le programme initial, celles-ci sont exécutables simultanément si les trois contraintes suivantes peuvent être levées :

- PC : l'instruction I1 produit une valeur d'une variable consommée par I2 ;

- CP : l'instruction I1 consomme une valeur d'une variable que I2 produit ;

- PP : les deux instructions produisent des valeurs d'une même variable. L'ordre de production doit être alors maintenu.

Ces conditions dites de Bernstein /Ber 66/ ont été reprises pour effectuer des outils de parallélisation automatique de programmes écrits en langage évolué comme VAST /Vas 82/ ou VESTA /CRS 83/.

Les relations CP et PP traduisent des contraintes issues de la réutilisation des contenants d'information et peuvent être supprimées en se plaçant dans un contexte dit d'Assignation Unique, concept introduit par Tesler et Enea /Tes 68/. La définition donnée à l'assignation unique est la suivante :

"Une variable ne peut recevoir au plus qu'une valeur pendant toute l'exécution du programme".

- <u>la dépendance des contrôles</u>

Ces dépendances varient selon les langages et sont dûes aux exécutions conditionnelles. Ainsi une dépendance conditionnelle éxiste entre un IF et ses parties VRAI et FAUSSE, l'exécution de l'une d'elle est conditionnée par le résultat du test.

De même il existe une dépendance entre une tête de boucle et les instructions de chaque itération, une dépendance de branchement existe entre GOTO et sa destination.

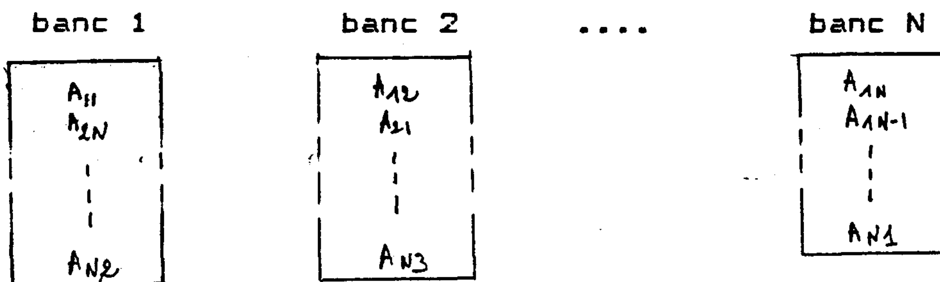
- <u>la dépendance des ressources</u>

Ce type de dépendance est dû au partage des différentes unités physiques d'une machine parallèle par les tâches qui sont exécutées simultanément.

Ainsi, par exemple, le nombre de tâches exécutables en parallèle peut être supérieur au nombre de processeurs utilisables engendrant une certaine séquentialisation des tâches. Le problème difficile de la distribution des travaux sur une machine parallèle est posé, comment distribuer aux processeurs les tâches exécutables afin d'obtenir le meilleur rendement ? La réponse est d'autant moins aisée que le nombre de facteurs à prendre en compte est important (nombre et partage des ressources, priorité et vieillissement des tâches, répartition des processus en fonction de la localité des programmes) ...

Un des problèmes majeurs dans l'exploitation du parallélisme est d'éviter les goulots d'étranglement dûs aux partages des ressources, ils séquentialisent les travaux et dégradent de façon importante les performances. Ainsi de nombreuses études (Résumés dans /Roc 83/) ont été réalisées afin de définir des systèmes de communication entre les différents éléments d'une machine qui soient le moins bloquants possible.

De même les problèmes d'accès et d'adressage de la mémoire sont particulièrement importants, et de nombreux travaux ont été réalisés afin de définir des arrangements mémoire. Ils montrent qu'elle doit être divisée en M unités ou bancs et proposent des algorithmes de rangement des données, en fonction des problèmes à résoudre, de façon à réaliser M accès simultanément. Ainsi beaucoup de calcul font intervenir des vecteurs ou des matrices sur lesquels des processeurs opèrent simultanément, l'adressage des matrices doit être tel que les éléments d'une colonne ou d'une même ligne puissent être obtenus sans conflit-mémoire. L'exemple suivant (figure 1) est une illustration du stockage d'une matrice carrée d'ordre N, sur N bancs mémoires.



pas de conflit d'accès ni à une ligne, ni à une colonne.

fig. 1 : Exemple de stockage d'une matrice An,n

En particulier deux modes de stockage sont couramment utilisés :

- le stockage vertical, correspondant par exemple au rangement des éléments d'un vecteur sur un seul banc, et

permet de préserver la localité des variables en privilégiant une connexion particulière processeur-banc ;

- le stockage horizontal, correspondant au rangement des éléments d'un vecteur, successivement sur l'ensemble des bancs, et favorisant le partage entre N processeurs d'un ensemble de variables.

La mémoire doit être suffisamment grande pour contenir un grand nombre de données, et, d'autre part assez rapide pour ne pas ralentir la vitesse d'exécution. Le nombre même de bancs est important, ainsi le multiprocesseur FMP /LB 80/, construit à la NASA, a-t-il été conçu avec un nombre premier de bancs pour diminuer les conflits.

La grande difficulté dans la conception des systèmes parallèles est d'éliminer les points goulots d'étranglement sans déplacer les problèmes ou en créer de nouveaux.

- <u>la détection du parallélisme</u> :

La détection et la mise en évidence du parallélisme peuvent être faites de plusieurs manières sur un programme :

- Lors du choix des algorithmes : un algorithme donné introduit certaines dépendances entre les données qu'il manipule et sa structure même est plus ou moins bien adaptée à l'exploitation du parallélisme. La parallélisation d'algorithmes numériques fait l'objet de nombreux travaux en analyse numérique (transformée rapide de Fourrier (FFT), méthode itérative de Gauss-Seidel, Méthode de résolution de systèmes linéaires, méthode de discrétisation pour la résolution des systèmes d'équations différentielles aux données partielles, algorithmes élastiques...) ;

- à la programmation : les structures de certains langages de programmation permettent d'expliciter le parallélisme dans un programme. Ainsi les tâches en Ada sont des processus pouvant être exécuté en parallèle ;

- à la compilation : l'analyse des programmes à la compilation ou la pré-compilation peut supprimer des dépendances (vectorisation ou parallélisation automatique, optimisation) ;

- au décodage des instructions : l'unité de contrôle peut examiner une ou plusieurs instructions, détecter des dépendances sur des flots d'instuctions ou sur des ressources. Les instructions peuvent être spécialisées pour certains types de traitement comme les opérations vectorielles ou méterielles.

- à l'exécution des instructions : ce type de dépendance correspond essentiellement à la structure matérielle de la machine sur laquelle est exécutée l'instruction (processeur,

mémoire et réseau d'interconnexion).
Sur une machine vectorielle, l'unité de contrôle détermine comment commander les différentes parties de la machine en fonction de l'instruction à exécuter.

1.1.2. Les diverses formes de parallélisme

Le parallélisme le plus simple qui puisse être envisagé est l'exécution simultanée de travaux pour plusieurs usagers. Il suffit dans ce cas d'implanter un système d'exploitation assez classique.

Le parallélisme de niveau suivant est l'exploitation du parallélisme existant dans l'énoncé des programmes lorsque ceux-ci sont écrits dans des langages offrant des outils puissants et agréables à utiliser.

Dès 1963 /Con 63/ proposait des outils spécialement conçus pour expliciter le parallélisme d'un programme. Il a introduit des instructions de contrôle dont l'effet est de lancer un calcul parallèle sur un autre processeur et d'attendre que les calculs soient terminés pour continuer : les FORK et JOIN :

- FORK A, J : provoque le démarrage d'un processeur sur le programme situé en A ; le compteur J est incrémenté de 1. Le processeur qui a émis le FORK continue en exécutant l'adresse suivante.

- JOIN J, B : assure la décrémentation du compteur J et la poursuite de l'exécution de la séquence d'instructions débutant par B si la valeur du compteur est nulle. Dans le cas contraire, l'unité de traitement est désactivée.

Ce type de parallélisme est conçu pour une exécution sur une architecture à mémoire commune car il ne prévoit aucun mécanisme de communication de données.

Cette explicitation du parallélisme par le programmeur peut être faite grâce aux structures même des langages de haut niveau. Ainsi des langages comme LTR, Ada (USG 83) ou concurrent Pascal 7Brh 75/ matérialisent un concept de processus, à l'aide de tâches en Ada ou de processus en LTR.

De plus ces langages disposent d'outils de synchronisation et de communication entre processus. En Ada la synchronisation et la communication entre tâches sont exprimées par la notion de "rendez-vous".

- Le parallélisme utilisé dans les machines les plus

puissantes actuelles est dû à des actions similaires sur un ensemble de données, dans un tableau ou une matrice par exemple. C'est la caractéristique des machines SIMD (cf §1.2.1) ou encore synchrone. Ce parallélisme est mis parfois en évidence dans les langages, en général ce sont des extensions des langages de haut niveau comme FORTRAN, PASCAL /Wir 71/, APL /Ive 62/, ALGOL /Wij 69/. Un langage spécialisé a été défini pour la machine tableau Illiac IV, il proposait des opérations vectorielles ou matricielles, il s'agit de TRANQUIL /Kuc 69/.

Bien que ces méthodes permettent l'expression du parallélisme présent dans les programmes ou dans les algorithmes, elles demandent au programmeur un effort particulier pour le détecter et l'exprimer. Cette opération n'est pas toujours simple au moment de la programmation, c'est pourquoi, dans la plupart des cas, ce parallélisme est recherché dans des programmes existants à l'aide d'outils appropriés tels que VESTA pour FORTRAN ...
Ces outils peuvent être adaptés à la recherche du parallélisme dans d'autres langages.

- Une autre possibilité de parallélisme intéressant, et de mise en oeuvre assez simple, est la redondance pour l'obtention d'un système extrêment fiable. C'est le cas de certains systèmes temps réels qui peuvent avoir des propriétés absolues car ils mettent en cause la sécurité des systèmes. Par exemple les systèmes commandant des centrales nucléaires ou de futurs systèmes multiprocesseurs embarqués nécessitent une très grande fiabilité.

- Un parallélisme nécessitant aussi des machines extrêmement puissantes est un parallélisme de nature asynchrone. Ce parallélisme est par exemple implicitement mis en oeuvre dans les langages comme LISP ou PROLOG. Ainsi la programmation logique offre de nombreuses opportunités d'exploitation du parallélisme, comme le parallélisme "OU", lançant simultanément plusieurs tentatives d'unification avec un même terme, et le parallélisme "ET", lançant simultanément la résolution de plusieurs buts d'une même proposition. S. Conery /CK 81/ présente une interprétation parallèle des programmes logique et démonbre de nombreux cas de traitement parallèle.

L'examen du projet FGCS des Japonnais permet de constater que leurs efforts sont principalement concentrés sur ce type de problèmes. Il semble que l'application du parallélisme à la programmation logique contribuera au développement de l'intelligence artificielle dont la place sera de plus en plus importante dans les prochaines années.

- Il existe également une forme d'expression du parallélisme basée sur les relations entre les données elles mêmes, ce type de fonctionnement correspond aux systèmes dirigés par des données ("Data flow").

Un programme dirigé par les données est un ensemble partiellement ordonné d'opérations qui deviennent exécutables dès que leurs opérandes sont disponibles. Une telle représentation est différente des représentations classiques, car le programmeur n'a pas un contrôle général, mais un contrôle local.

Les langages spécialisés au fonctionnement dirigé par les données sont des traductions de graphe de dépendance.

Les modèles de machine "data flow" permettent une détection du parallélisme à l'exécution et s'attachent à proposer des outils d'exploitation "maximum" du parallélisme.

1.2. LES MODES DE FONCTIONNEMENT

Pour prendre en compte la description du fonctionnement d'une machine, Toursel dans /Tou 79/, propose la notion de cellule composée de l'instruction et des données opérandes, et définit le fonctionnement d'un ordinateur comme la présentation successive, par l'unité de commande, d'une cellule à l'unité d'exécution. Le schéma de commande est défini comme la détermination à partir d'une cellule activée, des cellules suivantes.

Toursel distingue 8 schémas de commande répartis en 3 groupes :

- les machines à commande centrée sur les instructions ;

- les machines à commande centrée sur les données ;

- les machines à commande mixte où la détermination de la cellule suivante à exécuter dépend des emplacements de l'instruction et des données origines.

Dans ce chapitre les systèmes appartenant aux deux premiers groupes seront examinés les machines du 3ème groupe donnant lieu à des architectures tout à fait particulière, appelée s-architecture, ne sont pas abordées.

1.2.1. Fonctionnement centré sur les instructions

Une étude bibliographique des multiprocesseurs a été réalisée par Douspis et Rocacher /Dr 83/ au Centre de recherche BULL, aussi cette partie ne sera-t-elle que rapidement abordées dans cette thèse afin de pouvoir réaliser un examen et une critique plus approfondis des machines cadencées par les données. Les machines MIMD/SIMD/Data flow citées par la suite ont été étudiées dans /Dr 83/.

De nombreuses approches de classification des architectures parallèles sont possibles, celle de Flynn /Fly 72/ bien qu'imprécise à l'avantage d'être connue de beaucoup.

Quatre classes sont distinguées :

- SISD : (Single instruction stream, single data stream). Les machines SISD sont des machines classiques de type VON NEUMAN qui exécutent les instructions séquentiellement.

- SIMD : (Single instruction stream, Multiple data stream) Les machines vecteurs, tableaux et les processeurs associatifs se situent dans cette classe. Elles correspondent à des machines qui avec un flot d'instructions manipulent plusieurs flots de données, comme les vecteurs. Illiac IV /Bar 68/, STARAN /Bat 74/, CRAY 1 /rus 78/, MPP /Bat 80/, ISIS /Tim 84/ sont des machines SIMD.

- MISD : (Multiple instruction stream, single data stream) Les machines pipe-line sont parfois rangées dans cette classe. En effet un pipe-line est composé de plusieurs unités de calcul travaillant sur une entrée, la sortie d'une unité devenant l'entrée de la suivante /Cor 78/.

- MIMD : (Multiple instruction stream, multiple data stream) Cette classe de machine est très générale et regroupe les systèmes comprenant plusieurs processeurs reliés entre eux, chacun exécutant son propre flot d'instructions. La tendance actuelle semble être au développement de telles machines. C.m.m.p. /WB 72/, CM* /Ful 78/, CHOPP / SBK 77/, EPOS /Mac 79/, HM2P /SLS 82/, ETH /Bue 82/ SMS 201 /KK 79/, FTDCL /Nel 80, SN 80/, la machine de Heidelberg /MD 81/, MUPI /ELT 82/ sont des multiprocesseurs entrant dans cette catégorie.

Cette classification apparait fort imprécise ainsi des architectures entièrement différentes peuvent se trouver dans une même classe.
En outre il est difficile de ranger dans une classe particulière les machines pipe-lines qui sont classées en SIMD, SISD, MISD selon les interprétations.
Toutefois, les définitions suivantes sont usuellement reconnues pour les calculateurs scientifiques :

- SIMD : soumission d'un nombre élevé d'unités de traitement a une commande unique. En général, la notion de SIMD est semblable aux notions de machines centralisée et de synchronisme.

- MIMD : ensemble de processus pouvant communiquer et obéissant à leur propre flot d'instructions. En général la notion de MIMD est semblable à celle de machine décentralisée et d'asynchronisme.

De récentes études sur le parallélisme et les recherches sur l'architecture des réseaux d'interconnexions (cf chapitre 4) ont fait apparaitre un nouveau type de système qui semble particulièrement prometteur, ce sont les machines partitionnables. Ces machines ont la capacité de pouvoir se scinder en plusieurs sous machines indépendantes. En général ces propriétés sont principalement dûes à la capacité des réseaux de communication à pouvoir se décomposer en sous-réseaux. Selon les architectures, les sous-machines obtenues peuvent appartenir à des classes différentes de la classification de Flym. Ainsi par extension trois nouvelles classes peuvent être distinguées :

- MSIMD : (Multiple SIMD) C'est le cas des machines SIMD partitionnables dont les sous-machines obtenues ont un fonctionnement SIMD. Dans le projet initial de la machine Illiac IV les 256 processeurs élémentaires pouvaient être partitionnés en 1, 2 ou 4 blocs de respectivement 256, 128, 64 processeurs.

- MIMD-SIMD : les machines qui rentrent dans cette classe sont celles qui peuvent fonctionner soit en mode MIMD, soit en mode SIMD. FMP /LB 80, Bar 81/ un projet de multimicroprocesseur de haute performance développé pour la NASA et MP/C /AG 82/ sont des machines de ce type.

- MIMD-MSIMD : de telles architectures offrent une grande souplesse d'utilisation et permettent un fonctionnement en mode MIMD et multi-machines SIMD. Le projet le plus caractéristique de cette classe est probablement le multiprocesseur dynamiquement reconfigurable PASM (PArtionnable SIMD/MIMD), /Sie 81, MS 82, KS 81/. D'autres architectures comme ZMOB /Rie 81/ multiprocesseurs construit autour d'une mémoire circulante, ARRAY/Net /UTL 81/ ou PUMPS /BDH 81/ ont été étudiées avec ce type d'objectif : la reconfigurabilité.

De nouvelles machines non orientées "data flow" mais capable d'en supporter l'implémentation sont étudiées actuellement. Ces machines peuvent être classées dans une catégorie très générale MIMD/MSIMD/Data flow.

L'ULTRACOMPUTER étudié par Gottich et al /GS 82, Got 83/ est un projet ambitieux cherchant à connecter 4K processeurs de type oméga.

Un projet intéressant CEDAR développé par Gajski, Kuck et al /GK 83/ a pour but de montrer que les super-ordinateurs peuvent être "general purpose" et facile à utiliser. Ce projet parait cohérent mais hélas peu de documentation est accessible actuellement.

La reconfigurabilité est également une des caractéristiques de TRAC /Sej 80, JDL 81, JB 82/ qui peut exécuter tout type d'exécution parallèle : asynchrone MIMD, asynchrone pipe-line, asynchrone data flow, vectoriel SIMD, synchrone.

Enfin HEP construit par Denelcor est une machine MIMD disposant de mécanismes data flow (présentation de HEP chez BULL le 9 Février 84). Cette machine tourne et déjà plusieurs exemplaires ont été vendus. Une version plus performante HEP 2 devrait sortir fin 85 et devrait fournir une gamme de puissance de 250 MIPS, 500 MIPS, 750 MIPS, ... selon les configurations.

De récentes études montrent que la technique MIMD est efficace pour le traitement de nombreuses applications. C'est l'un des enseignements des expériences de Burroughs qui après avoir construit BSP, une machine SIMD, a adopté une technique MIMD pour leur machine FMP, à la suite d'une étude faite pour la NASA montrant que pour obtenir un débit élevé l'approche vecteur n'était pas nécessaire.

Dans tous les cas les concepteurs prennent maintenant comme objectifs la construction de machines très souples d'utilisation

capables de supporter diverses fonctionnalités MIMD, SIMD voire multi-SIMD et data flow. Il apparait que la qualité principale des multi-microprocesseurs est leur adaptabilité autant au niveau potentiel de calcul, capacité à traiter un grand nombre de problèmes, qu'au niveau fonctionnement. Les caractéristiques suivantes : extensibilité, modularité, flexibilité, tolérance aux pannes, fiabilité apparaissent souvent dans la littérature sur les multiprocesseurs.

Quant aux performances elles sont difficiles à prévoir. Les concepteurs les plus prudents n'avancent pas de chiffres sur la performance théorique de leur machine. Ceux qui annoncent des performances sont souvent très optimistes et la réalité est parfois fortement différente des prévisions. Bernhard /Ber 82/ cite par exemple l'ILLIAC IV conçue pour exécuter 200 Mfops et qui n'a pas réussi à dépasser 25 MFlops. Il présente différentes spéculations, pour un calcul réaliste de la vitesse des multiprocesseurs, basées sur l'expérience de nombreux chercheurs. Les prévisions les plus pessimistes prévoient une vitesse équivalente à Log2N processeurs pour une machines de N processeurs en parallèle, d'autres chercheurs pensent que la vitesse maximum qui puisse être atteinte est 0,3 N.

1.2.2. Fonctionnement centré sur les données

Dennis /Den 74/ fut le pionnier en ce qui concerne les études des architectures non VON NEUMAN que sont les machines dirigées par les données (ou "data flow"), il est à l'origine des études sur les graphes de dépendance.

Les programmes dirigés par les données sont habituellement décrits en terme de graphes orientés, utilisés pour décrire le flot des données entre les instructions.

Un graphe représentant un programme dirigé par les données (fig 2) est composé :

- de noeuds qui représentent les opérations,

- d'arcs qui représentent le chemin des données entre les opérateurs,

- les marques représentant les valeurs des données. Celles-ci se propagent le long des arcs.

fig 2 : graphe représentant l'opération z=(x+y)*(x-z)

La règle de base d'exécution des programmes dirigés par les données est :

- un opérateur est exécutable quand tous ses opérandes d'entrée sont disponibles.

L'exécution consiste à traiter les données d'entrée et à produire un résultat. Ainsi, quand un opérateur a été exécuté, ses données d'entrées sont disponibles, un nouvel ensemble devra être calculé pour qu'il devienne à nouveau exécutable.

Sur l'exemple de la figure 2, l'application de cette règle implique que les opérations '+' et '-' peuvent s'exécuter en parallèle et la production du résultat de chacune d'elles autorise l'exécution de '*'.

Les principaux types de noeuds sont :

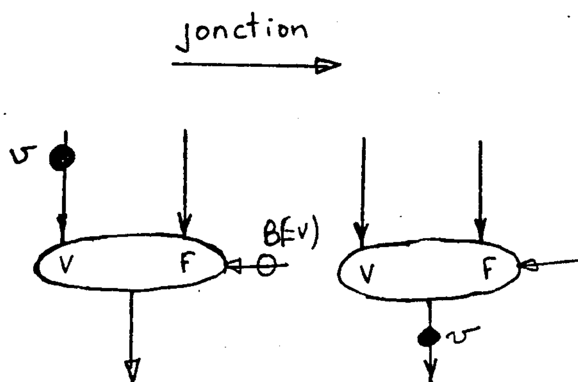
- les noeuds de chainage permettant dans un graphe, de dupliquer une marque donnée (représentée par 1 point)

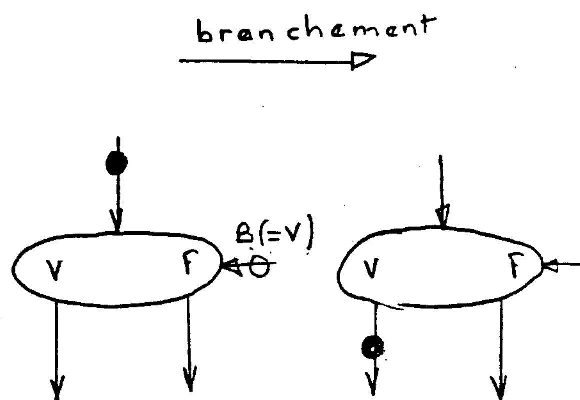
duplication

- les noeuds acteurs qui produisent une marque résultat à partir de leurs marques d'entrée

opérateur f

prédicat





la marque est propagée si le booléen en B est vrai

la marque est propagée sur la branche correspondant à la valeur du booléen B

Le principe "data flow" est basé sur le déplacement des valeurs des données et sur des mécanismes de contrôle parallèle de celles-ci (contrôle des marques). Les marques données sont passées directement de l'instruction producteur à l'instruction consommateur.

Ce concept ne fait donc pu apparaître la notion de compteur ordinal, ni même de cellule mémoire partagée pour le stockage des variables, à la différence du principe "control flow" (dirigé par les instructions).

Afin d'éviter l'accumulation de marques sur certains arcs et les confusions quant à leurs origines, comme lors de l'exécution des itérations d'une boucle, cinq stratégies peuvent être adoptées :

1) L'utilisation de graphe réentrant est interdite. Chaque itération doit être décrite par un graphe. Cette solution nécessite un espace de stockage du programme assez important.

D'autre part, pour les boucles dynamiques où le nombre d'itérations n'est connu qu'à l'exécution il faut générer du code dynamiquement.

2) L'utilisation de graphe réentrant est autorisée mais une itération ne peut démarrer que lorsque la précédente est terminée. Cette solution ne permet pas le parallélisme entre itérations et nécessite des instructions spéciales pour déterminer la fin d'une instruction (figure 3).

3) L'utilisation du graphe data flow limite le nombre de

marque à une au plus par arc. Ainsi un noeud acteur n'est activé que lorsque les marques sont présentes sur tous ses arcs d'entrée et qu'il n'y a pas de marque sur l'arc de sortie.

4) Les marques transportent leur numéro d'itération, elles sont dites colorées. Un noeud est exécutable que si toutes ses entrées ont la même couleur.

5) Les marques sont stockées sur les arcs dans leur ordre d'arrivée (stream).

Exemple de programme : Calcul du centième nombre de la suite Fibonacci

```
(x,y) := (1,1) ;

i:=3 ;

WHILE i = 100 DO

   BEGIN

     (x,y) := (y, x+y) ;

     i := i+1 ;

     END ;

     OUT PUT Y ;
```

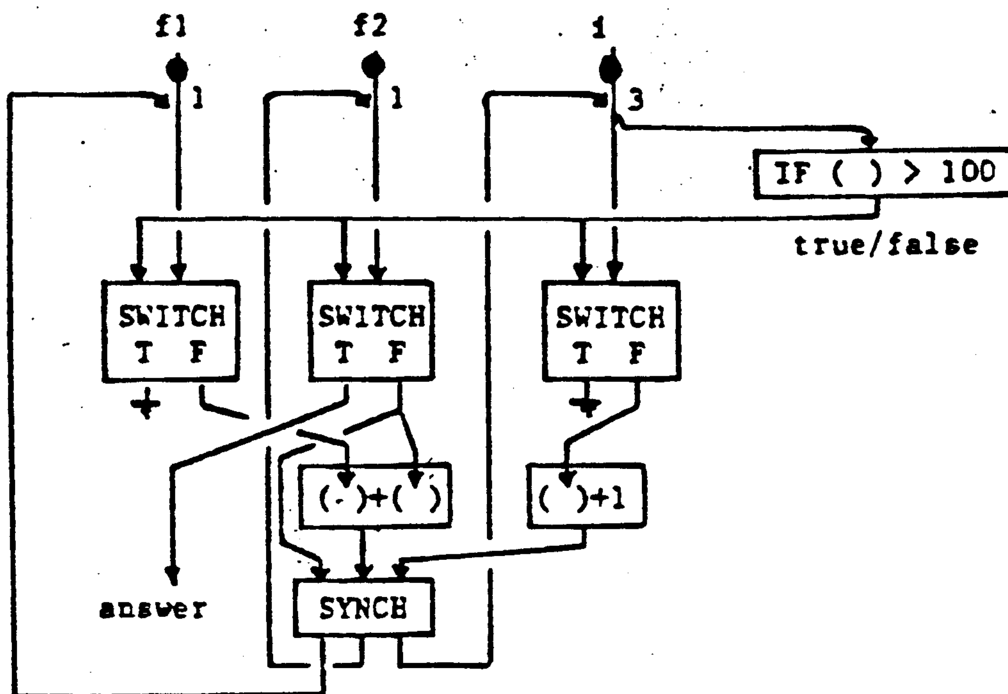


fig 3 : une représentation graphique d'un programme data flow

Les langages dirigés par les données sont en général basés sur le principe de l'assignation unique qui impose qu'une variable ne peut recevoir, au plus, qu'une valeur durant toute l'exécution du programme. Ils sont la traduction des graphes de dépendances.

Dans la suite de ce chapitre est présenté, un petit nombre de travaux sur les machines dirigées par les données (leur architecture et leur langage associé) qui ont paru les plus caractéristiques, puis une critique en est faite afin de tirer un enseignement des expériences passées.

1.2.2.1. Travaux du M.I.T.

Les travaux du M.I.T. sur le "data flow" sont particulièrement significatifs et ont été la base de nombreux autres projets. Leurs études ont couvert les domaines des graphes de dépendance /Den 74/, de l'architecture /Den 79/ et le langage d'assignation unique VAL /Ack 79/.

L'organisation des programmes dans cette machine est typiquement data flow et correspond à un graphe de dépendance comportant des noeuds et des arcs. La propagation des marques dans le graphe se fait suivant le principe 1 décrit précédemment, c'est à dire qu'un noeud peut être activé si ses marques d'entrée sont présentés et si l'arc de sortie ne porte aucune marque.

Des marques de contrôle permettent de signaler aux

opérateurs producteur que leur arc de sortie a été libéré de sa marque donnée.

L'organisation de la machine est déterminée autour d'une stratégie de communication par paquet entre les différentes unités.

La communication par paquet consiste en la formation de paquets de travail ou cellules circulant d'unité en unité d'une manière asynchrone (fig 4). Un programme est alors vu comme un ensemble de cellules indépendantes pouvant être activées, divisées ou fusionnées. Pour l'exploitation du parallélisme la communication par paquet est une stratégie simple, lorsqu'une unité a terminé son travail, elle cherche dans l'ensemble des travaux une nouvelle tâche à traiter.

Des unités de même nature peuvent travailler parallèlement (comme p processeurs de traitement) et des unités de nature différente peuvent exécuter simultanément des travaux sur des cellules en circulation (exécution pipe-line des traitements).

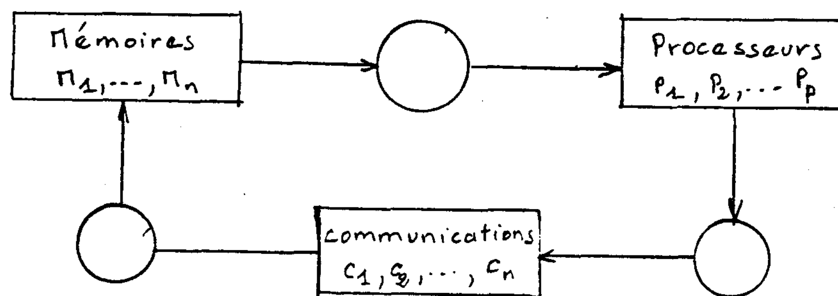


fig 4 : Machine organisée autour d'un principe de communication par paquet

La machine du M.I.T. comporte cinq unités :

- une unité mémoire où sont stockées des cellules contenant les instructions et les opérandes

- une unité de traitement constituée de n processeurs

- un réseau arbitre distribuant aux processeurs les cellules exécutables contenu dans l'unité mémoire

- un réseau de contrôle fournissant à l'unité mémoire

les cellules provenant de l'unité de traitement

- un réseau de distribution fournissant les données résultats, issues de l'unité de traitement, aux cellules en attente dans l'unité mémoire.

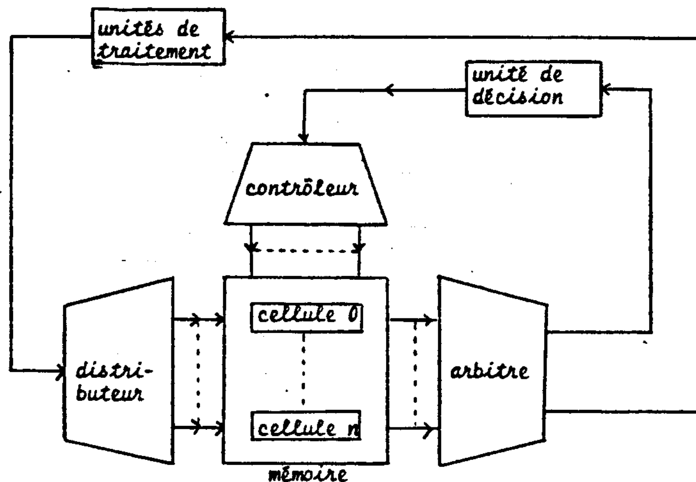


fig 5 : Structure de base de la machine dirigée par les données du M.I.T.

Une cellule est constituée d'une instruction matérialisée par un registre, contenant le code de l'instruction et les adresses des registres, auxquelles le résultat de l'exécution doit être communiqué, et les opérandes matérialisés par deux registres.

Une cellule devient exécutable quand elle a reçu ses opérandes et son signal d'autorisation d'exécution dû à la libération de ses marques de sorties.

1.2.2.2. Travaux du C.E.R.T.

Le système LAU /Com 76, Syr 77, Com 79/ est une machine dirigée par les données devant exécuter le langage de haut niveau LAU (Langage d'assignation unique).

LAU est un langage évolué d'assignation unique, non basé sur une représentation graphique, facilement utilisable parce que très proche des langages de programmation habituels, et offrant au moins les mêmes outils que FORTRAN, ALGOL, PASCAL.

Le principe de fonctionnement est du type 2 décrit

précédemment, c'est à dire qu'une itération ne démarre pas avant la précédente. Ainsi dans une boucle un objet est décomposé en trois :

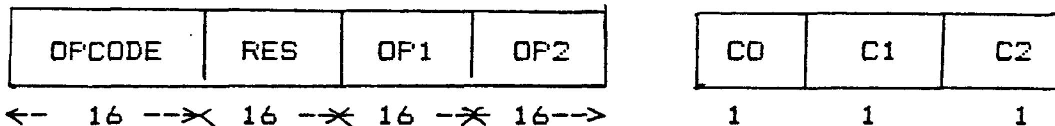
- OLD X : dénote la valeur de X à l'itération précédente

- NEW X : dénote la valeur de X à l'itération courante

- OUT X : dénote la valeur finale par X.

Cependant, une instruction EXPAND permet l'exécution parallèle de certaines itérations, elle peut être comparée au DO PARALLEL. Le code obtenu est statique avec plusieurs copies du corps de boucle.

Une instruction machine comprend le code instruction, trois adresses, l'une pour les résultats et les deux autres pour les opérandes, et 3 bits de contrôle C0, C1, C2.

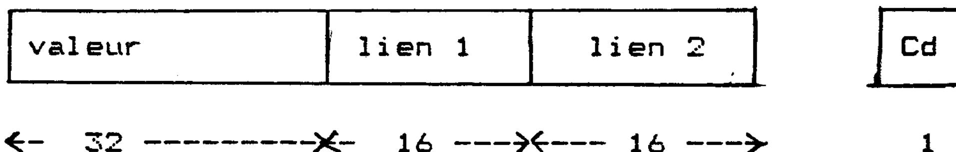


En fait, l'organisation des programmes est basée sur un des principes du "control flow" qui détermine le passage des données par des cellules mémoire partagées, accédées par les adresses contenues dans les instructions.

Associé à chaque instruction les bits C0, C1, C2 contrôlent et synchronisent l'exécution. C1 et C2 indiquent que les opérandes respectifs 1 et 2 ont été calculés. C0 indique si l'instruction est dans un contexte autorisant son exécution (exemple : Instruction conditionnelle).

Une instruction est exécutable si C0C1C2 = 111

Un opérande est un ensemble de trois entités :



- un champ valeur de l'opérande de 32 bits ;

- deux champs liens de chainage de 16 bits chacun, contenant l'adresse vers les instructions qui utilisent l'opérande ;

- un bit de contrôle C0 indiquant si l'opérande a été "produit" ou non.

L'architecture de la machine LAU est organisée sur un type de communication par paquet.

Elle est constituée de trois éléments fonctionnels (mémoire centrale, unité de commande, unité d'exécution) reliés par des bus (fig 6).

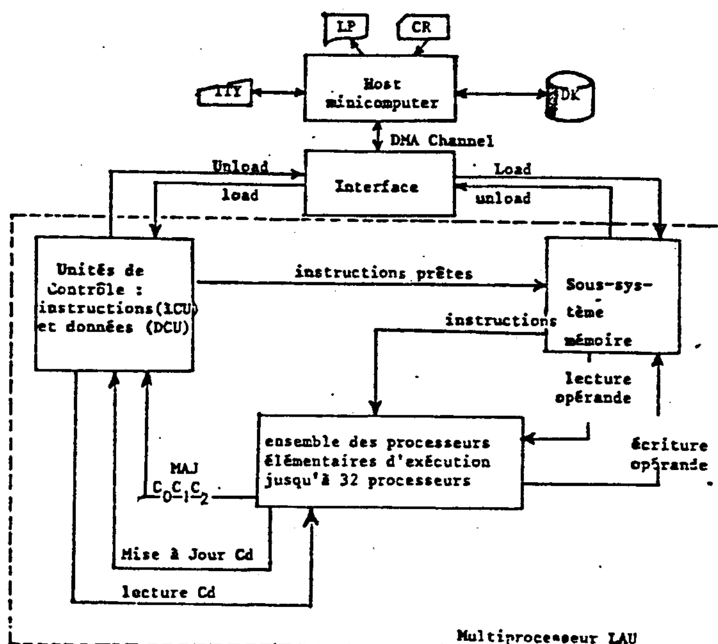


fig 6 : Diagramme fonctionnel du multiprocesseur LAU

Le fonctionnement général est le suivant :

- une instruction est déclarée prête (C0C1C2=111) par l'unité de contrôle instruction (ICU) qui envoie un ordre de lecture d'instruction en mémoire centrale ;

- l'instruction est lue en mémoire et envoyée dans une file des instructions prêtes. Dès qu'un processeur d'exécution termine son instruction en cours, une instruction lui est envoyée par l'intermédiaire du bus instruction ;

- le processeur d'exécution lit alors, s'il y a lieu, les valeurs des opérandes d'entrée en mémoire centrale, exécute l'opération, écrit le résultat en mémoire centrale, récupère en retour deux champs pointeurs de l'opérande écrit, met à jour le bit Cd dans l'unité de contrôle des données DCU à l'adresse de l'opérande, et met à jour dans ICU, les bits C1 C2 des adresses indiquées par les pointeurs de l'opérande résultat.

L'ensemble du projet LAU a duré de 1973 à 1979.

1.2.2.3. Les travaux de l'Université d'Irvine

La machine d'Irvine est basée sur un langage de haut niveau ID (Irvine Data Flow) /Arv 78, Arv 80/. ID est un langage a assignation unique conçu sur la notion d'expressions dont les quatres plus importantes sont les blocs, les expressions conditionnelles, les boucles et les procédures.

Les variables manipulées sont simples, et correspondent alors à une marque dans le graphe dirigée par les données, ou "stream" (en chaines), et correspondent à une suite ordonnée de marques sur un arc.

Dans ID le nommage unique des marques est effectué grâce à un identificateur, ou nom d'activité destination, comportant quatre champs (P/N/A/I) :

- P est un nom de bloc identifiant une procédure ou une boucle particulière ;

- N est un numéro d'instruction dans le bloc ;

- A est un numéro d'itération pour les boucles ;

- I est un nom de contexte identifiant l'activité concernée (procédure ou boucle).

Les variables "stream" permettent d'entrer ou de sortir d'un opérateur un nombre potentiellement infini de marques tout en utilisant un seul arc

De plus ceci permet d'introduire un niveau d'asynchronisme puisque les marques entrent ou sortent d'un opérateur de façon complètement asynchrone.

Pour ce langage un interpréteur de "débrouillement" (Unraveling Interpretor) a été développé. Il génère le maximum de "tâches" parallèles à partir d'un programme ID et permette l'exécution simultanée d'appels distincts de la même procédure et l'expansion automatique de boucle.

L'architecture de la machine ID, basée sur un concept de communication par paquet, est la suivante :

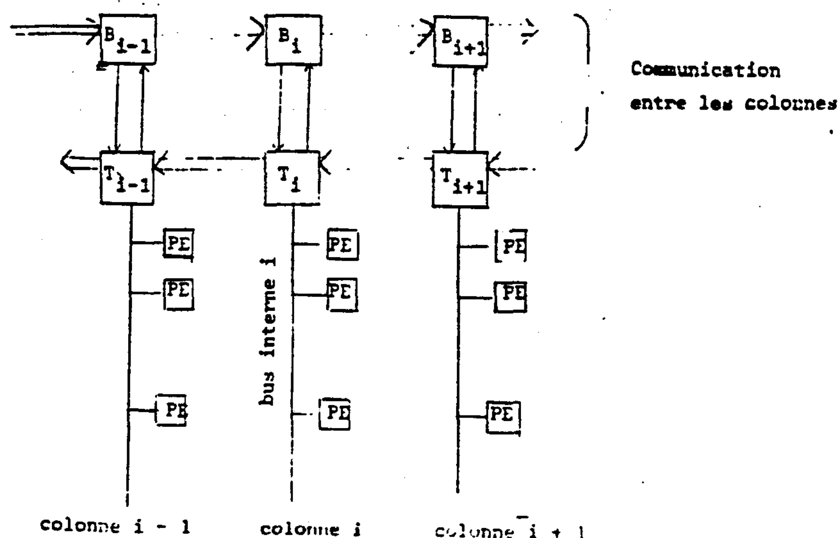


fig 7 : Architecture d'IRVINE

Les marques résultats circulent dans le système de communication à la recherche d'un PE libre ou qui les attend. La circulation des résultats (valeurs, destinations) se fait comme suit :

- le résultat produit dans la colonne i parvient en Ti

- Ti, qui contient une table d'adressage associative, effectue le travail suivant :

  <u>Si</u> destination dans sa table <u>alors</u> envoie du résultat sur le bus local

  <u>sinon</u> passer le résultat à Ti-1

  <u>Fsi</u>

Les liaisons Ti-Bi réalisent un système de communication en anneau capable d'être partitionné en plusieurs sous-anneaux, constituant ainsi plusieurs sous domaines physiques.

Afin de diminuer les communications au travers de l'ensemble de la machine, la répartition des travaux se fait en essayant de faire correspondre, d'une part les sous domaines physiques aux sous-domaines logiques (procédure), donc de localiser les communications, et d'autre part de faire correspondre un ensemble d'instructions à chaque PE.

Un processeur élémentaire comporte cinq parties :

1) une unité d'entrée qui reçoit les marques venant du système de communication.

2) une unité de recherche d'opérande qui réunit les marques en ensembles consommables par les instructions.

3) une unité de recherche des instructions qui recherche les instructions exécutables dans la mémoire locale.

4) une unité d'exécution.

5) une unité de sortie qui envoie les marques produites sur le système de communication.

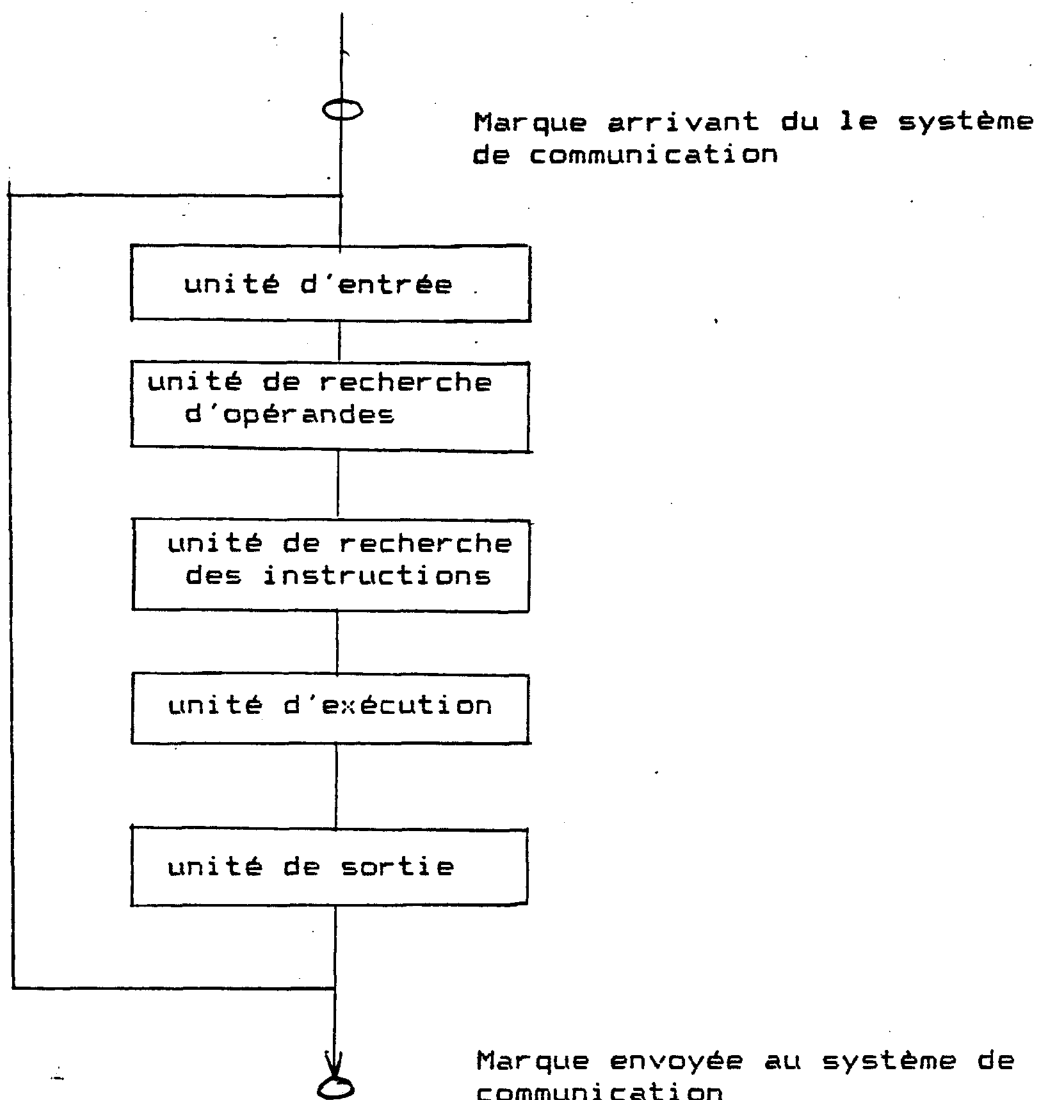


fig 8 : Le processeur élémentaire de la machine d'Irvine

1.2.2.4. Les travaux de l'Université de Lille

Le projet MAUD (Machine d'Assignation Unique Dynamique) développé à l'Université de Lille /Lec 79, Pet 81/ se démarque des projets précédents dans la mesure où les concepts de traitement dirigé par les données et d'assignation unique ne sont pas appliqués à l'échelle des instructions mais à celle d'un ensemble d'instructions, appelé "Bloc".

Ainsi pour être exécutable dans MAUD un programme doit être décomposé en blocs. Un bloc est constitué d'un ensemble

d'instructions, des objets d'entrée qui conditionnent l'exécution du bloc et des objets de sortie qui sont calculés par les instructions et utilisés comme objets d'entrée par d'autre blocs.

Un bloc ne devient prêt à exécuter ou exécutable que lorsque tous ses objets d'entrée ont reçu une valeur et que celles-ci lui ont été transmises, sinon il est en attente.

La particularité de ce modèle est que les blocs sont des ensembles d'instructions séquentielles classiques exécutées selon un shéma "Von Neumann" sur des processeurs élémentaires classiques.

Ce modèle appelé "modèle statique" ne permet pas d'exploiter le parallélisme d'une boucle dynamique. Pour répondre a ce besoin MP Lecouffe a introduit des structures de contrôle du type "FORK - JOIN" permettant d'expliciter l'exécution d'un autre bloc en parallèle, ou l'attente de la fin d'un autre bloc.

Ainsi une opération "EXECBLOC" permet d'ajouter un nouveau bloc exécutable à l'ensemble X des blocs exécutables, un modèle de ce nouveau bloc se trouvant dans une bibliothèque de bloc. Une opération "ATTENDRE" permet de placer un bloc B1, ayant demandé la création d'un bloc B2 (par un EXECBLOC), dans l'ensemble des blocs en attente, jusqu'à ce que B2 est terminé son exécution, c'est à dire qu'il ait produit ses objets de sortie.

MAUD est composée :

- d'opérateurs d'exécution Pi

- d'un opérateur de mise à jour MAJ, chargé de faire les communications d'objets entre les différents blocs à traiter

- d'un opérateur de construction de blocs, appelé constructeur, dont la fonction est de construire un nouveau bloc à partir des informations fournies par les opérateurs d'exécution lors de l'exécution des opérations EXECBLOC.

Ces opérateurs n'ont aucun moyen de communication directe entre eux.

Pendant l'exécution d'un programme, MAUD doit gérer :

- un ensemble X de blocs exécutables ;

- un ensemble A de blocs en attente ;

- un ensemble S de domaines de sortie ;

- un ensemble D de demandes d'exécution produites lors de l'exécution des opérations EXECBLOC ;

- un ensemble de blocs modèles, appelé bibliothèque.

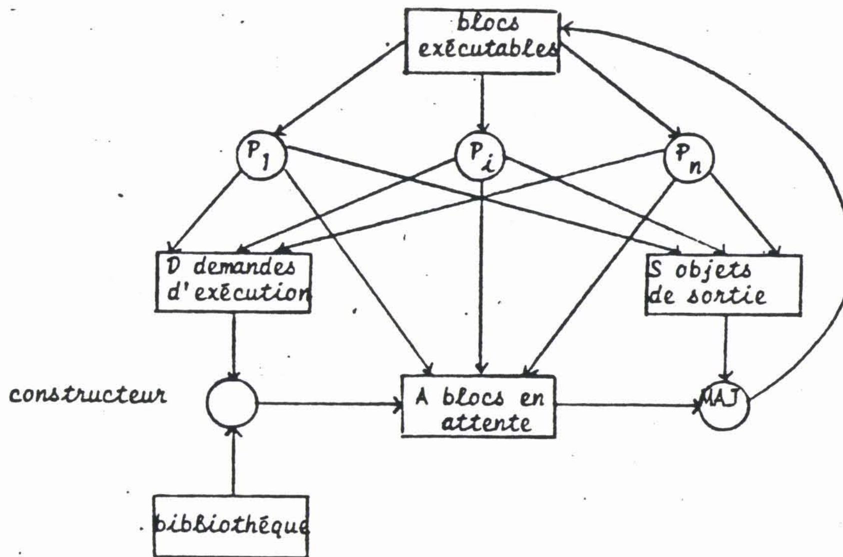


fig 9 : Description fonctionnelle de MAUD

Pour mettre en oeuvre ce modèle, une architecture originale a été conçue. Elle est basée sur une mémoire circulante sous forme d'anneau servant de moyen de stockage et de communication entre 32 processeurs d'exécution, un processeur de MISE A JOUR et un processeur CONSTRUCTEUR.

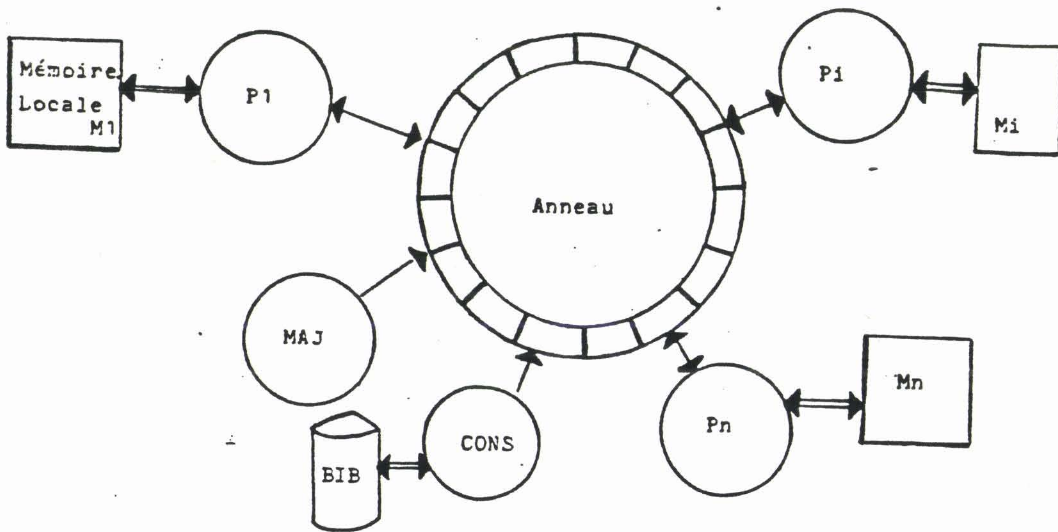
Les caractéristiques de la mémoire circulante confère à la machine des propriétés intéressantes :

- un moyen de stockage et communication à accès multiples : simultanés

- l'accès associatif des objets de sorties

- une gestion de la mémoire et des accès simples

- un système facilement extensible utilisant des composants classiques.



Pi : Processeur de calcul
MAJ : Processeur de mise à jour
CONS : Processeur Constructeurs
BIB : Bibliothèque de bloc

fig 10 : Architecture de MAUD

1.2.2.5. Autres Projets

La recherche dans le domaine des machines dirigées par les données s'est rapidement développée aux Etats-Unis, au Japon et en Europe. Ainsi Texas Instrument a fait des recherches dans ce domaine avec le "Distributed Data Processor" (DDP) /Con 79, Joh 79/ dont l'une des caractéristiques est qu'il est programmable en FORTRAN.

La machine construite à l'Utah /Dav 78/ : DMM1 exploite un parallélisme statique grâce à une architecture récursive flexible, reconfigurable et facilement extensible.

Les études de l'Université de Manchester ont débuté en 1975 et ont abouti à la construction d'une machine comportant une vingtaine de processeurs /Wat 79/ ayant une configuration classique circulaire des machines data flow à communication par paquet.

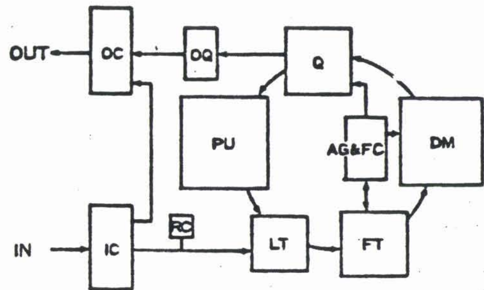
Au contraire des systèmes précédents qui sont spécifiquement data flow, l'objectif de l'université de Newcastle est de définir une machine de type MIMD universelle qui utilise les meilleures caractéristiques des deux organisations : flot de contrôle et flot de données /Trel 78/. L'architecture de cette machine est conçue autour d'une mémoire circulante.

L'intérêt pour les machines data flow est relancé depuis que les Japonais ont annoncé dans leur projet FGSC leurs intentions d'investigations dans le domaine des machines dirigées par les données.

Ainsi Nec System a annoncé récemment (février 84) un processeur VLSI ayant une architecture data flow adaptée au

NEC 7281

traitement d'image : IPP (Image Pipeline Processor) /Nuk 84/. Cinq unités fonctionnelles de ce processeur sont connectées en anneau de façon à réaliser la circulation et le traitement pipeline des marques de données (token).



OC	: Output Controller, Controls output data.
IC	: Input Controller, Controls input data token and judges whether or not the incoming data token should be sent to the internal ring.
LT	: Link Table (128 words x 16 bits). Stores destination address the token.
FT	: Function Table (64 words x 40 bits). Stores instruction parameters.
DM	: Data Memory (512 words x 18 bits). Stores constants or temporary data.
Q	: Queue (32 levels x 60 bits, 16 levels x 60 bits). FIFO queue.
PU	: Processing Unit. Processing unit which executes logical, arithmetic and bit operations.
OQ	: Out Queue (8 levels x 32 bits). FIFO queue for output.
AG&FC	: Address Generation and Flow Control. Address : generation for DM and flow control.
RC	: Refresh Controller. Refresh Control for internal DRAMs.

fig 11 : IPP block diagram.

En version multi processeurs, les processeurs IPP peuvent être connectés en cascade (fig 12), ou en anneau (fig 13), la sortie de l'un alimentant l'entrée de l'autre :

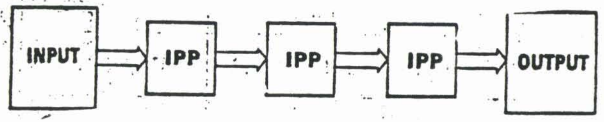


fig 12 : Config. en cascade

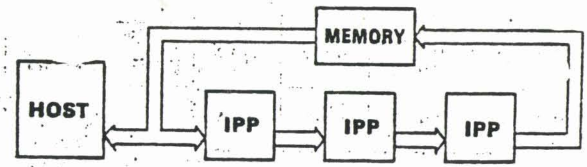


fig 13 : config. en anneau

1.3. CRITIQUES DES MACHINES DIRIGEES PAR LES DONNEES

Si l'approche machine dirigée par les données est un concept intéressant qui semble adapté au parallélime celui-ci reste dans le domaine de la recherche et est éloigné des problèmes industriels et commerciaux. Faire une machine qui tourne plus ou moins bien ne suffit pas, encore faut il qu'elle ait des caractéristiques qui permettent de la vendre et, à ce titre, les machines data flow ne sont pas toujours réalistes.

Ce paragraphe est un examen critique des systèmes dirigées par les données permettant, en conclusion de ce chapitre, de définir un cahier des charges en vue de la conception d'un système d'exploitation du parallélisme.

1.3.1 Point de vue logiciel

Le problème majeur des systèmes dirigés par les données est probablement logiciel. Les langages data flow sont ils commercialisables ? Chacun sait que le marché du logiciel est dominé par un certain conservatisme et des problèmes de compatibilité ; il est impensable de pas tenir compte du "parc logiciel" existant et à venir. Les caractéristiques certainement intéressantes des langages data flow sont elles suffisantes pour pouvoir imposer une classe totalement nouvelle de langage de programmation ? En outre, qu'elle est la productivité d'un programmeur en langage data flow ?

Les nouveaux langages tentent d'améliorer les problèmes de communication homme-machine, les contrôles et la clarté d'expression. Ainsi, par exemple, le regroupement d'instructions proches en exécution et des objets qu'elles utilisent correspond à la tendance actuelle de structuration des programmes dont les mérites ne sont plus à vanter. Il ne faut pas perdre de vue que les coûts de développement, de mise au point et d'évolution du logiciel sont énormes (Bry 83).

En 1978, le coût de développement, de mise au point et de maintenance du logiciel employé par le gouvernement des Etats-Unis était estimé à environ 8 milliards de dollars, incluant environ 4 milliards pour la défense ; l'US-Air-Force à elle seule a dépensé environ 80 à 90 % du coût total d'acquisition de systèmes informatiques pour le logiciel, contre 15 % en 1955.

Réduire les coûts de développement, de mise au point et d'évolution du logiciel sont des objectifs essentiels auxquelles les nouvelles machines doivent répondre.

Il semble que la notion d'efficacité doit, à l'avenir, ne plus s'exprimer uniquement en termes de puissance de calcul.

D'autres aspects deviennent importants, voire indispensables :

- la fiabilité des moyens fournis dans l'aide à la construction et à la mise au point des programmes

- la facilité d'utilisation de ces moyens et de l'ensemble des commandes des systèmes d'exploitation.

Or, en ce qui concerne les machines dirigées par les données, ces notions semblaient être oubliées, développer et mettre au point un programme dirigé par les données ne parait pas très aisé (cela est peut être un problème de jeunesse).

1.3.2 Point de vue matériel

Si le degré de perfectionnement, de complexité et d'intégration des microprocesseurs croit de façon impressionnante, comme l'indique la figure n° 14, le coût de conception, d'implantation, modification et de vérification de ces circuits tend à croitre également rapidement (fig n°15). Bien entendu la courbe n° 14 présente le coût en homme/années des cricuits conçus en logique anarchique et il est certain que ces coûts sont réduits de façon importante grâce à des méthodologies de conception de VLSI visant la structuration et l'utilisation d'outils de conception assistée par ordinateur (Anc 83), cependant le coût de conception et de développement de tel circuit reste relativement élevé.

Pour qu'un microprocesseur soit rentable il faut donc, une fois qu'il a été conçu, pouvoir en produire un très grand nombre pour les vendre à un prix raisonnable. C'est ce qui fait que le microprocesseur 6800 de Motorola se trouve actuellement sur le marché pour une centaine de francs et que bientôt le "68000" sera vendu dans cet ordre de prix.

Le problème est alors de savoir si les processeurs data flow auront un impact suffisant sur le marché pour pouvoir être intéressant commercialement. Il se peut qu'un marché s'ouvre pour eux avec les nouveaux types de langage logique utilisés en Intelligence Artificielle (tel Prolog) ou certains traitements spécialisés, comme le traitement d'image, mais cela reste incertain sachant que dans la prochaine décade, des ordinateurs Von Neumann très puissants seront disponibles sur un boitier et qu'il sera peut être préférable de les utiliser, même si ils ne sont pas entièrement bien adaptés aux applications logiques.

Ainsi concevoir une machine parallèle purement data flow représente un risque commercial car le rapport performance/prix,

qui théoriquement pour une machine multiprocesseur de type Von Neuman devrait être une des caractéristiques poussant à développer ce genre d'architecture, est difficilement prévisible.

La construction d'une machine parallèle à l'aide de circuits compatibles avec une ligne de produits, permet d'améliorer son prix de revient sa fiabilité et sa capacité d'évolution. Le choix d'un microprocesseur sera en particulier déterminé par sa souplesse d'adaptation à un environnement multiprocesseur car l'expérience des problèmes rencontrés dans CM* a montré que le développement d'interfaces d'un processeur conventionel avec l'extérieur était très important puisque la taille des interfaces devenait comparable aux processeurs utilisés.

Actuellement, les microprocesseurs sont plus souples d'utilisation et les efforts des constructeurs se portent sur :

- l'adjonction de mémoire et de capacité d'entrée-sortie intégré sur le "chip",

- l'accroissement des capacités et du débit par l'extension de la longueur des mots, de la capacité d'adressage, du jeu d'instruction, de la vitesse,

- l'amélioration de leur possibilité d'intégration dans un environnement multiprocesseurs par l'adjonction de lignes de contrôle et surtout de la micro-programmation,

ce qui perrmet d'envisager leur utilisation dans la conception d'un système parallèle.

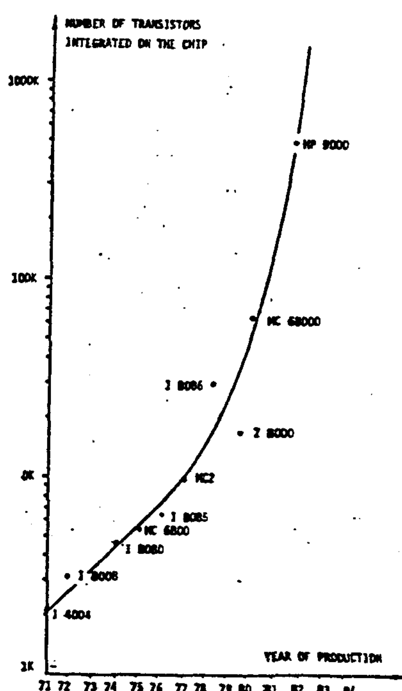


fig 14 : Evolution du nombre de transistors par chip

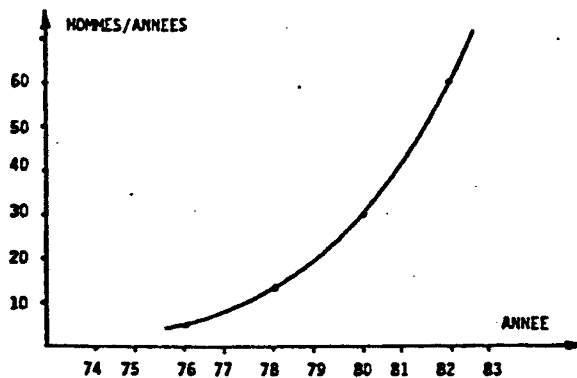


fig 15 : Coût en homme/années des conceptions, implantation, modification et vérification de la logique anarchique

1.3.3 Point de vue technique

Un des points les plus importants à retenir des enseignements des expériences en ce qui concernent les machines data flow, que la réalisation du prototype LAU met en évidence, est que l'objectif de l'exploitation du parallélisme "maximum", c'est à dire au niveau des instructions machine, n'est pas satisfaisant.

Un principe fondamental de l'exploitation du parallélisme est que la "granularité" du parallélisme, c'est à dire la taille des traitements exécutés en parallèle, doit être fonction des traitements dûs aux mécanismes de contrôle qu'il a fallu implanter pour pouvoir exploiter ce parallélisme.

Ainsi par exemple, si un processus P peut être décomposé en deux traitements notés TE1 et TE2 exécutables parallèlement, moyennant pour chacun d'eux l'exécution de traitements de contrôle TC1 et TC2 afin de pouvoir exploiter ce parallélisme, il faut au moins que max (TEi + TCi) soit inférieur au traitement séquentiel de P pour pouvoir espérer un gain de rendement dû à l'exploitation du parallélisme.

En règle générale, il faut que TEi/TCi soit le plus grand possible, c'est à dire qu'il faut que les traitements de contrôle soient petits et/ou que TEi soit proportionnellement grand.

En outre, un des facteurs importants d'accélération de l'exécution d'un programme dans une machine de Von Neuman est la possibilité de recouvrement (pipeline) dans l'exécution des instructions. Ainsi, les différentes phases d'une instruction, décodage d'instruction et d'adresses d'opérandes, recherche d'opérandes, exécution, peuvent être réalisées avec un recouvrement sur plusieurs instructions.
Par contre dans le modèle data flow les instructions se trouvant sur un même chemin du graphe de dépendance sont exécutées sans recouvrement puisque l'exécution de l'une dépend de la fin du traitement de la précédente.

Bien que le concept de data flow ne fasse pas intervenir la

notion de mémoire et d'adressage puisque la valeur des données est passée d'un opérateur au suivant, il faut constater que généralement les échanges se font par l'intermédiaire d'une mémoire commune.

Ainsi les échanges se font de mémoire à mémoire ce qui exclut les échanges locaux registre à registre qui sont effectués dans les machines classiques et qui sont un facteur d'accélération . En outre, les flots d'informations à communiquer, et les problèmes de goulots d'étranglement qu'ils provoquent, restent importants sur les machines "data flow". La communication des valeurs d'opérateur à opérateur est pourtant une caractéristique intéressante. Il semble que les machines systoliques (Kun 82) et les machines data flow ont des propriétés proches et que, dans l'avenir, leur structure pourrait être semblable, les configurations en cascade ou anneau du multi processeur IPP des japonais semblent confirmer cette tendance.

Enfin, les modèles dirigés par les données détectent à l'exécution ce qui peut s'exécuter en parallèle dans un traitement, cela engendre des flots de contrôle importants (cf LAU) exécutés dynamiquement et qui ne sont pas toujours justifiés comme dans les cas de suite d'instructions typiquement séquentielles dont les dépendances peuvent être détectées statiquement au moment de la compilation.

1.4 CONCLUSION

Le concept de machine dirigée par les données apporte des contributions intéressantes pour l'exploitation du parallélisme, cependant comme cela a été montré dans ce chapitre, il engendre des langages et des architectures particulières qui sont peu réalistes ou qui seront spécialisées à certaines applications comme le traitement d'images ou la programmation logique.

Dans un environnement industriel il s'agit de tenir compte des contraintes commerciales qui sont :

- logicielles : une machine doit être capable de reprendre les applications existantes et être suffisamment souple pour supporter efficacement de nouveaux langages et de nouveaux environnements de programmation. En outre la facilité de programmation et d'utilisation (en particulier de mise au point des programmes) sont des facteurs importants, maintenant mis en évidence dans des langages évolués, qu'il ne faut pas négliger.

- matérielles : outre un bon rapport performance/prix la machine de demain devra avoir des caractéristiques de fiabilité, de tolérance aux pannes, de modularité et d'extensibilité qui sont généralement (et théoriquement) attribuées aux machines multiprocesseurs.

Ainsi s'appuyant sur ses expériences logicielles (conception d'un transformateur automatique de programmes : VESTA, intérêts pour les langages évolués : ADA) et matérielles (étude d'une Machine Langage Ada : MLI) l'équipe de Transformation de Programmes du Centre de Recherche de BULL dont Claude Renvoise (chef de Service), Pierre Douspis et Daniel Rocacher, ont été amenés à proposer un système d'exploitation du parallélisme répondant au cahier des charges défini précédemment.

Le modèle d'expression du parallélisme, appelé Data Flow Macroscopique, basé sur la transformation de programme permet :

- la parallélisation des applications existantes et futures,

- de ne pas imposer un langage spécifique,

- d'exploiter un parallélisme raisonnable permettant d'équilibrer le rapport traitement de contrôle, nécessaire à l'exploitation du parallélisme, sur traitement effectif (c'est à dire calcul à exécuter),

- de reporter à la pré-compilation une grande partie des contrôles (statiques) dûes à la dépendance des

variables lues et modifiées dans un programme,

- de proposer un outil simple, très souple adaptable autant aux logiciels, qu'aux machines cibles,

- de permettre l'utilisation de mécanismes d'accélération classique lors de l'exécution d'un programme.

Une architecture Multi-Opérateurs Séquentiels, appelé AMOS, adaptée à ce modèle a été envisagée, elle présente :

- les nombreux avantages des architectures multiprocesseurs de type MIMD,

- un coût de conception et de réalisation faible grâce à l'utilisation de processeurs classiques de type Von Neuman.

Ainsi le Data Flow Macroscopique et AMOS en alliant performances (au sens général non limité à la rapidité de calcul) et adaptabilité (reprises d'applications existantes ...) sont des produits qui dans le cadre d'un milieu industriel ont les atouts permettant un impact commercial dans le domaine des machines haut de gamme.

CHAPITRE 2

LE MODELE DATA FLOW MACROSCOPIQUE

2.0 - INTRODUCTION

Claude Renvoise (Ren 83) a défini un modèle d'expression du parallélisme appelé "Data Flow Macroscopique". Ce modèle permet de représenter un programme sous la forme d'un ensemble de "macro-instructions" dont les enchainements sont déterminés par un graphe de dépendance. Une macro-instruction est un terme général permettant de désigner un ensemble d'instructions du programme, appelé traitement effectif et/ou un ensemble d'instructions de contrôle, appelé traitement de contrôle. Elle peut être considérée comme une instruction généralisée ayant plusieurs opérandes et le contrôle associé permettant l'enchainement des macro-instructions.

Le découpage du programme est fait dans une phase précédent l'exécution, grâce aux techniques de détection automatique du parallélisme basée sur l'analyse de la dépendance des données lues et modifiées par les instructions. Un analyseur de ce type, VESTA, a été réalisé au centre de recherche BULL (Ren 80) (CRS 83).

Dans sa version actuelle VESTA (VEctoriseur de programmes scientifiques par Traduction Automatique) est un outil paramétrable qui, à partir d'un programme FORTRAN, produit un programme vectorisé ou parallélisé suivant la nature de la machine cible.

Les fonctionnalités implantées dans VESTA permettent d'effectuer les transformations de programmes suivantes :

* Eclatement de boucle
* Permutation de boucles
* Expansion de scalaire
* Sortie de tests constants
* Substitution d'expression
* Reconstitution de tests
* Opérations pyramidales
* Création de temporaire

Elles sont utilisées pour supprimer certaines relations de dépendances.

Les techniques de détection du parallélisme ne sont pas développer dans cette thèse car elles sont détaillées dans les documents décrivant le fonctionnement de VESTA.

De cette analyse dirigée par les données il résulte un programme parallélisé multiséquentiel, c'est à dire composé de parties de programmes séquentiels, ou macro instructions, dont l'enchainement parallèle ou non des traitements est matérialisé par un graphe de dépendance dont chaque noeud représente une macro instruction et chaque liaison matérialise la dépendance d'une macro instruction fille à une macro instruction mère.

Ainsi, un outil logiciel de transformation de programme adapté permet l'exécution sur un mode "piloté par les données" d'ensembles d'instructions. Pour ces raisons le modèle correspondant est appelé "Data Flow Macroscopique".

Lorsque la transformation de programme, par le compilateur ou un outil approprié, définit complètement le graphe d'exécution, c'est à dire que l'utilisation de graphe réentrant n'est pas admise et que le programme est entièrement décrit par son graphe, le modèle est dit statique et les macro instructions sont appelées Atomes. Le paragraphe 2.1. décrit les principes de fonctionnement de ce modèle simplifié.

Afin de limiter ce caractère statique qui ne permet pas d'exprimer le caractère dynamique propre à certaines structures de programmation, un nouveau concept, la notion de molécule, a été introduit au chapitre 2.2.

Ce modèle, décrit par Claude Renvoise, permet un transfert transparent à l'utilisateur d'applications existantes, programmées dans des langages de haut niveau classiques, en programmes parallélisés exécutables sur une machine multi processeurs où chaque processeur fonctionne suivant le principe classique de Von Neuman.
Dans (Ren 83) il est également montré les applications du modèle aux structures classiques des langages de programmation (IF, case, do, appel de procédure) ; elles sont rappelées au paragraphe 2.3.

2.1 LE MODELE STATIQUE

Un programme, ayant été décomposé grâce à un détecteur automatique de parallélisme, se présente sous la forme d'un ensemble de n macro instructions dont l'ordre d'exécution n'est plus la séquentialité mais correspond à une structure plus complexe qui a été déterminée lors de l'analyse du programme à partir des variables lues et modifiées par les différentes macro instructions.

Cet ordre peut être matérialisé par un graphe de dépendance dont chaque noeud représente une macro instruction et où les liaisons entre noeuds matérialisent les relations de dépendances entre macro instructions, impliquant de cette façon un ordre d'exécution.

Ainsi, par exemple, un tel graphe peut avoir la forme suivante :

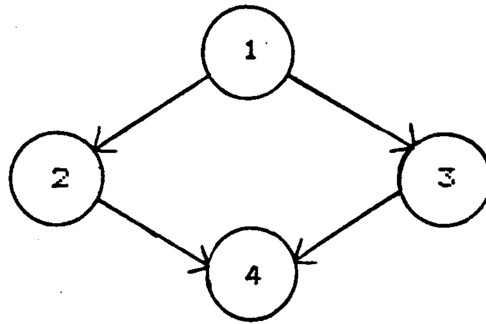


fig 1 : Graphe de dépendance

Dans ce cas la fin de l'exécution de la macro instruction 1 conditionne celle de 2 et 3 qui elles-mêmes conditionnent l'exécution de la macro instruction 4.

Lorsque le programme est décomposé en un ensemble de n macro instructions et que le graphe décrivant leurs enchainements est complètement défini lors de la parallélisation automatique, c'est à dire que l'utilisation de graphe réentrant n'est pas admise, par exemple toutes les itérations d'une boucle doivent être décrites par celui-ci, alors le modèle (simplifié) est dit statique et les macro instructions sont appelées atomes. Cette notion de graphe statique (et par suite de graphe dynamique) est définie par M.P Lecouffe (Lec 79).

2.1.1 Concept d'atome

L'ensemble des atomes d'un sous-programme est un sous ensemble de l'ensemble des macro instructions composant ce programme.

Un <u>atome</u> correspond à une suite d'instructions, exécutables de façon séquentielle selon le principe de Von Neumann, qui réalisent d'une part les calculs nécessaires à la production des résultats du programme, appelés <u>traitement effectif</u>, et d'autre part les calculs nécessaires à l'enchainement des macro instructions composant le programme, appelés <u>traitement de contrôle</u>.

La partie traitement effectif est un ensemble séquentiel d'instructions séquentielle exécuté par un opérateur de traitement.

La partie traitement de contrôle peut être exécutée par un opérateur de traitement sépcialisé ou par le même opérateur que celui utilisé pour le traitement effectif.

2.1.2 Traitement effectif

Le traitement effectif est donc une partie de programme séquentiel qui est composé d'instructions élémentaires (assignations, opérations de lecture et d'écriture, opérations arithmétiques, ...) ou d'instructions plus complexes (instructions de choix, boucles, appels de procédures, mécanismes de rendez-vous, ...).

Les données manipulées se trouvent dans une mémoire dont il n'est pas nécessaire de préciser la nature et l'organisation pour définir le modèle, ainsi cette mémoire peut être aussi bien globale que locale, il s'agira plus tard de déterminer une architecture adéquate.

Le découpage du programme, grâce à l'analyse des relations de dépendance basées sur la production et la consommation des variables, permet de garantir la cohérence des données lues et modifiées par chaque atome. Ainsi, lorsqu'un atome est exécutable toutes ses données sont prêtes à être utilisées.

La taille du traitement effectif d'un atome n'est pas limitée et peut être du niveau de l'instruction, d'un ensemble d'instructions, d'une tâche voire même d'un programme. Cependant, une bonne exploitation du parallélisme implique une taille minimum du traitement effectif.

En effet, il faut que la part du traitement effectif soit relativement suffisante par rapport au traitement de contrôle associé afin de ne pas passer trop de temps en calcul de contrôle ce qui, en dégradant les performances, limiterait l'intérêt de l'exploitation du parallélisme inhérent à chaque programme.

La souplesse du modèle data flow macroscopique permet d'adapter la taille des traitements effectifs à la taille des traitements de contrôle nécessaires à l'exploitation du parallélisme dégagé. L'originalité du modèle provient de cette adaptabilité à exprimer un parallélisme dont l'exploitation soit "rentable", c'est à dire que le temps passé pour exécuter des contrôles est ajusté au temps passé pour exécuter les traitements effectifs afin d'obtenir un gain de performance.

Cette notion de taille de traitements exécutables en parallèle est appelé "granularité" du parallélisme.

L'expérience de LAU montre que le parallélisme exploité au niveau des instructions n'est pas efficace car le contrôle de chaque instruction est proportionnellement trop important par rapport au travail effectué, la "granularité" du parallélisme exploité est trop faible.

Ainsi les caractères essentiels du modèle Data Flow Macroscopique sont son adaptabilité et sa souplesse pour exprimer un parallélisme qui soit fonction des traitements de contrôle associés, des caractéristiques de la machine cible, des caractéristiques des programmes (classe, priorité, taille mémoire,...). C'est l'outil parallélisant automatiquement un programme qui détermine la taille des traitements effectifs en fonction de ces paramètres.

2.1.3 Mécanisme d'enchainement d'exécution des atomes

Ce paragraphe détermine comment un atome devient exécutable.

Un atome est exécutable quand les traitements de l'ensemble des atomes dont il dépend sont terminés. Sur un graphe de dépendance un atome est exécutable lorsque les atomes qui le précèdent directement (atomes pères) ont terminé leur exécution.

Pour exprimer ce système de relations de dépendance entre atomes et déterminer si ils sont exécutables ou non, une technique, issue des commandes gardées selon Dijkstra (Dij 75), est utilisée. Elle détermine l'exécution d'une expression grâce à la valeur d'une expression booléenne associée (la garde). La méthode utilisée pour le modèle data flow macroscopique et semblable mais au lieu d'une variable booléenne comme garde, une variable entière associée à chaque atome, appelée <u>compteur de dépendance</u>, est utilisée.

La valeur du compteur de dépendance indique le nombre de macro instructions non exécutées dont dépend l'atome associé. Ainsi lorsqu'un compteur de dépendance a une valeur n, cela signifie qu'il existe n macro instructions mères de l'atome associé dont l'exécution n'a pas été terminée.

Lorsqu'un compteur de dépendance a une valeur nulle l'atome correspondant est exécutable car toutes les macro instructions dont il dépendait ont été exécutées.

Un atome doit donc pouvoir signaler la fin de son traitement aux atomes dépendant directement de lui, cette action se fait grâce à des opérations de décrémentation des compteurs de dépendance des atomes fils. La décrémentation d'un compteur correspond alors à la libération d'une dépendance et est réalisée par une instruction du traitement de contrôle de l'atome ayant terminé son traitement effectif.

Ce mécanisme permet de décrire l'ordonnancement d'exécution des atomes impliqué par le graphe de dépendance. Les valeurs initiales des compteurs de dépendance correspondant au nombre d'arcs entrant d'un noeud, le nombre d'opération de décrémentation à réaliser par le traitement de contrôle d'un atome est égal au nombre d'arcs sortant du noeud correspondant.

Ainsi pour l'exemple de la figure 1, les compteurs de dépendance des atomes A1, A2, A3, A4 auront initialement les valeurs respectives 0, 1, 1, 2 et les traitements de contrôle associés auront respectivement 2, 1, 1, 0 opérations de décrémentation sur les compteurs de dépendance des atomes cibles (atteints par les arcs).

Chaque atome du programme ainsi restructuré, s'apparente à un processus correspondant à l'unité de programme exécutable par un processeur. Les différents processus composant le programme s'exécutent de façon asynchrone, le mécanisme d'enchainement est non déterministe. Le fonctionnement d'une telle machine est donc de type MIMD selon la classification de Flynn.

2.1.4 Traitement de contrôle-mot d'état

Le traitement de contrôle d'un atome assure l'enchainement des traitements effectifs et la gestion de toutes les informations de contrôle nécessaires au déroulement de ceux-ci. Les traitements de contrôle ont pour effet de modifier des structures de données système qui caractérisent l'état courant d'exécution de la machine. Ils sont exécutés à la fin de chaque traitement effectif.

Cette structure de données, appelée "vecteur d'état" est composée d'un ensemble de "mots d'état".

Un <u>mot d'état</u> est un n-uplet qui contient un ensemble d'informations caractérisant un atome. Ces informations sont construites pendant l'analyse du programme ou correspondent, pour certaines d'entre-elles, à des passages de paramètres. Aussi la valeur de certaines informations varient au cours de l'exécution du programme.

Un mot d'état comprend les champs suivants :

. CE (champ conditions d'exécution) qui décrit l'état d'exécution de l'atome.

Il comprend :

- un booléen CE.BOOL indiquant si il a été exécuté ou non

- le compteur de dépendance CE.COMPTEUR indiquant, la dépendance de l'atome par rapport à son environnement. Sa valeur initiale est calculée à la compilation, elle est ensuite décrémentée au fur et à mesure des terminaisons des atomes qui conditionnent directement l'exécution de l'atome associé. Sa valeur est nulle lorsque l'atome est exécutable et non significative lorsqu'il a été exécuté.

La notion de compteur permet d'obtenir une bonne compaction de l'information qu'il représente, mais elle implique que les décrémentations provoquées par divers atomes soient exécutées séquentiellement. Des mécanismes, matériels d'accélération de recherche mémoire (du type mémoire associative) peuvent être utilisés pour minimiser la perte de temps que cela peut engendrer (cf. chapitre IV). D'autres réalisations de compteurs ont été envisagées, du type suite de bits adressables en parallèle, mais elles ont été rejetées car elles nécessitent une technologie trop coûteuse pour des avantages peu significatifs.

du traitement effectif correspondant à l'atome

vont travailler les instructions. Il peut être un pointeur vers une table qui regroupe les descripteurs de segments de données

tions de contrôle associé à l'atome.

Il faut noter que les ensembles des instructions effectives et des instructions de contrôle peuvent être réunis et désignés par un même champ lorsque les traitements effectifs et de contrôle d'un atome sont exécutés par un même opérateur.

Le modèle statique ainsi définit peut être représenté par le schéma synoptique du principe de fonctionnement :

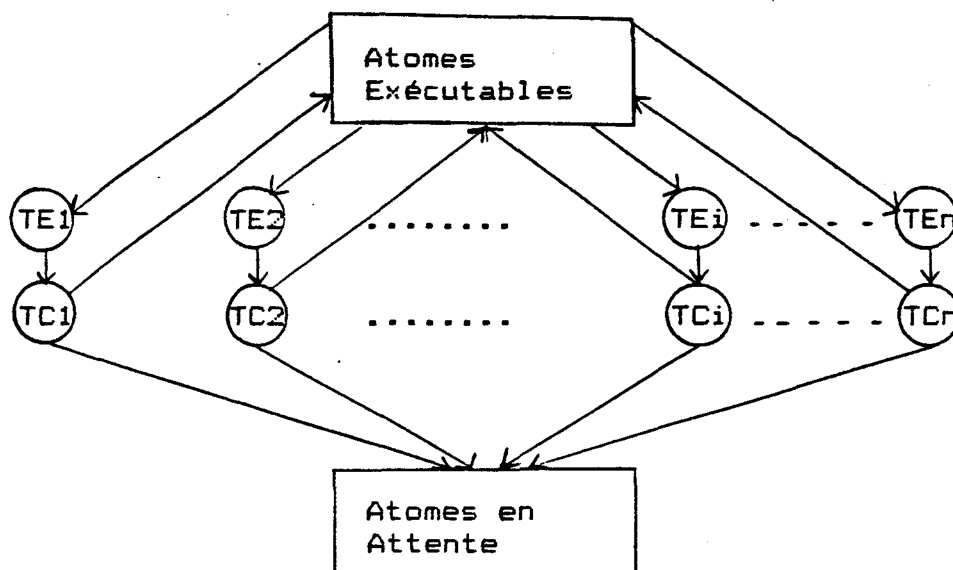


fig 2 : Modèle statique : traitement effectif et traitement de contrôle réalisés par des opérateurs différents

La convention notationnelle représentant un opérateur de traitement par un cercle et un ensemble d'atomes par un rectangle est utilisée.

Dans ce cas de figure les opérateurs de traitement effectif et les opérateurs de traitement de contrôle sont différents. Il est tout à fait possible d'envisager une architecture où un traitement effectif et le traitement de contrôle associé sont exécutés par le même opérateur de traitement, le schéma synoptique est alors le suivant :

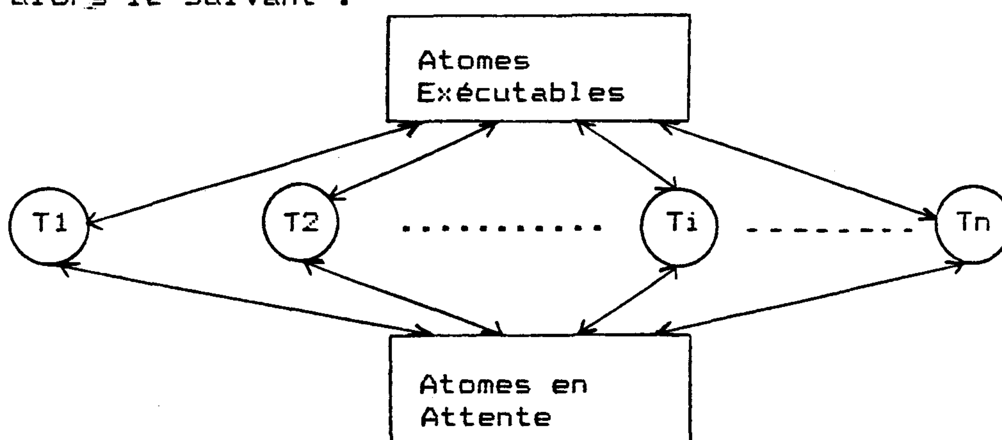


fig 3 : Modèle statique : Traitement effectif et traitement de contrôle d'un atome réalisés par le même opérateur

2.2 LE MODELE DYNAMIQUE

Le caractère statique du modèle ne permet pas une bonne adaptation de tous les décompositions aux possibilités d'exécution parallèle. Tout est figé dès la phase de décomposition du programme en atomes, il est donc nécessaire d'introduire un système permettant d'introduire un comportement dynamique au niveau de l'exécution du programme parallélisé.

La nécessité d'introduire un caractère dynamique dans le modèle data flow macroscopique est d'abord montrée puis la notion de molécule introduite par Claude Renvoise, permettant de répondre à ce problème est exposée.

2.2.1 Nécessité d'un modèle dynamique

Exemple 1 :

Soit à effectuer le traitement suivant, où F est une fonction :

```
S1 ;

for I  in  1 ..  N   loop

T(I) := F(I);

end loop ;

S2,
```

Dans ce cas, avec le modèle statique, la boucle ne peut être que contenue entièrement dans un atome, car la variable d'itération n'est pas connu statiquement et par conséquent il n'est pas possible de décomposer la boucle en plusieurs atomes exécutables en parallèle même si la fonction F(I) permet une exécution indépendante de chaque itération.

exemple 2 :

Soit une procédure ADA ayant un paramètre d'entrée N. Ce paramètre est défini dynamiquement et sa valeur est comme lors de l'exécution du programme. Cette procédure déclare un tableau de tâches T, de taille N. Ces tâches sont activées juste après l'élaboration de cette partie déclarative, c'est à dire juste après le mot réservé begin qui la suit.

```
procédure  P(N : INTEGER) is

type T is task

        entry  E  ;

end     T  ;

type  A  is array  (1 .. N) of  T  ;

A1 : A  ;

begin ---  activation de N tâches de type
                T : A1(1), A1(2), ..., A1(N)
  .
  .

end  T  ;
```

Le modèle statique défini précédemment, ne peut pas prendre en compte un programme de ce type car la taille du tableau A1 est inconnue à la compilation et, par conséquent, il est impossible de savoir combien de tâches de type T il faut activer. Le graphe de dépendance d'un tel programme n'est pas entièrement défini à la compilation.

exemple 3 :

Soit la procédure récursive P :

```
procédure  P(N : interger) is
     :
     :
begin
  if  N = 1  then
  else  P(N-1) ;
    :
end if ;

end  P ;
```

Le nombre d'appel de la procédure P n'est pas connu à la compilation, il dépend de la variable N qui est calculée à l'exécution. Ainsi il n'est pas possible d'inclure à l'intérieur du programme appelant les instructions du sous-programme appelé et une méthode, consistant à introduire les atomes qui représentent le sous-programme parmi les atomes du programme appelant, n'est pas ici satisfaisante.

La notion d'appel de sous-programme doit être maintenue au niveau du modèle data flow macroscopique.

2.2.2 Concept de molécule

Les exemples étudiés au paragraphe précédent montrent que le concept d'atome, pour exprimer le parallélisme inhérent aux programmes n'est pas suffisant car certains caractères dynamiques de la programmation classique ne peuvent être pris en compte. Pour pouvoir exploiter le parallélisme des parties de programme définies dynamiquement il faut pouvoir utiliser des structures de contrôle qui permettent de les générer durant l'exécution.

Les parties de programme ainsi générées ne sont pas forcément simples et réduites à une séquence d'instructions correspondant à un atome mais peuvent être plusieurs séquences d'instructions, exécutables ou non en parallèle, correspondant à un ensemble de macro instructions dont les enchainements sont décrits par un graphe de dépendance.

Une molécule est la matérialisation du regroupement conceptuel d'un ensemble d'atomes et/ou de molécules.

Une molécule est une macro-instruction qui ne contient pas de traitement effectif mais seulement des traitements de contrôle, et par suite son exécution a un effet uniquement sur le vecteur d'état de la machine.

De la même façon que pour un atome, une molécule est décrite par un mot d'état dont la valeur du compteur de dépendance indique si elle est exécutable ou non, contenant les caractéristiques et éventuellement les paramètres nécessaires à r/n exécution.

Le concept de molécule ainsi défini permet donc :

1) D'exécuter des programmes non exécutables dans un modèle statique tels ceux comportant une boucle dont la variable d'itération n'est connue qu'à l'exécution.

La structure de contrôle molécule permet de générer dynamiquement des macro-instructions. Ainsi dans le cas précis d'une boucle dont les itérations sont exécutables en parallèle, une molécule génère n macro instructions.

2) De matérialiser un regroupement logique de macro instructions pouvant correspondre par exemple à une unité de compilation (c'est à dire un sous-programme, une tâche, un sous programme de package) ou de façon générale à un bloc de macro instructions issu de la décomposition structurée des programmes.

3) De permettre l'utilisation de bibliothèques de molécules.

4) D'éviter l'écriture de plusieurs macro instructions identiques.

Par la suite le terme <u>macro instructions</u> désigne de façon générale un atome ou une molécule.

Les traitements de contrôle assurant les enchainements des macro instructions peuvent avoir un niveau sémantique plus ou moins élevé et permettent d'envisager plusieurs modes de contrôle selon les types d'enchainement à réaliser. Ainsi diverses méthodes peuvent être utilisées pour le traitement parallèle des boucles, elles sont plus ou moins adaptées à la configuration du programme, voire de la machine. Il appartient au compilateur ou à l'outil de parallélisation de choisir le mode de contrôle le meilleur, c'est à dire celui permettant l'exploitation d'un parallélisme efficace en fonction des structures traitées.

Il faut à nouveau noter la souplesse du modèle data flow macroscopique qui n'impose pas une expression figée du parallélisme selon les structures traitées.

En outre, la sémantique des traitements de contrôle permet d'adapter l'expression du parallélisme aux mécanismes de programmation ce qui facilite d'une part la reprise des applications existantes, et d'autre part les possibilités d'évolution.

Le modèle dynamique peut être représenté par les schémas synoptiques du principe de fonctionnement :

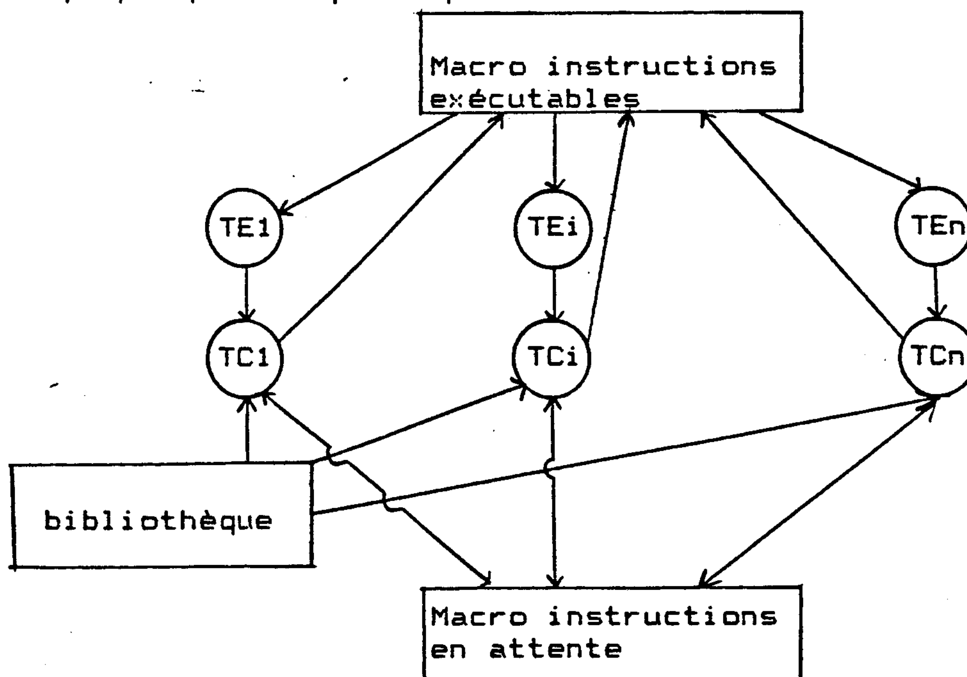


fig 4 : Modèle dynamique où les opérateurs de traitement effectif et de traitement de contrôle sont différents

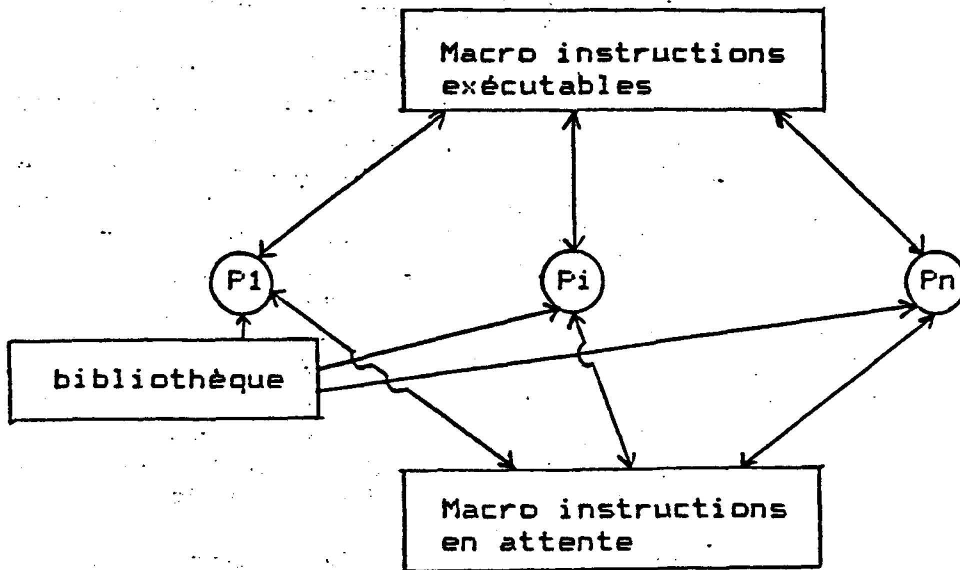


fig 4 : Modèle dynamique où les opérateurs de traitement effectif et de traitement de contrôle sont confondus

2.3. Traitements des constructions classiques des langages de programmation

Dans ce paragraphe le modèle data flow macroscopique qui vient d'être décrit est appliqué aux différentes constructions classiquement rencontrées dans les langages de programmation.

Cela conduit à proposer des traitements de contrôle pilotant l'enchainement des macro instructions du programme, qui soient de véritables programmes comprenant des structures de contrôles riches (IF, CASE, LOOP, ...).

La prise en compte des instructions de choix (IF, CASE), des instructions de boucle, des sous-programmes récursifs ou réentrants est explicitée. Parfois plusieurs modes de contrôle des enchainements sont présentés, ils sont plus ou moins bien adaptés selon les configurations des programmes et il appartient au compilateur ou à l'outil approprié de choisir le mode de contrôle le meilleur en fonction des structures traitées.

Les instructions étudiées par la suite peuvent apparaître ou non au niveau du modèle data flow macroscopique selon que le taux de parallélisme qu'il en ressort est suffisant et que son exploitation garantisse une diminution du temps global d'exécution du programme. Ainsi une instruction peut être complètement cachée dans le traitement effectif d'un atome ou

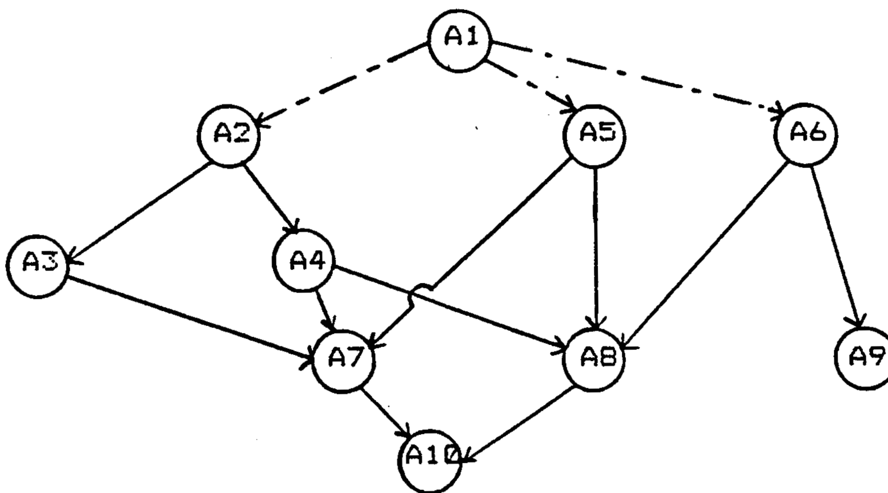
être prise en compte par le traitement de contrôle d'une macro instruction selon le choix de l'outil de parallélisation.

2.3.1 Prise en compte des instructions de choix (IF, CASE)

Soit le programme suivant et sa décomposition atomique

```
A1     { S1 ;
       { if condition then
A2       S2 ;
A3       S3 ;
A4       S4 ;
         else
A5       S5 ;
A6       S6 ;
         end if
A7       S7 ;
A8       S8 ;
A9       S9 ;
A10      S10 ;
```

autorisant, sur les 10 atomes générés lors de la parallélisation, les enchainements décrits par le graphe de dépendance :



(Note : Les arcs en pointillés marquent des dépendances liées à un branchement conditionnel)

Le problème est de savoir comment rendre exécutable les atomes A7, A8 et A9 puisque si l'aternant vrai est choisi les dépendances respectives de ces atomes sont 2, 1, 0 sinon elles sont de 1, 2, 1.

Une solution consiste à accepter qu'un atome puisse faire une décrémentation multiple sur le compteur de dépendance d'une macro instruction qui dépend directement de lui.

Soit une macro instruction MI, n le nombre de ses dépendances à un alternant a1 et m le nombre de ses dépendances à un alternant a2 (avec $n > m$). La dépendance de MI par rapport aux deux alternants est n (max (n, m)) ce qui est exact pour a1 mais faux pour a2. Il faut alors que l'un des atomes de a2 puisse faire une décrémentation multiple, égale à (n-m+1) du compteur de MI.

Les n-uplets générés par le compilateur pour le programme précédent sont :

```
[0]     [n-uplet de A1 ]
[1]     [n-uplet de A2 ]
[1]     [n-uplet de A3 ]
[1]     [n-uplet de A4 ]
[1]     [n-uplet de A5 ]
[1]     [n-uplet de A6 ]
[2]     [n-uplet de A7 ]
[2]     [n-uplet de A8 ]
[1]     [n-uplet de A9 ]
[2]     [n-uplet de A10]
```

Le fonctionnement du programme est le suivant :

- le traitement effectif de A1 exécute les instructions S1 puis évalue l'expression de S1 et place la valeur du test dans une donnée locale booléenne du programme.

- le traitement de contrôle de A1 exploite la variable booléenne pour mettre à jour les compteurs des atomes qui dépendent de la fin de son exécution. Il correspond à :

```
if test then
     [A9]    :=    [A9] - 1 ;  -- A9 ne dépend d'aucun
                                  atome de l'alternant
     [A2]    :=    [A2] - 1 ;     vrai, c'est donc A1
                                  qui libère A9 par une
                                  décrémentation
                                  multiple égale à n.
else
     [A5]    :=    [A5] - 1 ;
     [A6]    :=    [A6] - 1 ;
endif ;
```

Les n-uplets deviennent après l'exécution du traitement de contrôle de l'atome A1 :

si test			si non test	
[X]	[n-uplet de A1]		[X]	[n-uplet de A1]
[0]	[n-uplet de A2]		[1]	[n-uplet de A2]
[1]	[n-uplet de A3]		[1]	[n-uplet de A3]
[1]	[n-uplet de A4]		[1]	[n-uplet de A4]
[1]	[n-uplet de A5]		[0]	[n-uplet de A5]
[1]	[n-uplet de A6]		[0]	[n-uplet de A6]
[2]	[n-uplet de A7]		[2]	[n-uplet de A7]
[2]	[n-uplet de A8]		[2]	[n-uplet de A8]
[0]	[n-uplet de A9]		[1]	[n-uplet de A9]
[2]	[n-uplet de A10]		[2]	[n-uplet de A10]

- l'exécution de l'un des alternants progresse suivant le modèle général ;

- l'enchainement des macro instructions suivant l'alternant choisi se fait grâce à des décrémentations multiples.

Ainsi le programme de contrôle de A4 est :

```
[A7]  :=  [A7] - 1
[A8]  :=  [A8] - 2      -- une double décrémentation pour une
                           dépendance simple.
```

et celui de A5 est :

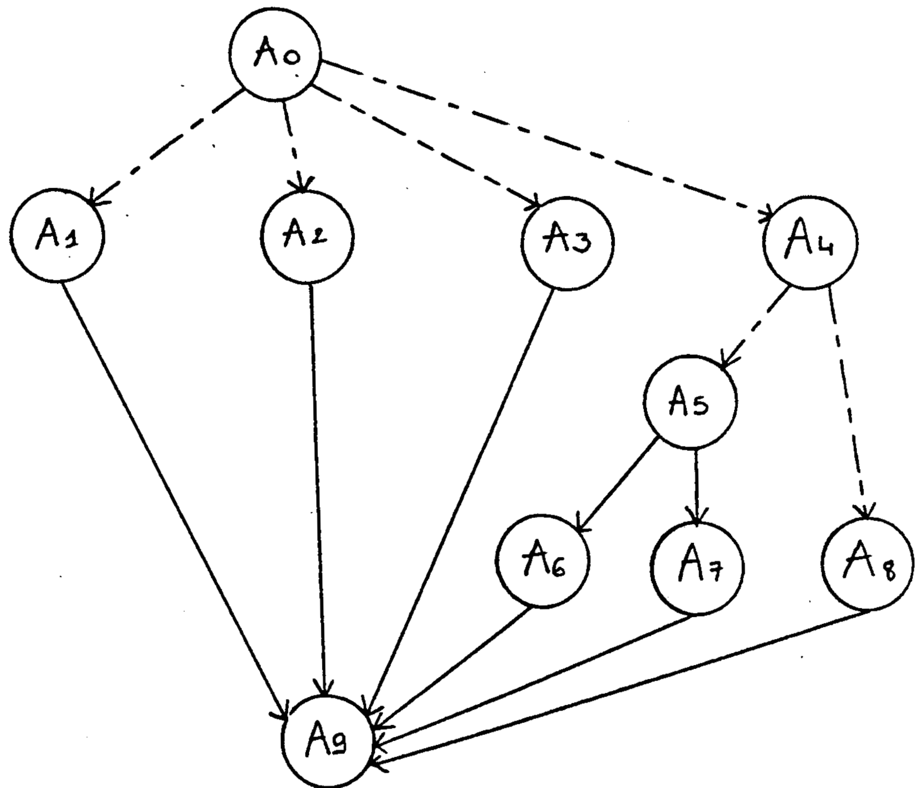
```
[A7]  :=  [A7] - 2      -- double décrémentation pour une
[A8]  :=  [A8] - 1         dépendance simple.
```

Quand plusieurs instructions conditionnelles sont imbriquées, cette solution est applicable. Ainsi lorsque le nombre de dépendance d'une macro instruction est conditionnée par le choix de plusieurs alternants, il suffit de déterminer la suite d'alternants qui produit le plus grand nombre de dépendance sur cette macro instruction et d'initialiser son compteur de dépendance avec ce nombre. Les décrémentations par les atomes se trouvant sur les autres chemins possibles sont ajustées (décrémentations multiples) de façon à ce que la macro instruction soit libérée de toutes les dépendances dûes aux alternants lorsqu'un chemin a été entièrement exécuté.

exemple

Soit le programme suivant, sa décomposition atomique et son graphe de dépendance :

```
     { So ;
A0   {
     { if condition 1 then

A1     S1;

       else

A2     S2 ;

A3     S3 ;

A4   { S4 ;
     { if condition 2 then

A5     S5 ;
A6     S6 ;
A7     S7 ;

       else;

A8     S8 ;

       endif ;

   endif ;

A9     S9;
```



Les n-uplets générées par le compilateur sont :

[0] [n-uplet de A0]
[1] [n-uplet de A1]
[1] [n-uplet de A2]
[1] [n-uplet de A3]
[1] [n-uplet de A4]
[1] [n-uplet de A5]
[1] [n-uplet de A6]
[1] [n-uplet de A7]
[1] [n-uplet de A8]
[4] [n-uplet de A9]

Le nombre de dépendance possible de A9 est : 1, 4, 3, selon les alternants utilisés. Le compteur de dépendance de A9 est initialisé à 4 lors de la parallélisation.

Les décrémentations des atomes se trouvant sur les différents chemins possibles sont ajustées de façon à ce que l'atome A9 soit exécutable à la fin de leur exécution.

Ainsi le traitement de contrôle de A9 produit : [A9] := [A9] - 4 celui de A8 est : [A9] := [A9] - 2.

Cette méthode est parfaitement applicable à des instructions de choix multiples comme les CASE. Il suffit de déterminer pour une macro instruction MI l'alternant qui produit le plus grand nombre de dépendance avec elle, et d'initialiser son compteur avec cette valeur. Les décrémentations par les atomes se trouvant sur les autres alternants possibles sont ajustées de façon à ce que MI n'est plus de dépendance dûe aux alternants lorsque l'un

d'entre-eux a été exécuté.

Cette solution permet de mettre en oeuvre des formes puissantes de parallélisme grâce aux programmes de contrôles simples. Cette simplicité provient du fait d'une analyse puissante des enchainements, lors de la compilation ou parallélisation. Il est toutefois possible de reprocher à cette méthode un manque d'homogénéité dûe à la possibilité d'effectuer des décrémentations multiples pour des dépendances simples et que par conséquent la valeur des compteurs de dépendance n'a pas toujours exactement la même signification que celle définie dans le modèle.

Cependant une méthode consistant à matérialiser par un évènement unique la terminaison de chaque alternant n'a pas été retenue car elle conduit, lorsqu'un alternant a plusieurs macro instructions terminales, à générer une molécule pour satisfaire cette propriété et aboutit à un taux d'exploitation du parallélisme moins bon que pour la méthode précédente, des décrémentations plus nombreuses, et l'activation d'une molécule supplémentaire.

En outre, une méthode consistant à modifier dynamiquement la valeur des compteurs de dépendance par des traitements de contrôle a été envisagée mais non retenue car les programmes de contrôle sont alors légèrement plus importants.

2.3.2 Prise en compte des boucles

Le modèle data flow macroscopique permet de proposer plusieurs formes d'exécution d'une boucle. Elles dépendent des ressources de la machine, de la possibilité d'exécution simultanée des instructions composant le corps de la boucle et/ou des itérations de la boucle.

Un des avantages du système molécules, atomes et mots d'état associés, est qu'il constitue une structure non figée qu'il est possible d'adapter à la résolution de nombreux problèmes. Ainsi dans ce paragraphe quelques méthodes de traitements des boucles sont présentées mais d'autres techniques peuvent être imaginées, par exemple en tirant partie d'informations dynamiques sur l'état de la machine ou de caractères propres à la boucle et aux données qu'elle traite (comme pour les boucles vectorielles).

De nombreuses formes d'exploitation du parallélisme sont possibles grâce à des programmes de contrôle ayant une sémantique élevée, mais il faut cependant ne jamais négliger le rapport temps de traitement effectif sur temps de traitement de contrôle qui, s'il n'est pas suffisant, rendrait inefficace toute stratégie de parallélisation.

1 - boucle séquentielle

Le schéma d'exécution d'une boucle peut devoir être séquentiel, c'est à dire que les itérations doivent être exécutées successivement. Cela peut résulter par exemple de ce qu'une itération produit une donnée qui est ou peut être consommée à l'itération suivante, ou encore parce que la boucle parcourt une liste dont la structure peut être modifiée pour toute itération. Dans ce cas, il n'apparait pas utile que la boucle apparaisse au niveau de la machine.

La boucle est placée dans un atome, seule ou avec d'autres instructions avec lesquelles elle est en relation de dépendance. L'exécution de la boucle est réalisée dans le traitement effectif associé à l'atome qui la contient, et la boucle n'apparait pas au niveau des traitements de contrôle de la machine.

2 - boucle statique parallélisable

Lorsque le nombre d'itérations d'une boucle est connu à la compilation chaque itération doit être décrite par un graphe qui selon sa complexité est composé d'atomes et/ou de molécules.

L'enchainement de la fin de chaque itération et de la suite du programme se fait selon les relations de dépendances entre les macro instructions de chaque itération et des macro instructions suivantes détectées à la pré-compilation et caractérisées par la valeur des compteurs de dépendance et les traitements de contrôle.

En fait la boucle est incluse à l'intérieur du programme et aucun traitement particulier ne fait apparaitre cette notion.

Il se peut que le traitement effectif d'une itération d'une boucle parallélisable soit insuffisant par rapport au traitement de contrôle mis en oeuvre pour exploiter celui-ci, dans ce cas, il est possible d'exploiter un parallélisme inférieur au nombre d'itérations de la boucle et chaque traitement parallèle pourra comprendre n itérations de boucles traitées séquentiellement, formant chacun un atome. Pour une boucle de I itérations, I/n atomes pourront être traités en parallèle.

Dans le cas des boucles où le nombre d'itérations est important, rendant difficile la description de l'ensemble de la boucle sous forme de graphe, il est possible d'utiliser des graphes réentrants, le principe de fonctionnement s'apparente à celui des boucles dynamiques parallélisables décrites dans le paragraphe suivant.

3 - Boucle dynamique dont les itérations sont parallélisables

Les itérations d'une boucle dont la variable d'itérations n'est connue qu'à l'exécution, peuvent être, soit du fait du texte du programme ou soit après transformation automatique, exécutées simultanément.

Le principe général de traitement d'une boucle dynamique parallèle est le suivant :

- la boucle est transformée par le compilateur ou l'outil logiciel approprié en une molécule qui ne contient que des traitements de contrôle ayant pour effet d'activer les itérations de la boucle. L'exécution de ce programme de contrôle a pour effet d'introduire dans le vecteur d'état des n-uplets décrivant les itérations à exécuter et agit sur la variable d'itération (décrémenter, tester) qui, comme paramètre caractérisant la molécule boucle, est stockée dans le mot d'état de celle-ci.

Le corps d'une itération est, suivant sa complexité, représenté, après transformation de programme, par un ou des atomes et éventuellement une ou des molécules dont les relations sont décrites par un graphe. L'exécution de chaque itération est cadencée par les mécanismes généraux décrits en 2.1 et 2.2.

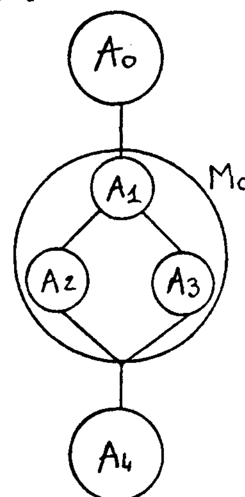
Lorsque les vecteurs d'état des n itérations actives sont générés, le programme de contrôle de la molécule décrémente de n la variable d'itération, et rend exécutable (compteur de dépendance = 0) les macro instructions initiales du graphe de dépendance de chaque itération après avoir initialiser son compteur de dépendance à la valeur n*T, où T est égal au nombre de macro instructions terminales de chaque itération.

Lorsqu'une itération a été exécutée le compteur de la molécule boucle a été décrémenté de T. La valeur nulle de ce compteur entraine la réactivation de la molécule qui selon la valeur de la variable d'itération génère de nouveaux vecteurs d'états itérations ou réalise les enchainements avec la suite du programme.

Le choix du nombre d'itérations générées reste à déterminer, il peut être constant ou au contraire variable et calculé par des algorithmes prenant en compte le nombre total d'itérations à exécuter, la charge de la machine, la priorité des travaux à exécuter, par exemple.

exemple

Soit un programme comportant une boucle dynamique dont les itérations sont exécutables en parallèles. Sa décomposition macroscopique et son graphe de dépendance sont :




```
A0        S0 ;

          D0 10  I = 2,N

   ⎧A1    S1 ;
M0 ⎨A2    S2 ;
   ⎩A3    S3 ;

     10 continue
A4        S3 ;
```

Les n-uplets générés par le compilateur sont :

```
[0]      |n-uplet de A0|
[1]      |n-uplet de M0|
[1]      |n-uplet de A4|
```

Le traitement de A0 permet de calculer la valeur de N qui est passée en paramètre à M0. Le compteur de dépendance de M0 a été décrémenté par le traitement de contrôle de M0, et a pris la valeur 0, la molécule boucle peut donc être activée. Elle génère m vecteurs d'états :

```
[1]      [n-uplet de A1]
[1]      [n-uplet de A2]
[1]      [n-uplet de A3]
```

puis réinitialise son compteur de dépendance a 2m, décrémente de m sa variable d'itération, et décrémente les m compteurs de dépendance des atomes A1 initiaux pour les itérations lancées. La fin de l'exécution des atomes terminaux de chaque itération (A2, A3) entraine la décrémentation du compteur de dépendance de M0. Lorsque celui-ci est nul, selon la valeur de la variable d'itération de nouvelles itérations sont générées ou le compteur de dépendance de A4 est décrémenté.

2.2.3 Prise en compte des sous-programmes

Deux attitudes sont envisagées pour prendre en compte les sous-programmes au niveau de la machine.

Une première attitude consiste à inclure le sous programme appelé à l'intérieur du programme appelant.

Ainsi un sous programme décomposé en un système d'atomes et de molécules, dont les relations sont décrites par un graphe, est complètement intégré dans le système représentant le programme appelant.

Cette approche est semblable aux techniques d'optimisation de programme et correspond aux techniques de copie et d'inclusion textuelle. Elle doit bien sûr être utilisée à chaque fois que cela est possible car tous les traitements exécutés à la compilation ne sont plus à réaliser au cours de l'exécution.

Il existe des cas où cette méthode est soit inefficace pour des raisons d'emplacements mémoire par exemple, soit impossible comme dans le cas des sous programmes récursifs. Il faut alors conserver la notion de sous-programme dans la représentation parallélisée du programme. Ce caractère dynamique est introduit grâce à la notion de molécule. Ainsi un sous-programme, peut être encapsulé dans une molécule, dite molécule sous-programme dont le rôle sera, lors de l'exécution, de générer le système d'atomes et de molécules caractérisant le sous-programme qui est décrit par un vecteur d'état.

Le principe de fonctionnement de ce système est :

- le compteur de dépendance de la molécule sous-programme est décrémenté par les macro instructions dont le sous-programme dépend directemenet.

- la mise à 0 de ce compteur active le traitement de contrôle de la molécule sous-programme qui génère le vecteur d'état correspondant au sous-programme et réinitialise son compteur de dépendance par le nombre T de macro instructions terminales du graphe correspondant au sous-programme.

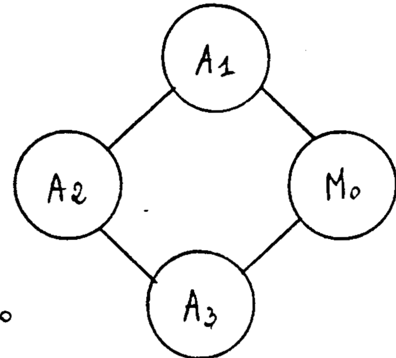
- les compteurs de dépendance des macro instructions initiales du sous-programme généré sont ensuite mis à zéro, ce qui active le sous-programme.

- la fin de l'exécution des macro instructions terminales du sous-programme généré entraine la décrémentation du compteur de dépendance de la molécule sous-programme.

La mise à Ø de ce compteur active le traitement de contrôle de la molécule sous-programme qui assure les enchainements avec la suite du programme.

exemple

Soit un programme P appelant un sous-programme récursif R dont les décompositions macroscopiques sont :

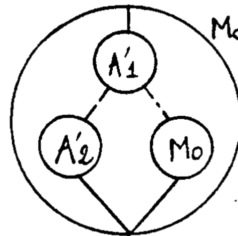


P :

```
A1      S1 ;
MØ      call R ;
A2      S2 ;
A3      S3 ;
```

R :

```
A'1     S'1 ;
MØ      call R ;
A'2     S'2 ;
```



Les vecteurs d'état de P sont :

[Ø]	[n-uplet de A1]
[1]	[n-uplet de A2]
[1]	[n-uplet de MØ]
[2]	[n-uplet de A3]

Le fonctionnement est le suivant :

- le traitement de contrôle de A1 décrémente les compteurs de dépendance de A2 et MØ qui deviennent exécutables. Le vecteur d'état de P devient :

[-]	[n-uplet de A1]
[Ø]	[n-uplet de A2]
[Ø]	[n-uplet de MØ]
[2]	[n-uplet de A3]

- lorsque A2 a terminé d'exécuter son traitement effectif son traitement de contrôle décrémente le compteur de dépendance de A3.

- l'exécution de la molécule MØ génère le vecteur d'état correspondant au sous-programme R :

```
[ 1 ]        [ n-uplet de A'1 ]
[ 1 ]        [ n-uplet de A'2 ]
[ 1 ]        [ n-uplet de MØ ]
```

réinitialise son compteur dépendance à 1 et décrémente le compteur de dépendance de A'1. Le vecteur d'état du programme devient :

```
[ - ]        [ n-uplet de A1 ]
[ Ø ]        [ n-uplet de A2 ]
[ 1 ]        [ n-uplet de MO ]
[ 2 ]        [ n-uplet de A3 ]

[ Ø ]        [ n-uplet de A'1 ]
[ 1 ]        [ n-uplet de MØ ]
[ 1 ]        [ n-uplet de A'2 ]
```

L'exécution de A'1 détermine si le sous-programme R est rappelé, si oui le procédé précédent est itéré, sinon le compteur de A'2 est décrémenté, ce qui implique l'exécution de cet atome dont le traitement de contrôle décrémente le compteur de dépendance de la molécule MØ correspond à la 1ère instanciation de R.

Le vecteur d'état devient dans ce cas :

```
[ - ]        [ n-uplet de A1 ]
[ Ø ]        [ n-uplet de A2 ]
[ Ø ]        [ n-uplet de MØ ]
[ 2 ]        [ n-uplet de A3 ]

[ - ]        [ n-uplet de A'1 ]
[ 1 ]        [ n-uplet de MØ ]
[ - ]        [ n-uplet de A'2 ]
```

La molécule réactivée décrémente alors le compteur de dépendance de A3 afin d'assurer l'enchainement avec la suite du programme.

La fin du traitement de A2 déclenche la décrémentation du compteur de dépendance de A3.

· Le vecteur d'état est alors :

```
[-]        [n-uplet de A1 ]
[-]        [n-uplet de A2 ]
[-]        [n-uplet de MØ ]
[Ø]        [n-uplet de A3 ]
[-]        [n-uplet de A'1]
[1]        [n-uplet de MO ]
[-]        [n-uplet de A'3]
```

2.4. CONCLUSIONS

Dans ce chapitre il a été rappelé les principes du modèle d'exploitation du parallélisme appelé Data Flow Macrascopique. Ce modèle, caractérisé par sa simplicité, permet de mettre en oeuvre de multiples formes de parallélisme transparents à l'utilisateur. Ainsi, le mode d'enchainement des instructions autorise aussi bien l'exécution parallèle des termes des expressions arithmétiques ou booléennes, que les instructions du programme isolées ou regroupées ou encore des tâches au sens des systèmes actuels.

Ce modèle a des caractéristiques "DATA FLOW" parce que les traitements de contrôle reflètent les dépendances induites par les variables lues et modifiées par les instructions. Comme pour les machines "dirigées par les données" les enchainements entre groupes d'instructions, ou macro-instructions, sont décrites par un graphe de dépendance. Chaque macro instruction est exécutable selon le principe de "VON NEUMAN". Ceci permet d'allier les avantages du parallélisme et du séquentiel, et permet d'adapter le parallélisme exploité au parallélisme inhérent des programmes, aux structures de contrôle nécessaire pour son exploitation et aux caractéristiques de la machine cible, en vue d'une efficacité optimun.

CHAPITRE III

LE DATA FLOW MACROSCOPIQUE APPLIQUE AU LANGAGE ADA

Chapitre 3 - LE DATA FLOW MACROSCOPIQUE APPLIQUE AU LANGAGE ADA

3.0 INTRODUCTION

Outre les structures classiques de programmation qui ont été examinées dans le chapitre précédent, il apparait important que le modèle Data Flow Macroscopique puisse supporter facilement les structures de programmation du Langage Ada permettant la récupération d'"erreurs", grâce aux traitements d'exception, le support du temps réel (synchronisation et communication entre processus), grâce aux tâches.

Les spécifications de ADA en font un langage algorithmique à vocation universelle (scientifique, gestion, temps réel, ...), qui financé par le "Department of Defense" américain, est amené à connaître un développement important comme en témoigne le nombre de projets d'implantation à travers le monde (compilateurs Bull/Alsys/Siemens, Western Digital, implantation sur IAPX 432, environnement de programmation pour Data General et Rohm, etc...).

Dans un premier paragraphe les problèmes sémantiques de la parallélisation d'un programme Ada sont étudiés puis différentes techniques d'implémentation des concepts de tâches, de rendez-vous et d'exception grâce aux mécanismes d'atomes et de molécules sont examinées. Il faut noter que les problèmes systèmes d'exclusion mutuelle, de sémaphore avec message et de gestion de files d'attente ne sont pas traités, des études comme (Müh 81), (Ren 81), (Jon 82) approfondissent ce sujet.

Ces techniques systèmes de synchronisation et de communication sont utilisées dans des implémentations classiques des programmes et il n'est pas nécessaire de préciser leur fonctionnement pour appliquer le concept de Data Flow Macroscopique au Langage Ada bien qu'elles soient sous-jacentes aux mécanismes associés.

La puissance et la capacité d'adaptation du modèle grâce à des structures de contrôle d'un haut niveau sémantique permet d'exploiter de nombreux cas de parallélisation tout en conservant une "granularité" de parallélisme (cf 1.3.3.) entrainant un coût d'exploitation raisonnable. Dans ce chapitre il est montré comment le modèle Data Flow Macroscopique permet d'exploiter le parallélisme inhérent aux programmes Ada.

3.1. PARALLELISATION DES PROGRAMMES ADA

Le langage Ada dispose des structures permettant au programmeur d'expliciter le parallélisme, ce sont les concepts de tâches, de synchronisation et de communication, exprimés par la notion de "rendez-vous". Le but du Data Flow Macroscopique étant de proposer un moyen d'expression du parallélisme inhérent au programme qui soit transparent à l'utilisateur, ce paragraphe traite des problèmes sémantiques qui découlent de son application à Ada.

En ce qui concerne les mécanismes classiques de programmation qui se retrouvent dans Ada, la transformation d'un programme en programme parallélisé grâce au Data Flow Macroscopique ne modifie pas les effets du programme sur les données lues et modifiées par les instructions, puisque la parallélisation automatique se fait en respectant les règles de dépendances entre les données.

Aux mécanismes classiques de lecture et de modification de variables, viennent se greffer dans Ada, des mécanismes de traitement d'exception qui peuvent être levées durant l'exécution d'un programme entrainant la suspension de l'exécution de l'unité qui la contient et le branchement au traitement d'exception adéquat. Ce type de branchement engendre des dépendances sur les contrôles (cf §1.1.) et implique des problèmes supplémentaires lors de la parallélisation.

Les mécanismes de synchronisation explicite, exprimées par les rendez-vous, entrainent aussi des dépendances de type contrôle qui n'apparaissent pas dans les langages comme FORTRAN.

Les problèmes sémantiques que ces types de concept peuvent engendrer lors de transformations de programmes sont étudiés dans les deux paragraphes qui suivent.

3.1.1 Ordonnancement des points de rendez-vous

Si la parallélisation se limite à l'examen seul des données lues et modifiées il est possible, à priori, que des évènements de type rendez-vous comme les "accept", les "select" et les appels, soient exécutables en parallèle, c'est à dire que leurs éxécutions soient indépendantes alors que dans le programme d'origine elles étaient exécutées suivant un ordre défini ou canonique.

Ce type de parallélisme ne semble pas avoir été prévu par les spécifications du manuel de référence Ada (USG 83) et de nombreuses précautions sont nécessaires.

Pour une tâche, lorsque plusieurs traitements de rendez-vous sont imbriqués, l'ordre d'exécution des synchronisations correspondantes doit être inchangé car : "pour chaque opération, le traitement englobant le plus interne ou l'instruction accept doit être le même, dans un nouvel ordre, que dans l'ordre canonique" (USG 83 §2.6.4).

Pour paralléliser des traitements de rendez-vous non imbriqués il faut déterminer les relations de dépendance entre les instructions propres d'une tâche, mais également entre plusieurs tâches, lorsque celles-ci lisent ou mettent à jour une (ou des) variable partagée. Ainsi lorsque "deux tâches lisent ou modifient une variable partagée, alors aucune d'elles n'est en mesure de présupposer quoique ce soit sur l'ordre dans lequel l'autre effectue ses opérations sauf aux points où elles se synchronisent". C'est à dire que les actions sur des variables partagées précèdant un point de rendez-vous ont été exécutées au moment de la synchronisation correspondante. Dès lors des hypthèses peuvent être faites sur les actions effectuées par deux tâches avant et après un point de synchronisation, de façon à maintenir la cohérence des données lues et modifiées. Dans de tels cas l'ordre d'exécution des points de rendez-vous doit être respecté sous peine d'aboutir à l'exécution erronnée du programme.

Enfin les spécifications du langage précisent qu'une tâche ne peut pas être dans deux files d'attente à la fois, c'est à dire que les appels de rendez-vous doivent être exécutés de façon séquentielle.

Outre les nombreux problèmes de transformation de programme posés par l'exécution parallèle de rendez-vous dans une tâche, de nombreux problèmes systèmes doivent être résolus, comme la gestion des files d'attente, le choix des rendez-vous à réaliser ... Un tel niveau de parallélisme est complexe à mettre en oeuvre et ne correspond pas aux objectifs poursuivis par ce projet visant une mise en oeuvre simple, et efficace, du parallélisme inhérent aux programmes. Ainsi les règles de décomposition macroscopique des tâches Ada doivent, selon notre interprétation, respecter l'ordre d'exécution des points de rendez-vous et garantissent que les effets dûs à la synchronisation et à la communication entre tâches sont inchangés du fait de la parallélisation. L'analyse des différentes parties d'un programme doit permettre de détecter ce type de dépendance entre les évènements "rendez-vous" et la décomposition macroscopique, avec les différentes relations entre macro-instructions, doit traduire cette séquentialité.

3.1.2 Parallélisme et traitement d'exception

Les mécanismes d'exception définis dans Ada pour traiter certains types d'erreurs ou de situations exceptionnelles, pouvant survenir durant l'exécution d'un programme, posent des problèmes de sémantiques particuliers qui nécessitent un examen précis lors de toute transformation de programme.

L'exploitation du parallélisme à l'intérieur d'une tâche est autorisée par le langage puisque le paragraphe 9.5 du manuel Ada (USG 83) indique :

"Lorsqu'une implémentation peut détecter que les mêmes effets peuvent être garantis si diverses actions d'une tâche donnée sont exécutées par différents processeurs physiques fonctionnant en parallèle, elle peut décider de les exécuter de cette manière ; dans ce cas plusieurs processeurs physiques constituent un unique processeur logique".

Il faut cependant respecter certaines règles, sous peine d'une exécution erronnée du programme. Les règles d'optimisations décrites au paragraphe 11.6 du manuel Ada permettent toutes les optimisations classiques des langages de programmation, comme l'élimination des sous expressions redondantes, les extractions d'invariant de boucle, les techniques d'anticipation ou de temporisation, l'élimination du code mort, ... (Bra 83).

Ces transformations de programmes sont autorisées du moment qu'elles garantissent l'équivalence sémantique avant et après optimisation, c'est à dire que :

"Une implémentation peut modifier l'ordre d'exécution de certaines actions seulement si elle peut garantir que les effets du programme sont inchangés par ce réordonnancement" (manuel Ada §11.6.2).

Les mécanismes d'exception nécessitent de prendre des précautions particulières pour respecter cette règle.

Deux types d'exceptions doivent être distinguées :

- les exceptions "synchrones" : ce sont des exceptions levées explicitement par le programmeur grâce à une instruction "raise". Cette levée d'exception permet la suspension d'une unité et un branchement à un traitement d'exception spécifique. La dépendance qui lie une instruction "raise" et les instructions qui la suivent est une dépendance sur les contrôles analogue à celle qui lie une instruction "if" et ses parties "vrai" et "fausse". La levée explicite d'une exception fait partie des techniques normales de programmation permettant de traiter certaines situations "exceptionnelles".

Ainsi une instruction I située après une instruction raise, dans l'ordre canonique, ne peut être placée après "optimisation", devant cette instruction, car, lors de la levée d'une exception, I serait exécutée dans le programme optimisé mais pas dans le programme canonique, les effets du programme sont donc changés par ce réordonnancement.

Ainsi les techniques classiques d'optimisation ne sont applicables qu'aux parties de programmes comprises entre deux exceptions synchrones du moment que les effets sont identiques à ceux du programme origine.

- les exceptions "asynchrones" : ce sont les exceptions prédéfinies matérielles ou logicielles levées par le système lors d'une anomalie. Elles permettent au programmeur de récupérer certaines erreurs grâce aux traitements d'exception. Comme pour les exceptions synchrones, leur levée entraine la suspension d'une unité et le branchement à un traitement d'exception spécifique. Cependant, il ne peut être tenu compte de ce type d'exception durant une optimisation ainsi que le précise le manuel Ada (§ 11.6.4) :

"Dans le but d'établir que le même effet est obtenu par l'exécution de certaines actions dans l'ordre canonique et dans un ordre différent, il doit être supposé qu'aucune opération prédéfinie invoquée par ces actions ne propage une exception (prédéfinie)".

Cette spécification permet de rendre conciliable les techniques de récupération d'erreur et d'optimisations qui, sinon, seraient très limitées. En effet, sans elles, seules les instructions ne pouvant comporter d'erreur pourraient être déplacées puisque les "optimisations" réalisées sur les autres instructions entrainent l'exécution d'actions différentes du programme d'origine selon les exceptions levées.

Basées sur l'analyse des nombreux paragraphes du Manuel Ada concernant l'optimisation, cette interprétation est confirmée par l'exemple 11.6.11 du manuel :

```
declare
  N : integer ;
begin
  N := 0 ;              -- (1)
  for J in 1..10 loop
  N := N + J**A(K); -- A et K sont des variables globales
  end loop ;
  PUT(N) ;
exception
 when others => PUT ("Some error arrose"); PUT(N) ;
end ;
```

L'évaluation de A(K) peut être réalisée avant la boucle, ou également juste avant l'instruction (1), même si cette évaluation peut lever une exception. Par conséquent, dans le traitement de l'exception la valeur initiale de N est soit indéfinie, soit déjà assignée. En outre, l'évaluation de A(K) ne peut être déplacée avant "begin" car une exception serait alors traitée par un traitement d'exception différent (l'exception serait traitée par une des unités englobantes).

La levée d'une exception lors de l'évaluation A(K) est de type asynchrone, par conséquent, il est possible de déplacer l'expression A(K) pour optimiser le programme, du moment que les effets du programme réordonné sont semblables à ceux du programme d'origine lorsqu'il ne comporte pas d'erreur.

Cependant, en supposant que la boucle de l'exemple précédent soit une boucle dynamique fonction de K (for J in 1..K loop) et que A soit un tableau avec un intervalle de définition indexé de 1 à 10, l'exécution du programme canonique avec K=-1 ne déclenche pas d'exception puisqu'alors aucune itération de la boucle A est exécutée ; par contre lorsque l'évaluation de A(K) est réalisée avant la boucle une exception est levée, une telle optimisation n'est donc pas autorisée selon l'interprétation faite précédemment puisque les effets du programme optimisé ne sont pas les mêmes que ceux du programme initial sans erreur. Cette interprétation est confirmée par la phrase du §11.6.2 du manuel : "En particulier, aucune exception ne sera levée dans le programme réordonné si aucune n'est levée dans le programme dans l'ordre canonique".

Les règles de parallélisation des programmes sont les mêmes que celles de la transformation des programmes dans le but d'une optimisation. D'ailleurs la parallélisation est une forme d'optimisation de l'exécution des programmes.

Aussi la décomposition macroscopique d'un programme Ada doit elle respecter l'équivalence sémantique avant et après parallélisation. En particulier, elle doit tenir compte de la dépendance liant une instruction "raise" et des instructions qui la suivent. Les instructions "raise" doivent être ordonnées après transformation de programme de la même façon que dans l'ordre canonique, et seul le parallélisme sur les sections de programme où aucune exception explicite ne peut être levée est exploitable. Ainsi une exception explicite est un point de convergence du programme, et son état en ce point sera caractérisé par l'exécution de l'ensemble des instructions qui précède l'exception et le fait qu'aucune des instructions suivantes n'aura été exécutée.

Soit par exemple, un programme composé de deux séquences d'instructions A et B qui dans l'ordre canonique sont exécutées l'une après l'autre et à l'intérieur desquelles les exceptions E1 et E2 peuvent être levées.

```
         ┌     ┌begin
         │A1  ─┤ :
A       ─┤     └Sn    ;    -- l'exception E1 peut être levée
         │A2   ┌ :
         └     └ :
         ┌B1  ─┤ :
B       ─┤     ┌Sp    ;    -- l'exception E2 peut être levée
         │B2  ─┤ :
         └     └ :
          exception
             when E1   ==>          -- traitement E1
             ---
             when E2  ==>          -- traitement E2
             ---
          end ;
```

- les deux séquences d'instruction A et B sont parallélisables, si :

. les lectures et modifications de variables de l'une n'interfèrent pas sur celles de l'autre

. les exceptions E1 et E2 sont des exceptions asynchrones

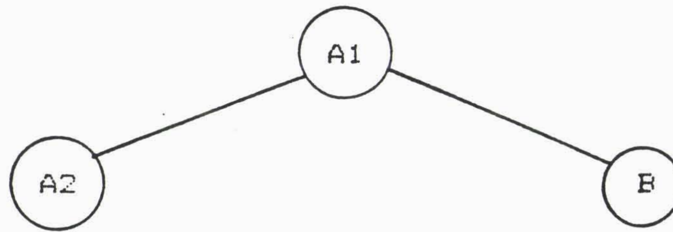
. il n'existe pas d'exception explicite dans A ni dans B.

Dans ce cas, que l'exception E2 soit levée avant (ou après) l'exception E1, du fait d'une exécution parallèle du programme, est autorisée selon notre interpétation, car alors le programme ordonné canoniquement est de toute façon erronné et une de ces exceptions aurait été levée et traitée.

Ainsi, un traitement d'exception dans le programme d'origine, entraine un traitement d'exception dans le programme parallélisé. Par contre, il ne faut pas qu'une exception soit levée durant l'exécution du programme parallélisé alors que dans l'ordre canonique aucune exception n'aurait été levée.

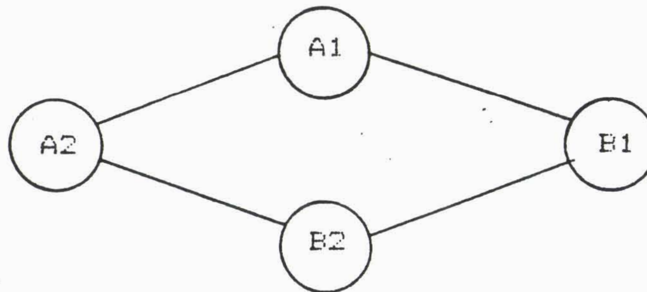
- Si par contre, l'exception E1 est synchrone, c'est à dire levée par une instruction "raise", alors les deux séquences ne sont plus entièrement parallélisables. Il faut garantir que les instructions qui précèdent l'exception dans le programme initial, soient, dans le programme parallélisé, toutes exécutées avant de pouvoir exécuter les instructions qui suivent

Un des graphes de dépendance de ce type de programme serait:



Il montre que la séquence d'instructions précèdant l'exception synchrone E1 doit être achevée avant de pouvoir exécuter de façon parallèle ou non les instructions qui suivent. Dans cet exemple les blocs d'instructions A2 et B sont exécutables en parallèle.

- Si E1 et E2 sont toutes les deux des exceptions synchrones un des graphes de dépendance possible serait :



Il montre que la séquence d'instructions située entre E1 et E2 est parallélisable. A2 et B1 sont exécutables une fois que le bloc d'instructions contenant E1 est achevé, de même B2 n'est exécutable que lorsque toutes instructions qui le précèdent dans le programme ordonné canoniquement sont achevées.

3.2 ØDATA FLOW MACROSCOPIQUE APPLIQUE AUX TACHES

Les tâches sont des processus explicités par le programmeur pouvant être exécutés en parallèle, leur syntaxe est (USG 83 chapitre 9)

```
task-body ::=
     task body task-simple-name-is
          (déclarative part)

     begin
          sequence of statements

     (exception
          exception-handler
          exception-handler   )
     end(task-simple-name) ;
```

L'activation d'un objet tâche commence juste après l'élaboration de la partie déclarative (c'est à dire juste après le mot réservé begin qui la suit) dans laquelle il est déclaré.

Une fois activées, les différentes tâches progressent indépendamment les unes des autres et de la tâche qui les active, sauf en certains points où elles se synchronisent.

Ainsi, une tâche apparait comme un regroupement logique d'instructions correspondant à un "programme" ou unité exécutée sur un processeur logique.

Comme pour un programme une tâche est analysée et décomposée, par l'outil logiciel de parallélisation automatique, le compilateur, en un ensemble de macro instructions (atomes ou molécules) dont les relations de dépendances sont décrites par un graphe de dépendance. Le compteur de dépendance et le traitement de contrôle de chaque macro-instructions traduisent ce graphe de dépendance et réalisent les enchainements à l'intérieur de la tâche.

Une tâche est donc une macro-instruction qui, selon sa complexité et le parallélisme engendrée, peut être décomposée en un ensemble de macro-instructions plus simples. Comme pour un programme, la décomposition macroscopique d'une tâche peut prendre en compte les différentes structures classiquement rencontrées dans les langages de programmation comme les instructions conditionnelles, les boucles ou les sous programmes. Mais elle peut également faire apparaitre des structures plus spécifiques aux tâches Ada telles que les instructions relatives aux mécanismes de rendez-vous ou d'exception.

3.2.1 Prise en compte des mécanismes de "rendez-vous"

Les tâches Ada sont des processus exécutables simultanément dont la synchronisation et les échanges d'informations sont exprimés par la notion de "rendez-vous".

Le modèle data flow macroscopique peut prendre en compte cette synchronisation explicite grâce à une décomposition en atomes et molécules des tâches. Cette section montre comment des mécanismes d'expression du parallélisme explicites (accept, appel, appel conditionnel, appel avec attente limitée, select) et ceux permettant un parallélisme transparent à l'utilisateur sont exploitables simultanément.

Les mécanismes, décrits par la suite, supposent l'utilisation d'outils système comme des sémaphores, des files d'attente,..., dont il n'est pas nécessaire de préciser le fonctionnement pour expliquer les principes d'enchainement du système parallélisé. Le chapitre V montre une implémentation possible du modèle sur un système particulier (MLI).

3.2.1.1 Prise en compte des instructions "accept" et appel d'entrée

Ce paragraphe comporte trois parties : la première est un bref rappel de la notion de rendez-vous ; la seconde présente le principe de fonctionnement du système data flow macroscopique lors d'un rendez-vous, cette étude est réalisée sur un modèle de décomposition macroscopique générale et simplifiée afin de faciliter la description des mécanismes d'enchainement ; la troisième partie montre comment à partir du modèle de décomposition générale d'autres systèmes de macro-instructions peuvent être déduits afin d'exploiter de nombreux cas de parallélisme.

a) Notion de "rendez-vous"

Les instructions "accept" et appel d'une entrée sont les structures de synchronisation et de communication de valeurs entre tâches (USG 83 §9.5).

L'instruction "accept" est exécutée par la tâche où est déclarée l'entrée à laquelle elle se réfère. Si une tâche T1 exécute une instruction "accept", avant tout appel de l'entrée correspondante par une autre tâche T2, alors elle est bloquée jusqu'à ce que cet appel soit réalisé.

De même, T2 est bloquée sur un appel à l'entrée de T1, tant que T1 n'est pas arrivée sur une instruction accept correspondant à l'appel et que le traitement de rendez-vous (optionnel) n'est pas exécuté. Lorsque plusieurs appels d'une entrée sont effectués avant une instruction accept correspondante, ils sont mémorisés dans des files d'attente propres à chaque entrée. Chaque exécution d'une instruction "accept" retire un appel de la file d'attente de son entrée.

La syntaxe est la suivante :

déclaration d'entrée::=

    entry ((intervalle-discret)) (partie formelle) ;

appel d'entrée::=

    nom-d'entrée (partie-paramètres-effectifs) ;

instruction-accept::=

    accept nom-simple d'entrée ((indice-de-
            l'entrée))(partie-femelle)(do
            suite-d-instructions
    end (nom-simple-d-entrée)) ;

indice-de-l-entrée ::= expression

b) Décomposition macroscopique générale

Le chapitre 3.11 a montré que l'ordre des traitements des instructions "accept" et des appels dans une tâche parallélisée doit être identique au programme canonique.

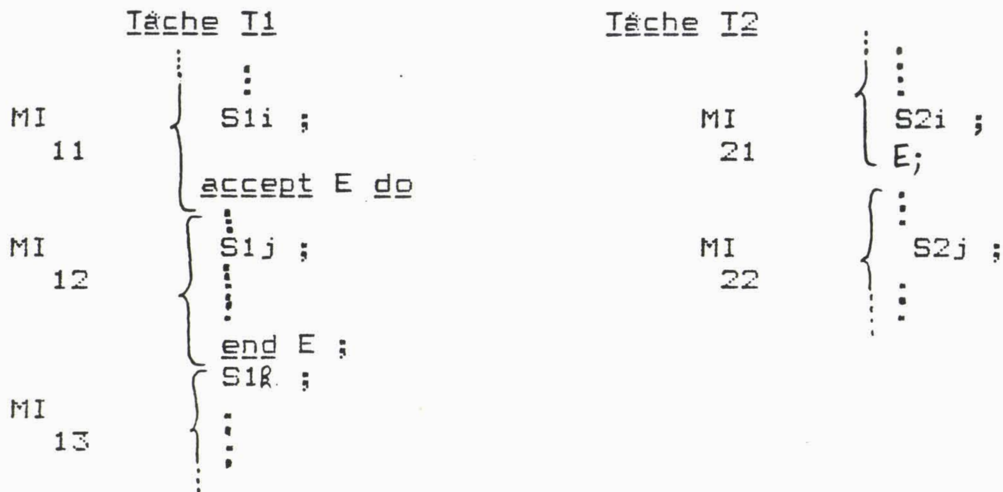
Cette propriété est garantie si le système de parallélisation assure que toutes les actions qui prècèdent ou suivent un rendez-vous dans un programme canonique garde le même ordre relatif par rapport au rendez-vous dans le programme parallélisé.

Ainsi, dans une tâche, les parties de programmes situées entre deux rendez-vous sont parallélisables, le début d'une rendez-vous apparaît comme un point de jonction d'actions exécutées en parallèle (JOIN) et la fin d'un traitement de rendez-vous apparaît comme un point de développement en actions parallèles d'un programme (FORK).

Chaque partie de tâche avant un point de rendez-vous ou après un traitement de rendez-vous est une macro-instruction, chacune d'elles pouvant, selon sa complexité, être décomposée en un ensemble de macro-instructions, si les règles de décomposition et d'enchainement du modèle Data Flow Macroscopique sont respectées.

De même la suite d'instructions correspondant au traitement de rendez-vous peut être considérée comme une macro-instruction pouvant selon sa complexité être décomposée en un ensemble de macro-instructions.

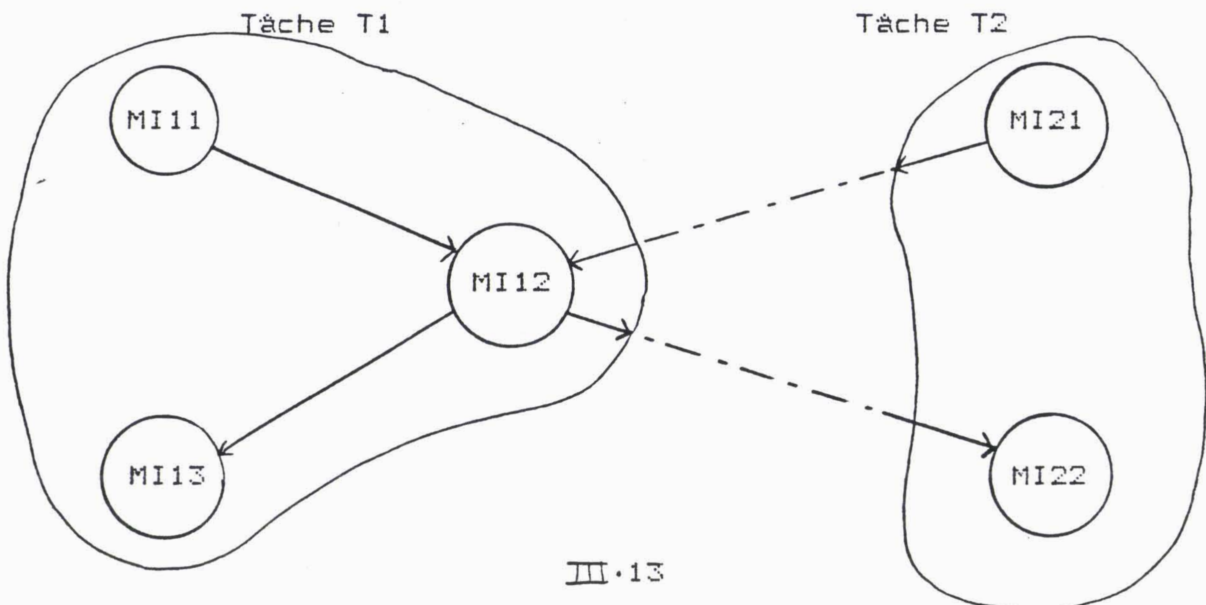
Ainsi, la décomposition macroscopique générale de deux tâches T1 et T2 pouvant communiquer en un point de rendez-vous d'entrée E est :



Dans ce cas simplifié les macro-instructions sont des atomes dont le traitement effectif réalise l'exécution d'une partie de la tâche, et le traitement de contrôle réalise les actions nécessaires à leurs enchainements.

L'exécution de la macro-instruction MI12 dépend de l'exécution de l'instruction "accept E", donc de l'achèvement de MI11, et de l'exécution d'un appel de l'entrée E par une autre tâche. Cet appel est inclus dans une macro-instruction appelée M21. La fin du traitement de rendez-vous déclenche la poursuite des tâches concernées ; MI22 et MI13 dépendent donc de la fin de l'exécution de MI12.

Les graphes de dépendance des tâches T1 et T2 sont :



Les arcs relient MI12 à MI21 et MI22 sont en pointillés car ces dépendances ne sont pas en général connuEs statiquement, c'est à dire lors de la décomposition macroscopique. En effet, durant cette transformation de programme, la dépendance de MI12 avec MI11 et une autre macro-instruction peut être garantie mais l'origine de cette dernière n'est pas connue.

Par exemple, si deux tâches s'exécutant simultanément contiennent des instructions d'appel à la même entrée E, la tâche appelante n'est connue que dynamiquement.

De même, il est possible de garantir que MI12 a deux descendants MI13 et une macro-instruction d'une autre tâche dont le nom n'est connu que lors que de la réalisation du rendez-vous. Ainsi les valeurs initiales des compteurs de dépendance sont connues dès la décomposition macroscopique du programme. Par contre, les traitements de contrôle de MI12 et MI21 doivent utiliser des mécanismes dynamiques pour déterminer les compteurs de dépendance à mettre à jour. Des paramètres relatifs aux décompositions macroscopiques des tâches, comme les identificateurs de compteurs de dépendance à mettre à jour, seront donc communiqués aux tâches de la même façon que les paramètres relatifs aux rendez-vous, lors de la réalisation des instructions "accept" et des "appels".

Les vecteurs d'état des tâches T1 et T2 sont :

V.E de T1		V.E de T2	
[1]	[n-uplet de MI11]	[1]	[n-uplet de MI21]
[2]	[n-uplet de MI12]	[1]	[n-uplet de MI22]
[1]	[n-uplet de MI13]		

Le fonctionnement du système est le suivant :

- Le traitement effectif des macro-instructions MI11 et MI21, exécutable lorsque leur compteur de dépendance est rendu nul du fait de l'activation des tâchesT1 et T2, correspond aux actions précédant le rendez-vous. En particulier, le traitement effectif de MI11 teste une variable booléenne "appel E" caractérisant l'état du rendez-vous, et qui lorsqu'elle est vraie signifie qu'un appel , au moins, de l'entrée E a été exécuté, c'est à dire que la file d'attente propre à l'entrée E n'est pas vide.

Si "appel E" est faux, les informations nécessaires au rendez-vous sont communiquées.

Ainsi, le paramètre de retour de synchronisation, qui, dans le cas de la décomposition envisagée, est l'adresse du mot d'état de la macro-instruction de T1 à décrémenter pour activer le traitement de rendez-vous de l'entrée E lorsque son appel est effectué (c'est à dire @ MI12), est mémorisé.

De même, le traitement effectif de l'atome MI21, contenant l'instruction d'appel de l'entrée E de T1, caractérisant l'état de rendez-vous, qui, lorsque sa valeur est vraie, signifie que celui-ci est dans la situation suivante :

- il n'y a pas d'appel en attente sur le point d'entrée E (la file d'attente était vide)

- et l'instruction accept a été exécutée.

Les paramètres de rendez-vous sont communiqués, comme l'adresse de retour de synchronisation qui dans ce cas de décomposition macroscopique correspond à l'adresse du mot d'état de la macro-instruction de T2 à rendre exécutable, c'est à dire MI12. Cette adresse est propre à chaque élément de chaque file d'attente de chaque entrée.

- Le traitement de contrôle de MI11 permet, lorsqu'un appel est en attente, de rendre exécutable le traitement de rendez-vous correspondant à l'atome MI12. Le compteur de dépendance de MI12 étant égal à 2, le programme de contrôle de MI11 doit effectuer une double décrémentation pour le rendre nul et ainsi libérer MI12 de sa double dépendance.

Si aucun appel n'est en attente, l'atome MI12 perd une dépendance du fait de la fin de l'exécution de M11, cependant par soucis d'optimisation cette libération sera retardée jusqu'au moment de l'arrivée d'un appel.

Le traitement de contrôle de MI11 est donc :

```
if appel then
     [MI12] := [MI12] - 2 ;
endif ;
```

- Le traitement de contrôle de MI21 permet, lorsque l'opération accept de l'entrée E a été exécutée, et qu'il n'y a pas d'appel déjà enregistré, de libérer la macro-instruction MI12 d'une dépendance, l'adresse du mot d'état de celle-ci ayant été communiquée lors du rendez-vous.

Dans les autres cas, le traitement de rendez-vous à exécuter n'est pas encore connu, la macro-instruction correspondante ne peut pas être libérée d'une dépendance, cette action est donc différée jusqu'au moment où l'instruction accept est exécutée.

C'est le traitement de contrôle de la macro-instruction qui contient cette instruction qui se chargera de la libération de la dépendance qui n'a pu être faite. Ceci justifie la double décrémentation réalisée par le traitement de contrôle de MI12.

Le traitement de contrôle de MI12 est :

```
if accept-et-pas-d'appel then
    [MI12] := [MI12] - 2 ;
endif ;
```

Ainsi, lorsque toutes les instructions précédant le point de rendez-vous ont été exécutées et que le traitement de rendez-vous est possible, alors le compteur de dépendance de MI12 est nul et cette macro-instruction est exécutable.

La fin de l'exécution de son traitement effectif autorise la réalisation des tâches T1 et T2, les dépendances de MI12 avec MI13 et MI22 sont donc levées grâce à la décrémentation de leur compteur de dépendance par le traitement de contrôle de MI12. L'adresse du mot d'état de MI22 est connue car celle-ci a été communiquée lors de l'exécution de l'appel.

Une difficulté supplémentaire apparaît du fait que généralement lors de la parallélisation d'une tâche, il n'est pas possible de connaître la structure de décomposition macroscopique des tâches avec lesquelles elle peut être en rendez-vous. Les points de sycnhronisation sont définis dynamiquement il faut donc implémenter des mécanismes permettant de connaître dynamiquement la structure des décompositions macroscopiques autour d'un point de rendez-vous.

Ainsi, pour des raisons d'efficacité, (cf la notion de granularité du parallélisme §1.3) il est possible qu'il soit préférable de ne pas faire apparaître les mécanismes de rendez-vous dans la décomposition macroscopique des programmes, les instructions de prise en compte de rendez-vous sont alors "cachées" à l'intérieur du traitement effectif d'un atome de la même façon que les structures classiques (IF, CASE, LOOP, ...) pouvaient l'être.

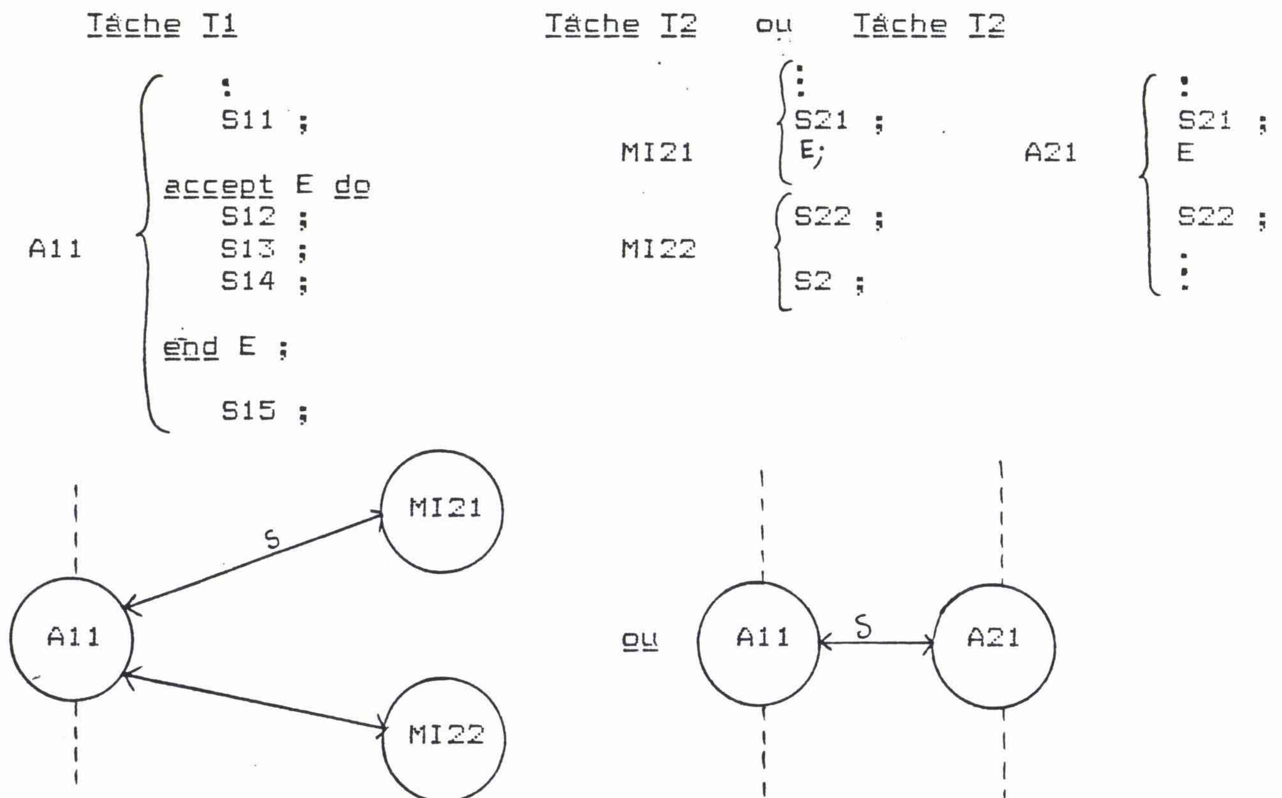
Trois cas doivent être pris en compte : l'instruction accept est cachée dans un atome mais pas l'instruction d'appel d'entrée ; inversement l'instruction d'appel est cachée mais pas l'instruction accept ; enfin les deux types d'instructions sont cachés.

Le problème est de pouvoir indiquer à chaque processus le mode de décomposition qui est utilisé, car les processus qui échangent des informations ne se reconnaissent qu'au moment de l'exécution. Par conséquent, outre les informations nécessaires au rendez-vous, il faut que chaque processus communique à l'autre la façon dont sa décomposition a été réalisée. Deux paramètres booléens, notés I1 et I2, permettent de communiquer l'état de la décomposition macroscopique des tâches T1 et T2. Par convention lorsque I a la valeur fausse alors l'instruction "accept" ou l'instruction d'appel est cachée dans un atome et I prend la valeur vraie quand l'instruction de rendez-vous et prise en compte dans la décomposition macroscopique. A une instruction accept il correspond un indicateur I1, et chaque appel est associé un indicateur de décomposition I2.

Le fonctionnement du système décrit précédemment subit donc quelques modifications :

* lorsque l'instruction accept est cachée dans un atome :

Les dépendances et les graphes de dépendance de ce type sont :



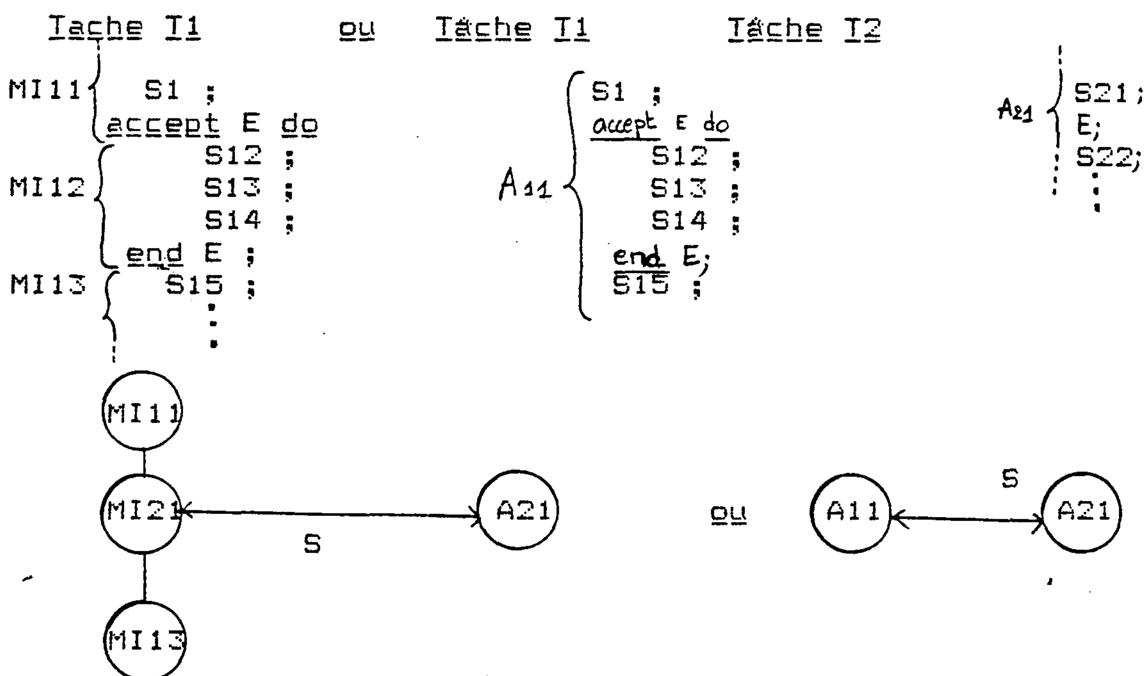
(S indique une relation de synchronisation classique entre deux processus. Ce type de mécanisme est décrit dans (Müh 81)).

- Le traitement de A11 communique lors de l'exécution de l'instruction accept E, toutes les informations utiles au rendez-vous. Si aucun appel de l'entrée E n'a été réalisé alors le paramètre I1 avec la valeur fausse et un paramètre de retour de synchronisation sont communiqués à la tâche appelante, et la tâche appelée est bloquée, sinon, les instructions de A11 sont exécutée.

Lors de l'exécution de l'instruction marquant la fin du traitement de rendez-vous, A11 effectue le traitement de synchronisation suivant afin de réveiller le processus appelant :

```
if I2 then
     [MI22] := [MI22] - 1 ;
else
     déblocage (retour de synchronisation) ;
endif ;
```

* Lorsque l'instruction appel d'entrée est cachée dans un atome les décompositions et les graphes de dépendance de ce type sont :



- Le traitement de A21 communique, lors de l'exécution de l'instruction d'appel, toutes les informations nécessaires au rendez-vous, en particulier, I2 avec la valeur fausse et un paramètre de retour de synchronisation afin de permettre le réveil du processus après le rendez-vous.

Le traitement suivant est exécuté lors d'un appel :

```
if accept-et-pas-d'appel then

    if   I 1        then
    [MI12] := [MI12] - 2

    else
         débloquage (retour de synchronisation);
    endif ;
endif ;
```

Le traitement de A21 est ensuite bloqué.

Le traitement de contrôle de MI12, lorsqu'elle existe, exécute le contrôle adéquat au type de décomposition de la tâche appelante, c'est à dire selon la valeur de I2. il devient :

```
if I2  then
    [MI22] := [MI22] - 1 ;
else
    débloquage (retour de synchronisation) ;
endif ;
    [MI13] := [MI13] - 1 ;
```

De même celui de MI21, lorsqu'elle existe, devient :

```
if accept et pas d'appel then
     if I 1   then
            [MI12] := [MI12] - 2
     else
            débloquage (retour de synchronisation)
     endif ;
  endif ;
```

c) Autres décompositions macroscopiques

La décomposition macroscopique précédente est un modèle simplifié de décomposition, ne faisant d'ailleurs pas apparaitre de parallélisme particulier (autre que celui explicité par les tâches), elle a pour but d'expliquer les mécanismes d'enchainement des macro-instructions autour d'un point de rendez-vous.

A partir de ce mécanisme de base il est possible de déduire d'autres décompositions macroscopiques. Ces nouveaux systèmes se déduisent du schéma précédent selon que les différentes macro-instructions sont divisées en plusieurs macro-instructions parce qu'elles sont trop complexes et parallélisables ou que, au contraire, elles sont de complexité trop faible, ou séquentielles et qu'il soit préférable de les regrouper.

C'est l'outil de parallélisation automatique de programme qui réalise ce travail, le but recherché étant, bien entendu, de dégager un parallélisme dont l'exploitation soit rentable, c'est à dire que l'ensemble des traitements de contrôle qu'il nécessite soit de taille raisonnable par rapport au traitement effectif exécuté.

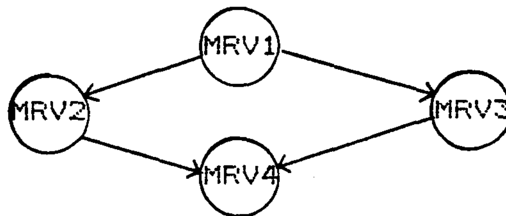
* Décomposition macroscopique du traitement de rendez-vous

Lorsque le traitement de rendez-vous est de taille importante et parallélisable, il est intéressant de pouvoir éclater la macro-instruction MI12, du schéma général, en plusieurs macro-instructions, afin d'exprimer du parallélisme.

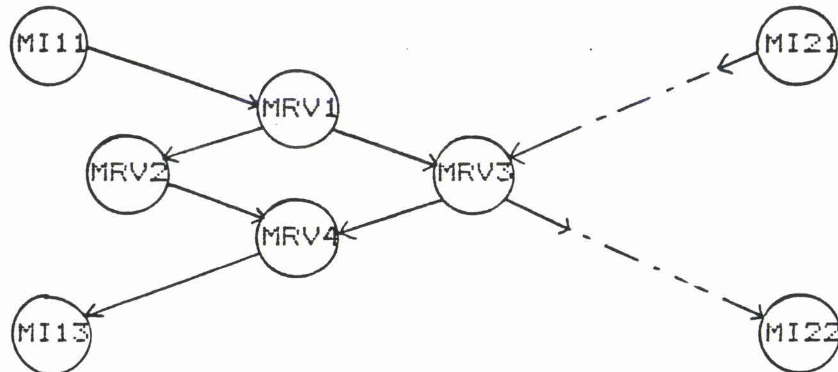
Les relations entre ces différentes macro-instructions sont décrites par un graphe de dépendance et leurs enchainements respectent les règles du modèle Data Flow Macroscopique.

Ce système doit être inclus dans celui de la tâche qui contient le traitement de rendez-vous. Deux cas peuvent se présenter :

- le graphe de dépendance de l'unité de traitement de rendez-vous commence par une macro-instruction initiale et se termine par une macro-instruction terminale, par exemple :



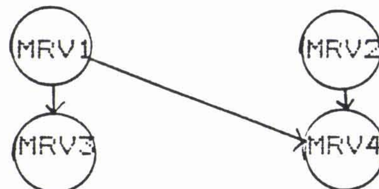
Dans ce cas, il est insérer dans le graphe de la tâche appelante de la façon suivante :



Si aucun appel n'a déjà été enregistré lors de l'exécution de l'instruction accept correspondante par MI11, alors l'adresse du mot d'état de MRV1 est mémorisée, et le compteur de dépendance associé sera décrémenté par le traitement de contrôle de MI21.

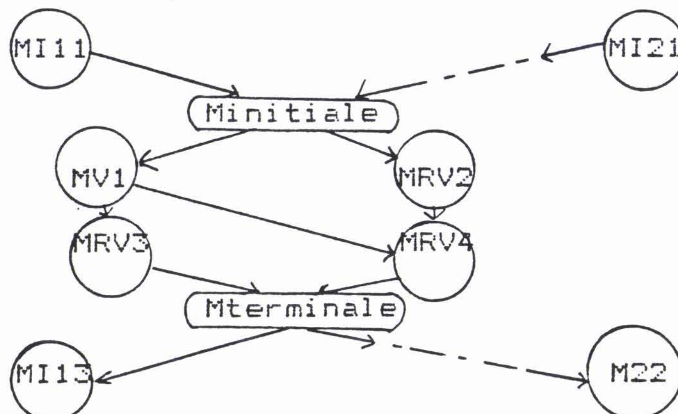
Le traitement de contrôle de MRV4 rend exécutable MI13 et MI22 dont l'adresse du mot d'état aura été communiqué à la tâche appelée lors de l'appel.

- Si le graphe de dépendance de l'unité de traitement de rendez-vous comporte plusieurs macro-instructions en début et/ou en fin de graphe, comme dans cet exemple :



Il faut implémenter un (ou des) molécules jouant le rôle de macro-instruction initiale et/ou terminale et permettant les enchainements avec les macro-instructions aux alentours du rendez-vous.

Le graphe de dépendance devient :

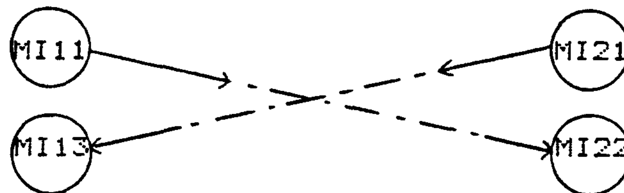


Le principe de fonctionnement de ce système est analogue à celui décrit précédemment.

- Si l'instruction "accept" n'a pas de séquence d'instructions, le rendez-vous est alors une simple synchronisation entre les tâches appelantent appelés.

Le système macroscopique général précédent n'est pas modifié, cependant la macro-instruction appelée MI12 ne contient plus de traitement effectif, c'est alors une molécule dont le seul rôle est d'assurer la synchronisation. Le schéma d'exécution des enchainements est inchangé.

Cependant, une décomposition macroscopique plus simple de la tâche peut être envisagée en supprimant la molécule MI12. Les graphes de dépendance ont alors la forme suivante :



Mais cette décomposition entraine l'exécution, par la macro-instruction MI21, d'un traitement de contrôle particulier :

```
if accept-et-pas-d-appel then
     [MI13] := [MI13] - 2 ;
     [MI22] := [MI22] - 1 ;
endif ;
```

Le choix de ce traitement de contrôle est déterminé en cours d'exécution en fonction des paramètres mémorisés par la tâche appelante au ours de l'exécution de l'instruction accept. L'adresse du mot d'état MI13 (MI13) et un indicateur de décomposition macroscopique font partie de ces paramètres et permettent de réaliser le traitement de contrôle adéquat.

Ainsi, cette dernière solution augmente la quantité d'informations à communiquer entre tâches et le nombre de test à réaliser en cours d'exécution afin de déterminer le traitement de contrôle adéquat. La première méthode implique, quant à elle l'activation d'une molécule de synchronisation.

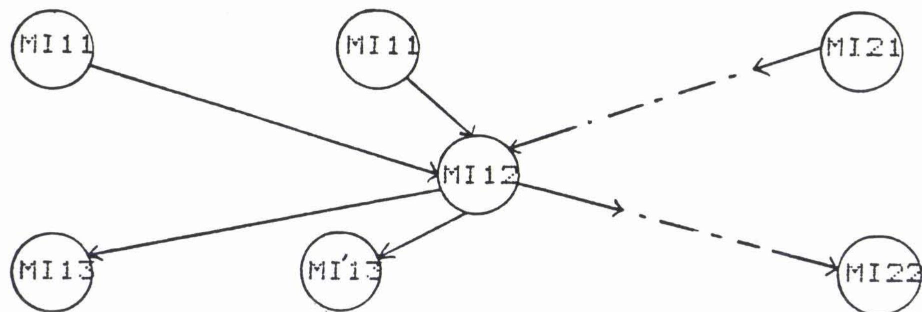
* Décompositions macroscopiques des traitements de la tâche appelée :

Les macro-instructions MI11 et MI13 peuvent, selon leur complexité et leur parallélisme exploitable, être décomposées en plusieurs macro-instructions dont les relations sont déterminées par des (sous) graphes de dépendance. Il est nécessaire que tous les traitements avant le rendez-vous dans le programme canonique soient achevés au moment de la synchronisation. Par conséquent, la macro-instruction initiale du traitement de rendez-vous dépend de toutes les macro-instructions terminales du sous-graphe de dépendance décrivant la tâche appelante avant le rendez-vous. Le compteur de dépendance de cette macro-instruction initiale est initialisé en fonction de leur nombre.

Symétriquement, l'exécution de la tâche appelée ne se poursuit que lorsque le traitement de rendez-vous est achevé, donc toutes les macro-instructions initiales du traitement du sous-graphe de dépendance de la tâche appelée après le rendez-vous dépendent de la macro-instruction terminale du traitement de rendez-vous.

La décomposition des traitements avant et après le traitement de rendez-vous est effectuée en respectant les règles de décomposition du modèle Data Flow Macroscopique.

Par exemple un graphe de dépendance de ce type peut être :



Dans ce cas le compteur de dépendance de la macro-instruction de traitement de rendez-vous MI12 est initialisé à 3.

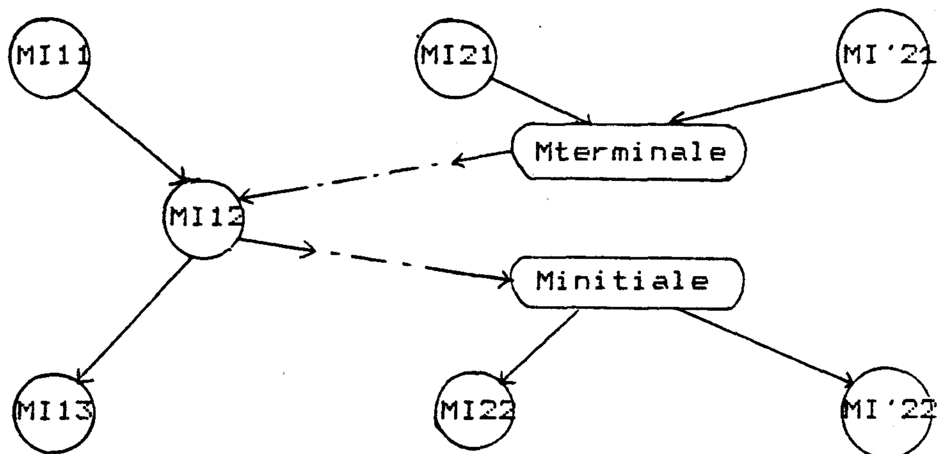
* Décompositions macroscopiques des traitements de la tâche appelante :

Les macro-instructions MI21 et MI22 correspondant aux traitements de la tâche appelante avant et après le rendez-vous peuvent être décomposées en plusieurs macro-instructions dont les relations sont décrites par des sous-graphes de dépendance.

Le sous-graphe correspondant au traitement de la tâche avant le rendez-vous doit se terminer pour une macro-instruction terminale qui est soit une macro-instruction finale dans le sous-graphe soit une molécule qui a été ajoutée afin de marquer l'achèvement de l'exécution de cette partie. C'est cette macro-instruction terminale qui assure la communication et la synchronisation avec la tâche appelée, son traitement de contrôle est semblable à celui de MI21 dans le modèle général décrit en début de cette section.

Symétriquement le sous-graphe correspondant à la partie de la tâche appelante après le rendez-vous doit commencer par une macro-instruction initiale (M initiale) qui est, soit une macro-instruction initiale de sous graphe, soit une molécule qui a été rajoutée pour marquer le début de celui-ci. L'adresse du mot d'état de cette macro-instruction a été communiqués à la tâche appelée lors de l'exécution de l'instruction d'appel. Le traitement de contrôle de la macro-instruction terminale de l'unité de traitement de rendez-vous de la tâche appelée décrémente le compteur de dépendance de Minitiale pour la rendre exécutable.

Le graphe de dépendance de ce type de décomposition est par exemple :



D'autres décompositions et techniques d'enchainements des macro-instructions aux environs d'un point de rendez-vous sont envisageables. Ainsi, dans l'exemple précédent, au lieu d'utiliser une molécule jouant le rôle de macro-instruction initiale de la partie de traitement de la tâche appelante après le rendez-vous, il peut être envisagé de passer en paramètre, les adresses des mots d'état des macro-instructions MI23 et MI24 et de rendre celles-ci dépendantes de MRV4.

Le choix d'une méthode ou d'une autre nécessiterait un travail de simulation qui n'est pas l'objet de cette thèse.

Il est important de constater la souplesse de fonctionnement du modèle qui à chaque problème permet d'envisager plusieurs solutions.

3.2.1.2 Prise en compte des appels conditionnels d'entrée

Un appel conditionnel d'entrée effectue un appel d'entrée qui n'a lieu que si un rendez-vous est immédiatement réalisable (USG 83 §9.7.2).

La syntaxe est la suivante :

```
select
        appel-d-entrée
            (suite d'instructions)
else
    suite-d-instructions
end select ;
```

Si l'appel d'entrée n'a pas lieu, les instructions de la partie "else" sont exécutées ; sinon le rendez-vous a lieu et les instructions qui suivent éventuellement l'appel sont exécutées.

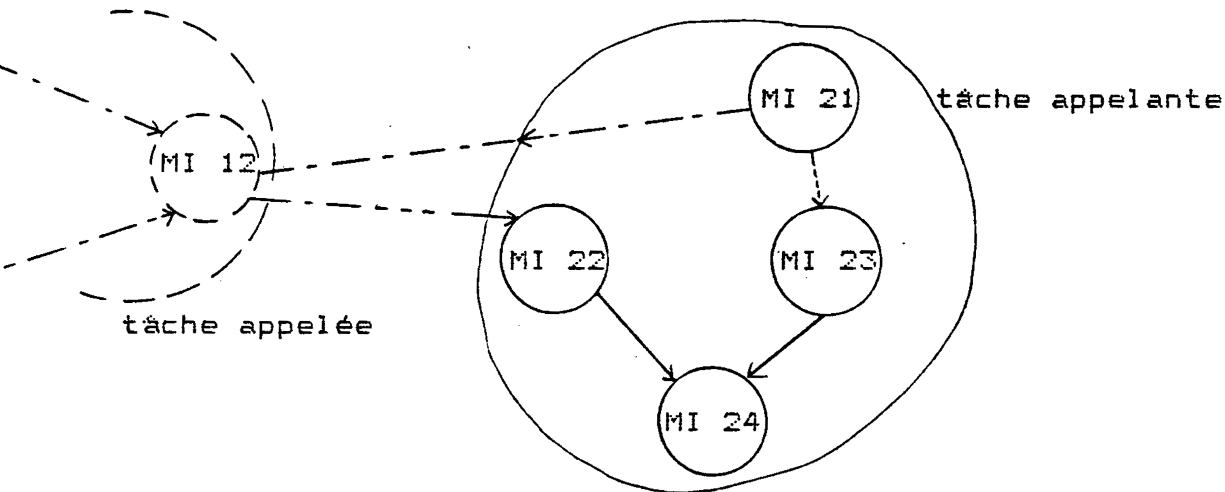
Ce type de traitement correspond à la conjugaison d'un appel d'entrée et d'un "if", aussi les méthodes de décompositions macroscopiques et d'enchainements des différents macro-instructions découlent de celles utilisées pour ces deux types d'instructions.

La description du fonctionnement d'un tel système est effectuée à partir d'un modèle de décomposition générale comme au paragraphe 3.2.1.a. D'autres décompositions pouvant être facilement déduites de ce modèle (§3.2.b) mais elles sont pas détaillées ici.

Soit la tâche suivante et sa décomposition macroscopique.

```
     tâche appelante
             :
             Si ;
MI 21        select
                   E ;
MI 22              Sj ;
                   :
             else
                   Sk ;
MI 23              :
             end select ;
                   Sl ;
MI 24              :
```

Le graphe de dépendance de cette tâche est :



La macro-instruction MI12 est la macro-instruction représentant le traitement de rendez-vous de la tâche appelée.

L'exécution de MI12 dépend d'un appel d'une tâche appelante (arc MI21 ⟶ MI12) et MI22 dépend de l'achèvement du traitement de rendez-vous.

Les arcs reliant MI21, MI12 et MI23 sont en pointillés car la levée de la dépendance que chacun représente est conditionnelle (rendez-vous immédiatement réalisable ou non) et ne dépend pas seulement de l'achèvement du traitement effectif de MI21.

C'est pourquoi, le compteur de dépendance de MI24 est initialisé à 1 puisque son exécution dépend soit de MI22, soit de MI23.

Le vecteur d'état de la tâche appelante est :

[1] [n-uplet de MI21]
[1] [n-uplet de MI22]
[1] [n-uplet de MI23]
[1] [n-uplet de MI24]

Le fonctionnement du système est le suivant :

- Durant l'exécution du traitement effectif de MI21 l'appel de l'entrée E est effectué. Si le rendez-vous est possible immédiatement parce la valeur du booléen "accept-et-pas-d-appel" est vrai, alors les paramètres nécessaires au rendez-vous, comme l'adresse de retour de synchronisation, qui dans ce cas de décomposition macroscopique correpond à l'adresse du mot d'état de la macro-instruction de la tâche appelante à rendre exécutable, c'est à dire à MI22, sont communiqués, sinon l'appel n'est pas enregistré.

- Le traitement de contrôle de MI21 réalise :

<u>if</u> accept-et-pas-d'appel" <u>then</u>

[MI12] := [MI12] - 2 ;

<u>else</u>

[MI23] := [MI23] - 1 ;

<u>endif</u> ;

Lorsque "accept-et-pas-d-appel" est vrai, l'adresse du mot d'état de MI12 est connue de la tâche appelante puisque l'exécution de l'instruction accept de l'entrée E a été réalisée entrainant la communication de MI12. MI23, est rendue exécutable par le traitement de contrôle de MI21 lorsque le rendez-vous n'est pas immédiatement réalisable.

- MI22 est rendue exécutable par le traitement de contrôle de MI12, si le traitement rendez-vous a été exécuté. Dans ce cas l'adresse du mot d'état de MI22 est connue de la tâche appelée car elle a été communiquée lors de l'appel.

Si le traitement effectif correspondant à MI22, dans la décomposition présentée, est vide car l'alternative de la partie else n'existe pas, alors le retour de rendez-vous ce fait sur MI24 dont l'adresse aura été auparavant communiquée lors de l'appel.

Enfin, si le traitement optionnel correspondant à MI23 n'existe pas dans le programme, la macro-instruction MI24 dépend directement de MI21.

- MI24 est rendue exécutable, selon le chemin choisi, par MI22 ou MI23, s'elles existent, sinon par MI12 ou MI21.

3.2.1.3 Prise en compte des appels d'entrée avec attente limitée

Un appel d'entrée avec attente limitée exécute un appel d'entrée qui est annulé si le rendez-vous n'a pas lieu en un temps donné. (USG 83 §9.7).

La syntaxe est la suivante :

```
appel-avec-attente-limitée ::=

select
     appel-d'-entrée

         [séquence d'instructions]

or

     instruction-delay [séquence d'instructions]

end select ;
```

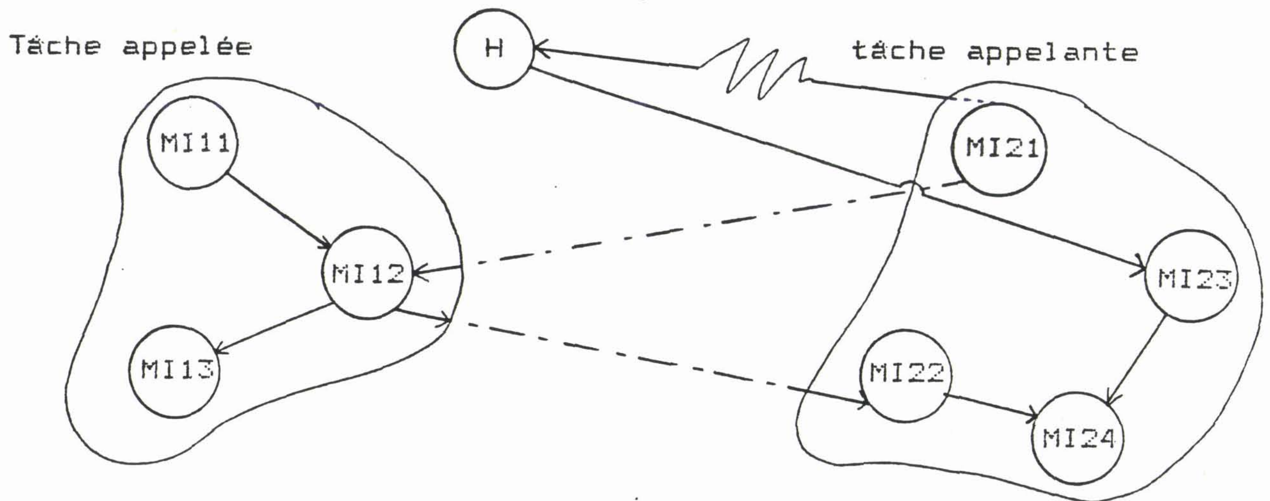
S'il est possible de réaliser un rendez-vous dans le délai spécifié, il est réalisé et la suite d'instructions qui suit éventuellement l'appel est ensuite exécutée, sinon l'appel est annulé et l'alternative "delay" est exécutée.

Le découpage macroscopique général autour d'un tel mécanisme est :

```
Tâche appelante
             :
MI21        { Si
              select
                   E ;
MI22        {      Sj;
                   :
              or
                   delay X ;
MI23        {      Sk ;
              end select ;
MI24        { Sl ;
              :
```

Ce type de traitement met en oeuvre un processus horloge spécialisé H qui gère les temps d'attente des processus en demande de rendez-vous. Lorsque le temps d'attente d'un processus devient nul, H déclenche les actions de réveil que cela implique.

Le graphe de dépendance du système est :



Comme auparavant, les arcs issus de MI21 sont en pointillés car ils correspondent à des branchements conditionnels.

* Fonctionnement de ce modèle général :

- Le processus MI21 exécute l'instruction d'appel d'entrée E à attente limitée, qui réalise le traitement suivant :

communication des informations nécessaires au rendez-vous, en particulier MI22 et un indication D précisant que l'appel est en attente limitée ;

évaluation du booléen "accept-et-pas-d-appel" ;

<u>if</u> not (accept-et-pas-d'appel) <u>then</u>

communication au processus horloge des informations permettant de réaliser les actions nécessaires lorsque l'attente de l'appel est expirée. En particulier, MI23, X le délai d'attente initiale, et l'adresse dans la file d'attente associée à l'entrée E des informations relatives à cet appel ;

<u>endif</u> ;

- Le traitement de contrôle de MI21 réalise :

<u>if</u> accept-et-pas-d'appel <u>then</u>

[MI12] := [MI12] - 2 ;

<u>endif</u> ;

- L'exécution du traitement de contrôle de M11 réalise :

if appel then

if D then

supprimer les informations concernant l'appel à attente limitée communiquée au processus H ;

endif ;

[MI12] := [MI12] ∸ 1 ;

else

communication des informations nécessaire au rendez-vous lorsqu'un appel se produit, en particulier @ MI12 ;

endif ;

- lorsque le temps d'attente d'un appel arrive à expiration le processus horloge exécute :

supprimer les informations concernant l'appel de la file d'attente de l'entrée E ;

[MI23] := [MI23] - 1 ;

- la macro-instruction MI24 dépend de l'achèvement de, soit MI12, soit MI23, son compteur de dépendance est donc initialisé à 1 et est décrémenté par le traitement de contrôle de l'une de ces macro-instructions.

A partir de cette décomposition générale et du mécanisme d'enchainement des macro-instructions décrits, il est possible d'exprimer de nombreux cas de parallélisme. Selon que ces différentes macro-instructions sont de taille suffisante et parallélisables, elles peuvent être divisées en plusieurs macro-instructions dont les enchainements sont décrits pour des sous-graphes de dépendance.

3.2.1.4 Prise en compte de l'instruction "select" :

Une instruction "select" permet de réaliser une construction d'attente et de choix entre plusieurs alternatives.

La syntaxe est la suivante :

```
attente selective ::=

select
     alternative-de-"select"
{or
     alternative-de-"select"}
[else
     séquence-d-instructions]
end select ;

alternative-de-"select"::=
    [when condition ===>]
     alternative-d'attente-selective

alternative-d'attente-selective::=alternative-
accept/alternative-délai/alternative-terminate

alternative-accept::=instruction accept[séquence
d'instructions]

alternative-délai::=instruction délai [séquence
d'instructions]

alternative-terminate::=terminate ;
```

Une alternative de "select" n'est sélectionnée que si elle est ouverte, c'est à dire quand elle n'est précédée d'aucune clause, ou si la condition correspondante est vraie.

Le langage spécifie que :

(a) une attente sélective peut contenir au plus une alternative terminate ;

(b) si une attente contient une alternative terminate elle ne peut pas contenir d'alternative délai ;

(c) une attente sélective contient au moins une alternative commençant par une instruction accept ;

(d) une partie else n'est autorisée que s'il n'existe aucune alternative terminate ou délai.

Ces spécifications conduisent à quatre types différents d'attente sélective :

(a) l'attente sélective contenant seulement des alternatives accept sans partie else

(b) l'attente sélective contient des alternatives accept et une partie else

(c) l'attente sélective contient des alternatives accept et des alternatives délai

(d) l'attente sélective contient des alternatives accept et une alternative terminate.

Ceci permet de décomposer le problème de l'attente sélective en quatre sous-cas, ce qui en facilite l'approche.

Les décompositions proposées par la suite sont générales, elles permettent de décrire le principe de fonctionnement du Data Flow Macroscopique appliquée aux attentes sélectives.

De ce modèle général, il est possible de déduire d'autres décompositions macroscopique, en appliquant les principes de décomposition ou de regroupement des différentes macro-instructions évoqués au paragraphe 3.3.2.3.

a - Attente sélective avec uniquement des alternatives accept

Les décompositions générales d'une tâche appelée comportant ce type de mécanisme et celles des tâches appelantes correspondantes sont :

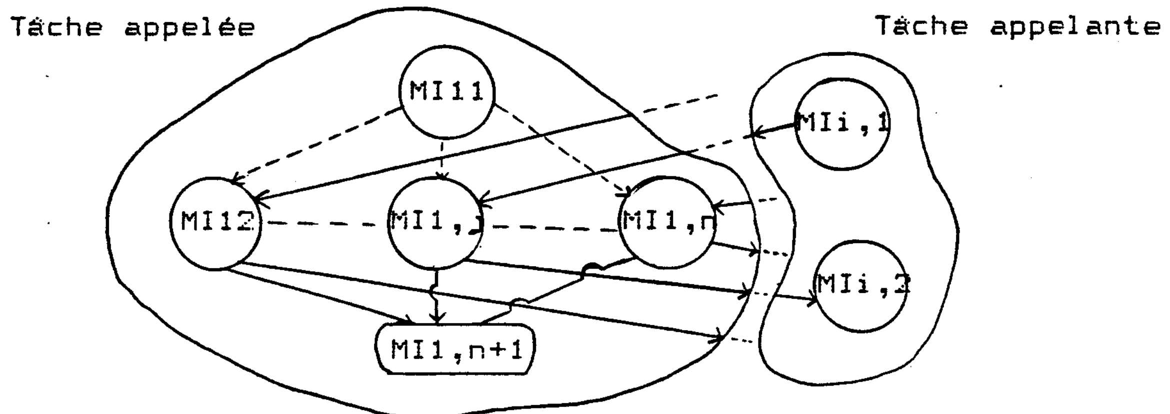
Tâche appelée

```
        :
MI11    S1i ;
        select
          when condition 2 =>
               accept E2 do
MI12           S1j ;
            :  :
          when condition n =>
MI11           accept En do
MI1,n          S1k ;
          end select ;
MI1,n+1 S11 ;
        :
```

Tâche appelante i

```
          :
MIi1      S21 ;
      Ei ;

MIi2      S22;
          :
```

Le graphe de dépendance est :



Les arcs issus de MI11 ---> sont en pointillés car la levée de la dépendance que chacun représente est conditionnelle. Seule une alternative est choisie après après le rendez-vous. Les arcs MIi,1 --> MIi,j comportent une partie en pointillé car la relation de dépendance entre ces deux macro-instructions est déterminée dynamiquement durant l'exécution du programme.

Le traitement effectif de MI11 réalise, lors de l'exécution de l'instruction "select", les opérations nécessaires au choix de l'une des alternatives. Ce travail consiste en particulier à :

- évaluer les gardes (si ils existent) qui contrôlent l'exécution des alternatives

- déterminer parmi les alternatives ouvertes, celle(s) dont la file d'attente comporte au moins un appel et choisir l'une d'entre elles.

- si aucune des alternatives ouvertes n'a été appelée, alors placer celle(s)-ci en état commun d'attente d'un appel après sélection. A chaque alternative ouverte correspond l'adresse du mot d'état de la macro-instruction traitement de rendez-vous associée. De plus un booléen "sélection" propre à la tâche appelée prend la valeur VRAI, il permet d'indiquer l'état du rendez-vous aux tâches appelantes et de déclencher les opérations propres à l'état de "sélection" lors d'un appel.

Ces opérations de sélection peuvent être facilitées par la création, à la compilation ou parfois à l'exécution, d'une table de sélection regroupant et décrivant les états des différentes alternatives. Cette technique correspond à celle utilisée par (Ren 81). Il suffit alors d'ajouter à chaque entrée de la table, l'adresse du mot d'état de la macro-instruction correspondant à l'alternative associée, ainsi que le montre le chapitre V.

En résumé le traitement de contrôle de MI11 exécute :

sélectionner;

<u>if</u> alternative sélectionnée <u>alors</u>
[MI-sélectionnée]:=[MI-sélectionnée] - 2

<u>else</u>

Sélection := VRAI ;

associer à chaque alternative ouverte l'adresse du mot d'état de sa macro-instruction ;

<u>endif</u> ;

Pour la tâche appelante, le traitement de contrôle de la macro-instruction MIi1, qui contient l'instruction d'appel à une entrée Ei, examine si l'alternative associée est ouverte, si c'est le cas toutes les autres alternatives sont fermées. Ce traitement de contrôle assure ensuite les opérations permettant les enchaînements avec la suite du programme, comme cela a été montré précédemment.

Les traitements de la macro-instruction correspondant à l'alternative sélectionnée sont analogues à ceux déjà décrits pour le traitement des macro-instructions correspondant aux traitements de rendez-vous, de plus le booléen "sélection" est réinitialisé à FAUX et les informations concernant l'opération de sélection (table de sélection) sont détruites.

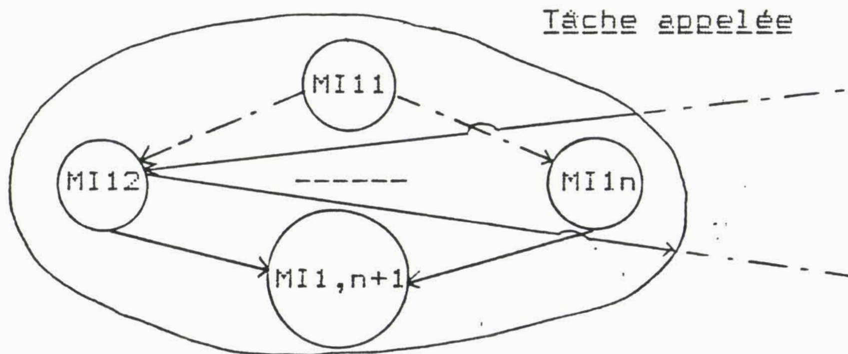
b - <u>Attentive sélective avec une alternative else</u>

L'alternative "else" d'une attente sélective est choisie si aucune alternative accept ne peut être immédiatement choisie.

La décomposition macroscopique générale et le graphe de dépendance d'un traitement de ce type sont :

Tâche appelée

```
MI11      { S11 ;
          { select
          {       when condition 2 ==>
          {                    Accept E2 do
MI12      {                     S12 ;
          {
          { :
          :
                  else
MI1n      {                    S1n ;
          {
          end select ;
MI1n+1    { S1,n+1
          {
```



Les traitements sont analogues à ceux décrits pour traiter une attente avec uniquement des alternatives accept. Lors de l'exécution de l'opération de sélection, si l'alternative else est sélectionnée, alors le traitement de contrôle de MI11 associe à "MI-sélectionné" l'adresse @ MI1n correspondant à l'adresse du mot d'état de la macro-instruction associée à l'alternative else.

c - Attente sélective avec une alternative délai

Une alternative délai ouverte dans une attente sélective est sélectionnée si aucune alternative accept n'est sélectionnée durant le délai spécifié.

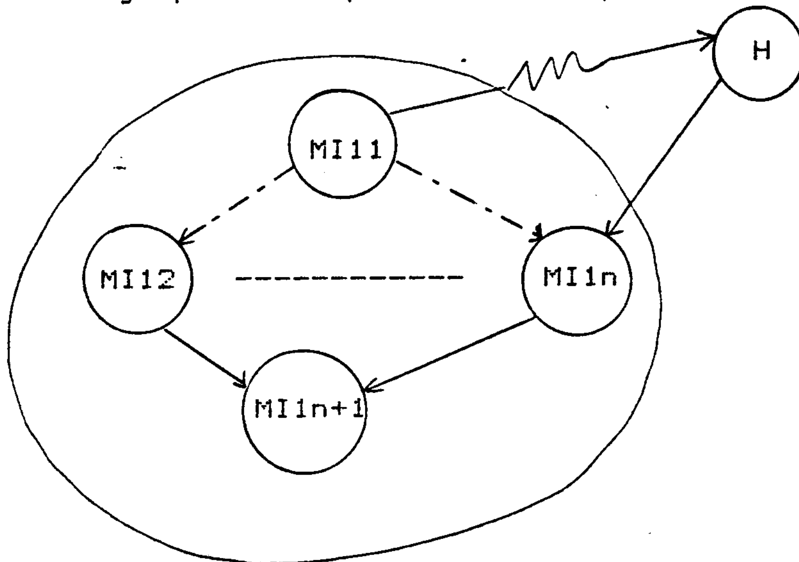
Le découpage macroscopique générale d'une partie de programme de ce type est :

Tâche appelée :

```
             ⎧ .
             ⎪ .
             ⎪ S11
MI11         ⎨ Select
             ⎪      when condition 2 ==>
             ⎩           Accept E do
MI12         { .    S12 ;
               .
               .
MI11         {      when condition n ==>
                    delay X ;
MI1n         {      S1n ;
               end select ;

MI1n+1       { S1n+1 ;
               .
               .
```

Ce type de traitement met en oeuvre un processus horloge H qui gère les temps d'attente des processus en demande de rendez-vous. Lorsque le temps d'attente d'un processus devient nul H déclenche le réveil de ce processus. Ainsi, l'activation d'une alternative délai d'un rendez-vous avec attente sélective limitée est-elle effectuée lorsque le temps d'attente associé devient nul.

Le graphe de dépendance du système est :



- Le traitement de contrôle de MI11 exécute :

```
sélectionner ;
if alternative - sélectionnée then
   [MI sélectionnée]:=[MI-sélectionnée - 2]
else
      Sélection := VRAI ;
      Délai     := VRAI ;
      Associer à chaque alternative l'adresse du mot d'état
      de sa macro-instruction ;
      Communication au processus horloge des informations
      nécessaires au réveil d'un processus lorsque le temps
      d'attente associé deviendra nul.
      En particulier, les paramètres @ MI1n, X le délai et un
      identificateur de la tâche appelée sont communiqués;
endif ;
```

- Le traitement de contrôle de la macro-instruction MI21, qui contient l'instruction d'appel de la tâche appelante, exécute :

```
if Selection            then
      if E dans table de selection then
           if Délai   then
                retirer les informations, relatives à cette
                sélection à attente limitée, du processus H ;
                endif ;
                fermer les autres alternatives du select ;
      endif ;
endif ;
exécution du traitement de contrôle relatif aux instructions
d'appels (décrit précédemment) ;
```

Les traitements de contrôle des macro-instructions de la tâche appelante sont inchangés.

Si le temps d'attente de la sélection arrive à expiration le processus horloge exécute :

```
if alternative délai ouverte then

      sélection := FAUX ;
      supprimer les informations associées à l'attente
      sélective, lire l'adresse du mot d'état de la macro-
      instruction associée à l'alternative délai (@MI1n) ;
      [MI1n] := [MI1n] - 1 ;
      supprimer les informations relatives à cette attente ;

endif;
```

d - Attente sélective avec une alternative terminate

Dans ce cas, le découpage macroscopique du programme et le principe d'enchainement des différentes macro-instructions sont identiques à ceux où l'attente sélective se fait uniquement sur des alternatives accept. Lorsqu'une alternative terminate est ouverte et qu'aucun rendez-vous n'est immédiatement possible la tâche est dans un état "prêt à terminer". Elle passe à l'état "terminée" si elle dépend d'un maître dont l'exécution est achevée et si toutes les tâches qui dépendent du maître considéré sont soit déjà terminées soit de façon similaire en attente sur une alternative "terminate" ouverte d'une instruction "select" (USG §9.4). Ces actions effectuées sur l'état d'une tâche ne déclenche pas des traitements effectifs programmés par l'utilisateur mais engendre sa terminaison, c'est à dire que les espaces systèmes alloués sont libérés.

Lorsque la tâche est dans l'état prêt à terminer et qu'un appel d'une de ses entrées est généré, elle repasse à l'état actif et la macro-instruction de l'alternative de l'entrée sélectionnée du rendez-vous et rendue exécutable par la tâche appelante grâce aux techniques décrites précédemment.

3.2.3 Prise en compte des mécanismes d'exception

Une construction du langage Ada permet à l'utilisateur d'associer à des unités comme les sous-programmes, blocs ou tâches, des instructions ayant pour objet de traiter les erreurs logicielles ou matérielles, et certaines situations exceptionnelles pouvant être détectées durant l'exécution d'un programme.

La syntaxe est la suivante :

```
traitement-exception ::=
     when choix-d-exception { /choix d'exception} ==>
          séquence-d-instructions

     choix-d-instruction ::= nom d'exception /others
```

Ce "traitement-exception" est inclus dans le corps d'une unité de la façon suivante :

```
begin
     séquence-d-instructions
exception
     traitement-d-exception
    {traitement-d-exception}
end
```

Dans une unité, une "exception" peut être levée par une instruction "raise" ou par une instruction provoquant une exception prédéfinie.

Une exception levée durant l'exécution du corps d'une unité déclenche l'abandon du traitement de celle-ci et le branchement au traitement de l'exception correspondant.

Lorsque l'unité contient ce traitement, celui-ci est exécuté, sinon l'exception est propagée à l'unité appelante, sauf pour un corps de tâche, auquel cas la tâche est achevée.

Une exception levée durant l'élaboration des déclarations d'une unité, engendre l'abandon du traitement de celle-ci et la propagation de l'erreur à l'unité appelante, sauf pour les tâches qui deviennent alors achevées.

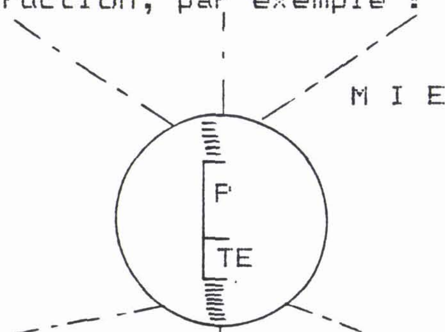
a - <u>Découpage Macroscopique</u>

La partie traitement des exceptions d'une unité peut, selon sa complexité et son parallélisme inhérent, être cachée dans un atome ou apparaître sous forme de une ou plusieurs macro-instructions dont les enchainements sont décrits par un graphe de dépendance. Dans ce dernier cas le graphe, correspondant à la partie traitement des exceptions, commence nécessairement par une macro-instruction initiale dont le rôle est d'assurer la propagation ou la sélection du traitement correspondant à l'exception.

En outre, l'achèvement de l'unité est conditionné par l'achèvement du traitement d'exception lorsque celui-ci est sélectionné. Aussi dépend-il des macro-instructions terminales du graphe du traitement de l'exception considérée.

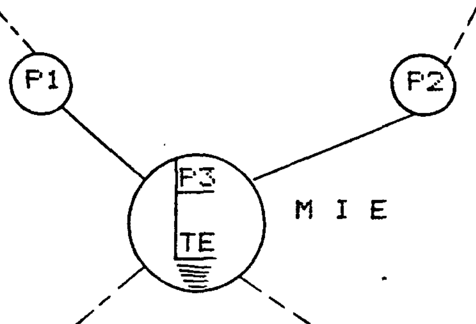
Plusieurs décompositions macroscopiques peuvent être distinguées :

1 - une unité P et ses traitements d'exceptions sont cachés dans une macro-instruction, par exemple :

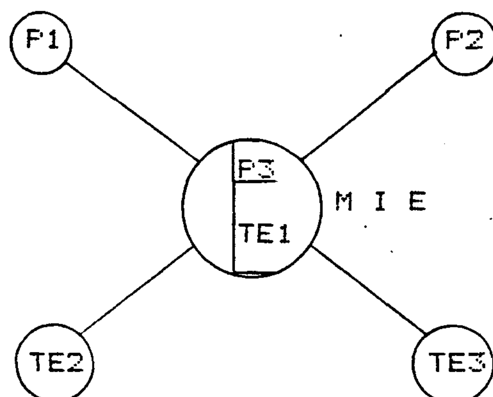


([P représente le code de l'unité de P et [TE représente le code de la partie traitements d'exceptions).

2 - l'unité P est décomposée en plusieurs macro-instructions Pi et ses traitements d'exceptions sont cachés dans un atome, par exemple :



3 - la partie traitement des exceptions est décomposée en plusieurs macro-instructions TEi dont l'une, MIE, à la qualité d'être initiale dans le graphe de dépendance de la partie traitement des exceptions, par exemple :



Ainsi une unité avec traitement des exceptions est caractérisée par une macro-instruction MIE, contenant la partie initiale des traitements d'exceptions, positionnée à une adresse @TE dans le code associé à MIE.

b - traitement d'une exception

La levée d'une exception dans une unité déclenche la suspension de son exécution ce qui engendre des problèmes nouveaux lorsque celle-ci est décomposée en plusieurs macro-instructions. Il s'agit, d'une part, de suspendre les traitements propres à l'unité où s'est produite l'exception, ceux-ci pouvant être contenus dans une ou plusieurs macro-instructions exécutables ou en exécution, et, d'autre part, suspendre l'ensemble des traitements des unités que celle-ci appelle directement ou indirectement et qui sont activées (exécutables ou en exécution) parallèlement.

Il faut donc déterminer parmi l'ensemble des macro-instructions exécutables ou en exécution, celles qui doivent être désactivées.

Une difficulté supplémentaire provient du fait que les appels de sous-programmes sont définis dynamiquement et que, par conséquent, les macro-instructions à désactiver doivent être déterminées dynamiquement.

La suspension d'une unité et des unités qu'elle appelle nécessite la mise en oeuvre de structures de données gérées dynamiquement, contenant les informations suffisantes pour permettre de déterminer les macro-instructions à désactiver.

En outre, lorsqu'une exception est propagée, la propagation s'effectue suivant un chemin défini par les différentes activations d'unités. Ce chemin est déterminé dynamiquement et est fonction de l'endroit du programme où l'unité, où s'est produite l'exception, a été activée.

Ainsi, lorsqu'une exception est levée dans un programme, l'ensemble des unités devant être suspendues, ainsi que la partie traitement des exceptions qui traite cette exception, sont définis dynamiquement.

L'exploitation du parallélisme, lorsque des mécanismes de traitement d'exception sont mis en oeuvre, engendre des problèmes complexes qui peuvent être résolus grâce à un superviseur puissant, dont la mise en oeuvre est coûteuse en temps d'exécution du fait des mécanismes dynamiques implantés. Notre but n'est pas de décrire un tel système mais de montrer que grâce à un découpage macroscopique adéquat des programmes et moyennant quelques restrictions sur l'exploitation du parallélisme, une prise en compte plus simple des exceptions peut être envisagée.

Pour décrire le principe de base de cette méthode il est nécessaire de définir le concept d'"Unité Logique Exception" (ULE). Par la suite une "unité logique exception" est un ensemble d'unités composé d'une unité avec une partie traitement des exceptions et de toutes les unités sans traitement d'exception que celle-ci englobe, (c'est à dire qu'elle appelle directement ou indirectement).

Soit l'exemple 11.4.1.13 du manuel Ada :

```
procedure P is
     ERROR : exception;
     procedure R;

     procedure Q is
     begin
     ..R;
       ...                  - error situation (2)
     exception
       ...
       when ERROR ==>       - handler E2
       ...
     end Q;

     procedure R is
     begin
       ...                  - error situation (3)
     end R;

begin
     ...                    - error situation (1)
     Q;
     ...
exception
     ...
     when ERROR ==>         - handler E1
     ...
end P
```

Selon notre définition l'exécution de P correspond à l'exécution d'une ULE et l'exécution des procédures Q et R correspond à l'exécution d'une autre ULE.

Du fait des mécanismes de propagation, si une exception se produit dans une unité logique exception, alors toutes les unités qui la composent doivent être suspendues.

Ainsi, si le découpage macroscopique est tel que, à tout instant, pour une tâche, seules les macro-instructions appartenant à une même unité logique exception sont exécutables, ou en exécution, alors, lorsqu'une exception est levée dans la tâche, toutes les macro-instructions actives pour elles doivent être désactivées. L'outil de transformation de programme doit être conçu pour produire ce type de décomposition.

Dans l'exemple précédent une exception levée en "error-situation (1)" est traitée par "handler E1" et une exception levée en "error situation (2)" ou "error situation (3)" est traitée par "handler E2". La décomposition macroscopique doit garantir que l'exécution des macro-instructions appartenant à Q et R ne se fait pas simultanément avec celles appartenant à P.

Cette technique permet de simplifier de façon importante le travail du système lorsqu'une exception est levée dans une tâche, puisqu'il lui suffit de déterminer les macro-instructions exécutables ou en exécution pour la tâche et de les désactiver. De plus, cela n'augmente pas la quantité d'informations à mémoriser puisque le système doit disposer de tels renseignements dans ses tables pour gérer les ressources.

Basées sur cette décomposition macroscopique deux méthodes de prises en compte des exceptions ont été étudiées. Leur principe de fonctionnement est présenté ci-dessous. La mise en oeuvre de telles techniques est décrite au chapitre 5.3, où il est montré que des associations MIE, @TE et unité logique exception peuvent être réalisées sans pénaliser, en temps d'exécution, un programme où aucune exception n'est levée.

* <u>solution 1</u>

La décomposition macroscopique est telle qu'à tout instant toutes les macro-instructions exécutables ou en exécution appartiennent à la même "unité logique exception". Lorsqu'une exception est levée dans une de ces macro-instructions, un appel système est effectué, et les macro-instructions actives pour la tâche sont tuées. Pour ce faire le système utilise les informations contenues dans les tables qui lui sont nécessaires pour assurer la gestion des processus.

De plus, il détermine, d'une part, l'adresse du mot d'état de la macro-instruction MIE qui contient la partie initiale des traitements d'exceptions de l'unité logique exception où l'exception a été levée, et, d'autre part, l'adresse de cette partie dans le code de MIE (voir une mise en oeuvre possible au §5.3). A chaque unité logique exception une macro-instruction MIE et une adresse @TE sont associées.

La macro-instruction MIE est alors activée et son exécution commence à l'adresse @TE. Si l'exception peut être traitée par un des traitements des exceptions de l'ULE celui-ci est activé. La fin de son exécution déclenche l'achèvement de l'unité logique exception et donc l'exécution du traitement de contrôle permettant les enchainements avec la suite du programme. Si l'exception ne peut être traitée, elle est alors propagée à l'unité logique exception appelante.

Comme auparavant, un appel système est effectué dans le but de déterminer l'adresse de la MIE et l'adresse @TE pour cette nouvelle ULE. Le procédé est ainsi itéré jusqu'à ce qu'un traitement pour l'exception soit trouvé. Lorsque la propagation est effectuée dans le corps d'une tâche et que celle-ci ne peut pas la traiter alors qu'elle se trouve en "rendez-vous", l'exception est alors propagée chez l'appelant.

Cette solution peut paraître restrictive dans la mesure où elle n'autorise que l'exécution d'une seule ULE par tâche, à chaque instant. Cependant, elle doit être comprise comme une technique de base pouvant subir de nombreux aménagements afin d'exprimer un parallélisme n'entrainant pas la mise en oeuvre d'un système trop complexe qui limiterait l'intérêt de l'exploitation du parallélisme.

Ainsi par exemple, il est possible d'envisager l'exécution simultanée de plusieurs ULE lorsque chacune d'elles est entièrement contenue dans une macro-instruction. Dans l'exemple précédent, il est possible d'envisager l'exécution simultanée de l'ULE (P) et l'ULE (Q-R) dans la mesure où l'unité "Q-R" et son traitement d'exception est entièrement contenue dans une macro-instruction. Des mécanismes peuvent être mis en oeuvre afin que si une exception se produit dans l'ULE (Q-R), le branchement au traitement des exceptions soit effectué à l'intérieur de la macro-instruction, entrainant la suspension de l'unité entière. Dans ce cas, l'exception est traitée "localement" à la macro-instruction qui contient Q-R et n'a pas de conséquence sur l'activation des autres macro-instructions.

D'une manière générale de nombreux cas de parallélisme peuvent être exploités lorsque les ULE(s) peuvent être définies statiquement. La solution proposée peut être améliorée pour prendre en compte ces cas de parallélisme. Il faut cependant se garder de mettre en oeuvre des mécanismes trop complexes qui seront rarement exploités et qui, par contre, pénaliseront l'exploitation de cas de parallélisme plus simples et plus fréquents.

* <u>Solution 2</u>

La méthode décrite précédemment implique que lorsqu'une exception est levée, le système interrompt plusieurs processus simultanément, ce qui est techniquement difficile à réaliser. La solution décrite ci-dessous propose une méthode ne nécessitant pas l'interruption de processeurs. Cependant, elle nous semble moins efficace que la précédente dans la mesure où la prise en compte des mécanismes d'exceptions pénalisent les programmes où aucune exception n'est levée.

Cette méthode est également basée sur le principe de décomposition macroscopique qui garantit que, pour une tâche, seules les macro-instructions appartenant à la même unité logique exception sont actives simultanément. De plus la partie traitement des exceptions est dépendante de l'ensemble des instructions de l'unité et, de ce fait, la ou les macro-instruction(s) contenant ce traitement, terminent nécessairement le graphe de dépendance de l'unité. En particulier, la macro-instruction MIE qui contient la partie initiale des traitements des exceptions est terminale dans le graphe de l'unité.

Lorsqu'une exception est levée dans une macro-instruction, différente de MIE, si aucune exception n'a été levée auparavant dans la tâche, alors la partie traitement de contrôle de la macro-instruction permettant les enchainements avec la suite du programme est exécutée. De plus, la présence d'une exception est signalée aux macro-instructions descendantes, en positionnant à VRAI un indicateur de présence d'exception dans le mot d'état des macro-instructions descendantes.

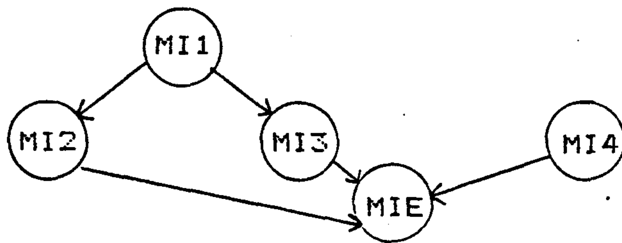
Si une exception est levée et qu'une exception à déjà été levée auparavant, alors il n'est pas tenu compte de la nouvelle exception et le traitement de contrôle de la macro-instruction permettant les enchainements avec la suite du programme est exécuté, ainsi que la mise à VRAI des indicateurs "présence exception" des macro-instructions dépendantes.

Lorsqu'une macro-instruction différente de MIE est activée, elle teste l'indicateur "présence exception" de son mot d'état. Si sa valeur est fausse, l'exécution normale de la macro-instruction est réalisée, sinon le traitement de contrôle est immédiatement réalisé ainsi que la mise à vrai des indicateurs "présence exception" des macro-instructions suivantes.

Lorsque la macro-instruction MIE est activée, si présence exception est vraie, alors l'exécution de la macro-instruction commence à l'adresse @TE. Lorsqu'une exception est levée dans MIE, un branchement à l'adresse @TE du traitement des exceptions est réalisé.

Lorsque l'exception doit être propagée à l'unité logique exception appelante le procédé est itéré sur les macro-instructions de cette unité.

Par exemple, soit un système de macro-instructions appartenant toutes à une même unité logique exception dont le graphe de dépendance est :



Une exception dans MI1 déclenche la décrémentation des compteurs de dépendance de MI2 et MI3 ainsi que la mise à vrai de leur indicateur de présence exception.

L'exécution de MI1 et MI2 déclenche immédiatement la décrémentation du compteur de dépendance de MIE et la mise à jour de son indicateur de présence exception. MI4 s'exécute normalement et son traitement de contrôle décrémente le compteur de dépendance de MIE. Lorsque MIE est activée, elle exécute la partie traitement d'exception située à l'adresse @TE.

La solution qui vient d'être présentée à deux inconvénients: lorsqu'une exception est levée sa prise en compte est longue et les traitements indépendants de la macro-instruction où l'exception a été levée continuent à s'exécuter.

En outre, les informations nécessaires à la gestion des processus par le système restent nécessaires.

3.3 CONCLUSION

Les chapitres II et III ont permis d'expliquer les principes de fonctionnement du modèle Data Flow Macroscopique, de rappeler les applications de celui-ci aux structures de programmation classiques (IF, CASE, LOOP, sous-programmes) et de développer comment il pouvait s'appliquer aux structures plus spécifiques telles que celles rencontrées dans le langage Ada (tâches, rendez-vous, exception).

Cette parallélisation des programmes est réalisée grâce à un logiciel de transformation de programme (tel que VESTA) et est transparente à l'utilisateur.

Le Data Flow Macroscopique est un moyen d'expression du parallélisme adaptable et performant :

- c'est un moyen d'expression du parallélisme pouvant être appliqué aux applications existantes et dont la puissance permet d'envisager de nombreuses évolutions

- il permet une expression du parallélisme adapter aux mécanismes de contrôle nécessaire à son exploitation. Ainsi un équilibre, entre la taille des traitements effectués en parallèle et des traitements de contrôle qu'ils nécessitent, peut être recherché. Cette notion appelée "granularité" du parallélisme vise à mettre en oeuvre un parallélisme efficace.

- il exprime un parallélisme qui peut être paramètrable et adapter aux ressources, en particulier au nombre de processeurs de la machine.

- il permet l'utilisation de machine construite à l'aide de processeurs du commerce, donc peu coûteuse. De plus, l'exécution d'une macro-instruction sur un processeur de type VON NEUMAN permet l'utilisation de techniques classiques d'accélération des programmes séquentiels telles que l'exécution "pipe-line" des instructions, l'utilisation de registres ou de mémoires caches.

CHAPITRE 4

Eléments d'une architecture multi-microprocesseurs

4.0 INTRODUCTION

Si la recherche de performances (c'est à dire le nombre d'instructions exécutées par seconde) est l'un des objectifs des réalisations multimicroprocesseurs, d'autres objectifs deviennent aussi importants voire indispensables, ce sont :

- la modularité et l'extensibilité de la machine qui apportent aux systèmes des qualités d'adaptation en termes de puissance de calcul et introduisent la notion de gamme de machine, dont les caractères de rentabilité économique ne sont pas négligeables pour un constructeur,

- la sûreté de fonctionnement, vaste notion, qui recouvre à la fois le matériel et le logiciel, retrouvée à travers :

  . la fiabilité du matériel ;

  . la tolérance aux pannes matérielles ou logicielles ;

  . la résolution de problèmes de protection ;

  . l'intégration d'un maximum de concepts et de techniques facilitant la mise au point des programmes.

- un bon rapport performance/prix qui prend autant en compte le coût de la machine que le coût de développement, de mise au point et d'évolution du logiciel.

Il apparaît alors, que outre la puissance de calcul obtenue par l'exploitation du parallélisme, les qualités des multimicro-processeurs sont leur souplesse d'utilisation et leur capacité d'adaptation logicielle et matérielle.

Ce sont de telles motivations qui ont guidé nos choix d'une part dans la conception d'un modèle d'expression du parallélisme "Le Data Flow Macroscopique" qui, comme, cela a été montré aux chapitres précédents, est très adaptable, et, d'autre part, dans la proposition d'éléments d'une architecture parallèle : AMOS.

AMOS/Architecture Multi-Opérateurs Séquentiels) est une machine MIMD, ce qui en fait une machine capable de répondre aux critères de :

- performance

- extension, fiabilité

- coût peu élevé.

Afin de ne pas disperser les efforts, de diminuer les prix de revient et de réduire les risques technologiques encourus par la réalisation d'une machine physique, AMOS est une machine conçue
à partir d'éléments existants. En particulier, l'utilisation de processeurs classiques de type Von Neuman du commerce permettra un coût de conception et de réalisation faible.

Cependant l'objectif quant au délai de réalisation n'étant plus actuellement défini, il n'est pas apparu intéressant d'examiner quelques microprocesseurs disponibles sur le marché, comme le motorola 68000 ou le HP9000, pour déterminer lequel serait le mieux adapté à notre environnement multi-processeurs.

Ce choix n'est pas urgent d'autant que, comme cela a été montré au paragraphe 1.2.2, les microprocesseurs évoluent rapidement devenant à la fois plus efficaces et plus souples d'utilisation.

Ainsi, examinant cette évolution des microprocesseurs et les études réalisées au centre de recherche sur un processeur appelé MLI (Machine Langage Intermédiaire) (cf §4.1) des convergences apparaissent dans les objectifs poursuivis. Plutôt que de choisir un processeur du commerce et d'étudier comment l'intégrer dans un environnement multi-processeurs, il est apparu plus intéressant de montrer comment les caractéristiques et les motivations de MLI sont adaptées aux objectifs de AMOS.

En outre, MLI est apparue un support intéressant pour montrer une mise en oeuvre possible du modèle Data Flow Macroscopique appliqué au langage Ada. Sa structure n'est pas complètement figée ce qui permet de prévoir des adaptations à l'environnement multi-processeurs dans lequel nous envisageons son utilisation.

Ce chapitre décrit donc rapidement les caractéristiques principales de MLI dont la connaissance est nécessaire pour comprendre la mise en oeuvre du Data Flow Macroscopique sur ce système qui est étudiée au chapitre suivant. Puis après avoir rappeler l'état l'art en matière de réseaux d'interconnexion, des éléments d'architecture du multiprocesseur AMOS sont présentés.

4.1 MLI, un Processeur de Base

Les études sur la définition d'un processeur langage intermédiaire adapté à tous les langages évolués à structure de blocs (LTR/V3, Chill, Pascal, Ada, ...) ont été effectuées au centre de recherche Bull dans le service transformation de programmes dirigé par Claude Renvoise avec la collaboration de Pierre Douspis, Marc Legoux, Agnès Bradier et Jacques Brygier.

L'ensemble des études relatives à ce projet nommé MLI (Machine Langage Intermédiaire) se trouve dans (RDO 81a), (RDO 81b), (RDL 82a), (RDL 82b), (Bra 83), (Bry 83).

Les recherches et développements tant dans le logiciel que dans l'architecture des machines tendent à créer des outils plus sûrs, plus fiables, et fournissant des meilleurs services en restant cependant les plus transparents possibles aux utilisateurs.

Généralement, les recherches menées dans ces deux domaines sont effectuées séparément, ce qui a pour conséquence de limiter la portée des résultats obtenus.

Pourtant, une recherche coordonnée d'un langage et d'une architecture, l'intégration dans le matériel de fonctionnalité habituellement prise en charge par le logiciel offre de nombreux avantages et apparaît être un bon moyen pour atteindre les objectifs énoncés ci-dessus.

Ainsi les architectures généralement appelées machines langages comme MLI apportent les améliorations suivantes :

- L'accroissement des performances. La richesse du jeu d'instructions grâce aux progrès de la microprogrammation et de leur exécution sur une machine adaptée permet une exécution performante des programmes. En outre, une représentation plus compacte du code réduit son encombrement mémoire et diminue son temps d'exécution.

- la sûreté de fonctionnement : la facilité d'utilisation des instructions machines d'un niveau sémantique relativement élevé facilite la communication homme/machine, améliore sensiblement la fiabilité de la machine et contribue à la sûreté de fonctionnement.

Cependant, il faut craindre que le choix d'un jeu d'instructions de trop haut niveau sémantique fige la machine qui devient alors dépendante d'un langage unique. C'est pourquoi il est nécessaire de trouver un compromis grâce à la richesse du jeu d'instructions et aux avantages apportés par la microprogrammation.

La description de MLI dans cette thèse ne peut être que très rapide et seules ses caractéristiques principales, sont rappelées.

4.1.1 La fonction mémoire

Le principe de base de la mémoire de MLI, motivé par la recherche d'un protection maximale, est celui d'une ségmentation à deux niveaux.

La segmentation d'une mémoire permet une meilleur protection mais engendre des problèmes de coût de création et gestion par le système d'exploitation.

Aussi, pour ne pas avoir à supporter le coût élevé dû à un partitionnement important de la mémoire en segments virtuels, un nouveau type de segment, appelé segment logique, correspondant à une zone continue d'espace mémoire contenue dans un segment virtuel, a été introduit.

Ainsi, la mémoire virtuelle de MLI a une capacité de 2**40 octets segmentables en :

- segments virtuels créés et détruits par le gestionnaire de mémoire virtuelle ;

- segments logiques créés et détruits par les instructions de la machine.

Chaque segment étant divisé en page de 512 octets.

Cette solution permet une configuration de la mémoire virtuelle en groupe de segment virtuels (SV), chaque groupe correspondant à un ensemble de SV de taille T2.

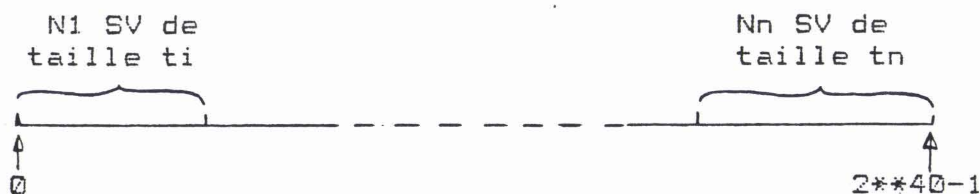


fig 1 : Représentation de la mémoire virtuelle

La gestion de ces groupes s'effectue par l'intermédiaire d'une table de groupe de segments (TGS) dont chaque élément contient :

- la taille d'un segment d'un groupe

- l'adresse de base d'implantation de ce groupe

- l'adresse du catalogue de segment de chaque groupe.

La table TGS permet au gestionnaire de mémoire virtuelle :

- de résoudre rapidement les défauts de pages virtuelles

- une gestion performante de la mémoire virtuelle.

L'implémentation de la mémoire de MLI offre de nombreux avantages, elle est :

- performante, ce qui s'avère important dans un contexte temps réel

- facile à reconfigurer

- apte à assurer un schéma d'adressage optimum dans une configuration sans mémoire virtuelle.

4.1.2 L'adressage lexical

Le mode d'adressage de MLI est basé sur l'imbrication des blocs qu'implique les langages structurés.

Le niveau lexical d'un bloc correspond à son niveau d'imbrication dans un programme. Un bloc a une visibilité potentielle sur tout ce qui précède hiérarchiquement sa déclaration.

Dans MLI, la méthode d'implémentation de l'accessibilité aux objets consiste à matérialiser une pile d'activation aux travers d'une table mémoire, appelée Tables des Niveaux Lexicaux (TNL), qui est incluse dans une structure plus riche, appelée Bloc de Contrôle de Processus (BCP), associée à chaque processus et contenant les informations les caractérisant.

Chaque entrée de la table TNL contient un descripteur de segment logique, appelé Segment Logique de Contexte Lexical (SLCL).

Un SLCL est associé à toute unité correspondant à un bloc défini par le langage (par soucis d'optimisation certains blocs peuvent être réunis dans une même unité dit "de programmation"). Il contient :

- des descripteurs de segments logiques regroupant les objets du bloc par affinité : SL de données locales, SL des constantes, ...

- des informations de contexte : adresse de retour, niveau lexical du bloc appelant, compteur de tâches non terminées, ...

L'accès à un opérande manipulé par le langage machine se fait grâce à une adresse lexicale, se présentant sous la forme d'un triplet (i,c,d) avec :

- i le niveau lexical correspondant à un descripteur de SLCL d'une TNL

- c représente une catégorie d'objet et désigne un descripteur de segment logique (SL) d'un SLCL

- d représente le déplacement dans le SL et désigne l'objet à accéder.

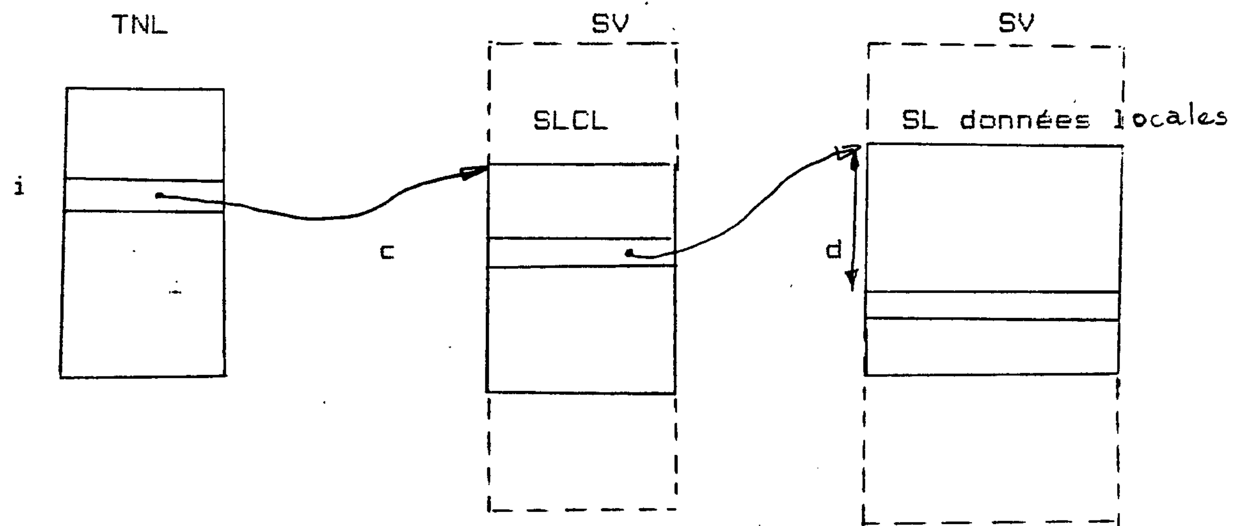


fig 2 : Principe d'adressage de type lexical adapté dans MLI

Une description détaillée de la gestion de la TNL est fournie dans (RD 81a).

Cet adressage de type lexical assure une bonne sûreté de fonctionnement grâce à :

- la prise en compte des règles de visibilité

- la réalisation de petits domaines de protection dûe à l'existence de segments logiques

- la simplicité de mise en oeuvre et d'utilisation surtout en ce qui concerne les appels et retours de sous-programmes.

Il faut cependant constater que ce système d'indirection engendre un adressage long. Aussi, un élément matériel d'architecture spécifique (le TAL cf §4.1.4) permet de remédier à cet inconvénient important.

4.1.3 Mécanismes de gestion des tâches

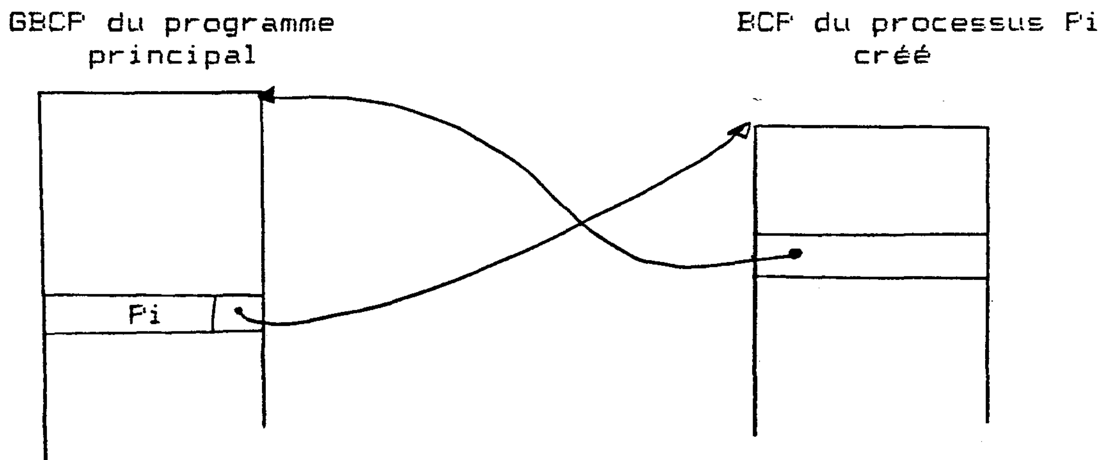
Dans cette partie les structures de données et les mécanismes utilisés dans MLI pour prendre en compte les processus ou tâches du langages Ada sont rapidement décrits. Ces mécanismes sont détaillés dans (RD 81b). Ces structures seraient utilisées au cours du chapitre V afin de montrer une mise en oeuvre possible du Data Flow Macroscopique.

4.1.3.1 Mécanismes d'activation d'une tâche

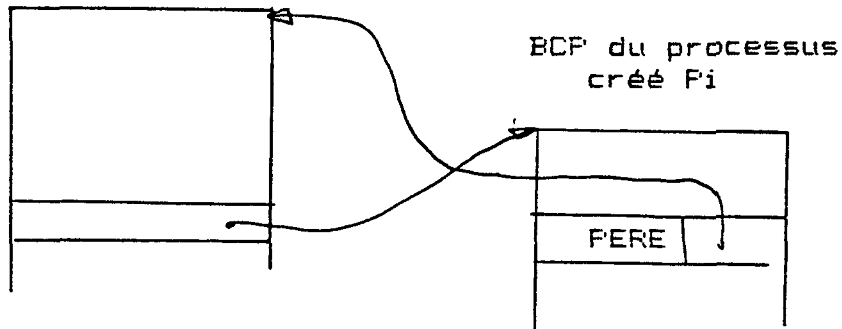
Chaque processus (ou tâche) est caractérisé par un ensemble de propriétés portant sur son état, ses ressources propres et ses liens avec d'autres processus. Elles sont regroupées dans une structure de données spécifiques appelée Bloc de Contrôle de Processus (BCP).

Chaque BCP est accessible à partir d'une tâche appelée GBCP, groupant les informations globales du programme exécutable.

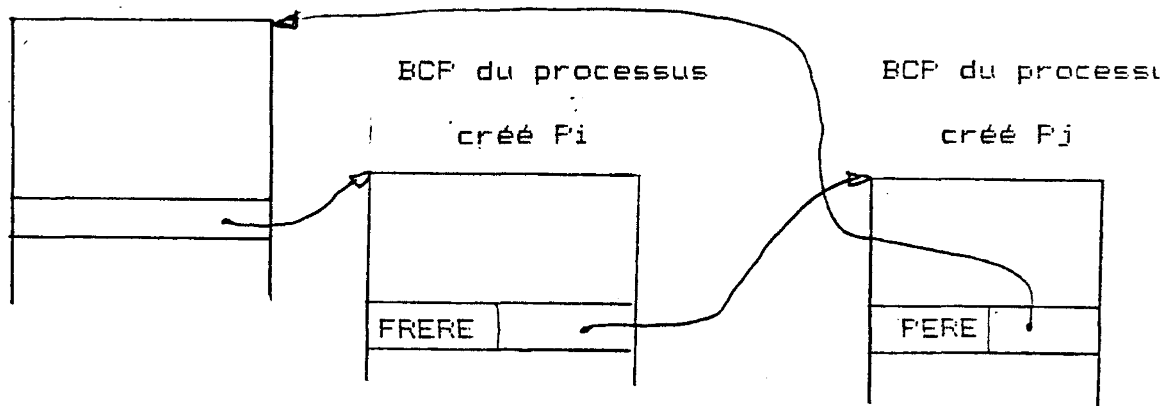
La création d'une tâche se caractérise donc par la création d'un BCP et de liaisons avec d'une part, le GBCP :



et d'autre part, le BCP du processus créateur :



ou éventuellement les liaisons avec les processus frères lorsqu'ils existent



Les différentes informations du BCP sont initialisées, en particulier la TNL du processus créateur est recopiée dans la TNL du processus créé jusqu'au niveau lexical-1. Le nouveau processus a donc la visibilité qu'à le processus créateur sur les objets du programme au moment de l'élaboration.

Enfin, une liaison entre le BCP du processus créé et le SLCL créateur, c'est à dire l'unité appelante, est établie.

Lorsqu'un processus crée d'autre(s) processus il se trouve bloqué sur un sémophore "attente activation" tant que tous les processus créés ne sont pas actifs.

En effet, les spécifications du langage Ada impose à une tâche maitre d'attendre que la (ou les) tâche(s) créée(s) ait élaboré sa partie déclarative, car, si une exception est levée durant cette élaboration, il faut propager l'exception au point de création, c'est à dire à la fin de la partie déclarative du processus créateur.

Aussi, lorsqu'une tâche passe à l'état actif, elle sort de la liste des processus créés par le processus créateur pour entrer dans la liste des processus dépendants du processus propriétaire. De plus une liaison entre le SLCL de l'unité de programme contenant le type tâche et le BCP de cette dernière est établie. Ce lien est utilisé pour retrouver l'unité propriétaire et mettre à jour son compteur de tâche non terminée. La valeur de ces compteurs permet de déterminer l'état de terminaison ou non d'une tâche.

4.1.3.2 Mécanismes de rendez-vous

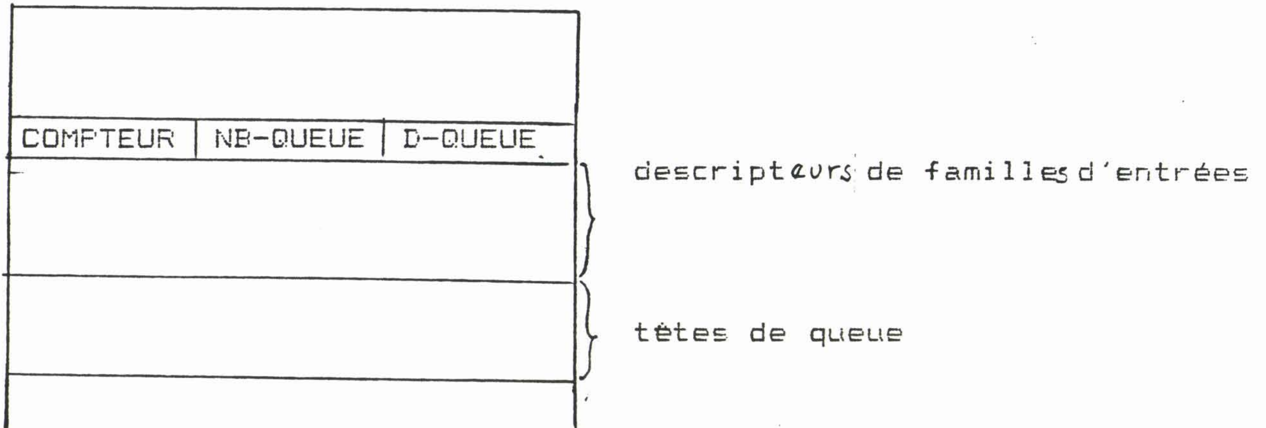
Dans Ada, les tâches peuvent se synchroniser et communiquer au moyen de rendez-vous. Les rendez-vous sont des structures de contrôle propres à une tâche et les informations correspondantes sont stockées dans son bloc de contrôle de processus (BCP).

Ainsi chaque tâche possède un sémaphore de rendez-vous implanter dans son BCP.

Ce sémaphore contrôle une zone de mémoire dans laquelle sont mémorisées toutes les informations nécessaires aux différents rendez-vous possibles sur les différentes entrées ou familles d'entrées de la tâche.

En particulier, les appels à chaque entrée sont mémorisés sous forme de liste dont l'accès se fait par l'intermédiaire d'une tête de liste contenant l'état du rendez-vous (accept ou appel arrivé ou non). Chaque maillon de chaque liste contient les informations caractéristiques du processus qui veut communiquer et une zone où les paramètres éventuellement transmis pour la communication sont mémorisés.

La structure générale (c'est à dire non détaillée) du sémaphore de rendez-vous est la suivante :

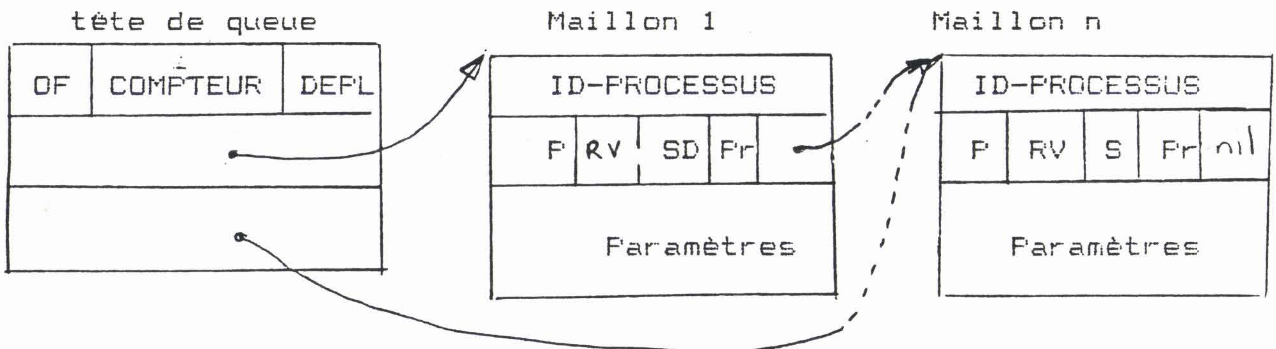


COMPTEUR : c'est le compteur global du sémaphore de rendez-vous ;

NB-QUEUE : indique le nombre total de têtes de queue contenues dans le sémaphore ;

D-QUEUE : indique le déplacement octet de la première tête de queue dans le sémaphore.

Les files d'attente ont la structure générale suivante :



OF : indique si la tête de queue est ouverte ou fermée

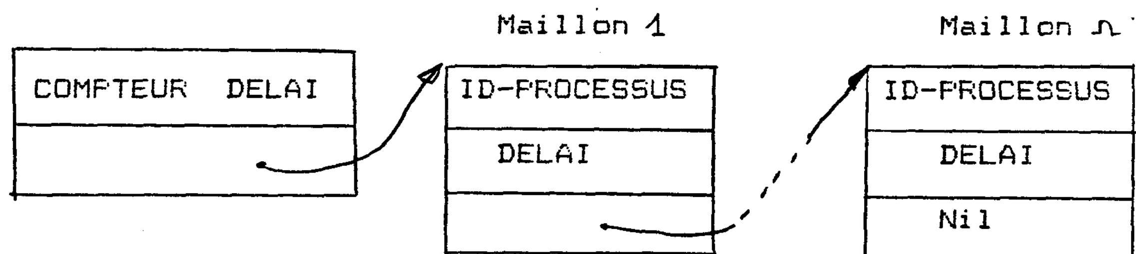
COMPTEUR : c'est la ressource sur laquelle tout processus qui veut communiquer effectue une P opération pour ce mettre dans cette queue

ID-PROCESSUS : contient la capacité du BCP du processus qui veut communiquer

P : indique la présence ou non de paramètres
RV : indique si le rendez-vous est en cours
SD : indique si le processus décrit dans ce maillon est également dans la queue du sémaphore délai
Pr : c'est la priorité du processus qui veut communiquer

L'exécution d'instruction "accept" ou appel d'une entrée correspond à la manipulation des informations contenues dans les queues. Ainsi, par exemple, l'exécution d'un "accept" sur une entrée entraine l'ouverture de la tête de queue si un rendez-vous n'est pas immédiatement possible (queue vide) sinon la réalisation du rendez-vous. Réciproquement, l'exécution d'un appel entraine une mise en attente dans un maillon de la liste lorsque le rendez-vous n'est pas immédiatement possible, c'est à dire que la tête de queue est fermée ou que la liste des processus en attente n'est pas vide.

Les rendez-vous avec délai sont réalisés grâce à un sémaphore délai appartenant au système. Ce sémaphore regroupe dans une liste (ordonnée par délai croissant) tous les processus suspendus sur un délai, il a la structure suivante :



COMPTEUR DELAI : contient l'écart de temps au bout duquel le premier processus doit être réveillé

DELAI : c'est un délai relatif au temps où se produira le réveil du processus qui précède dans la liste

Lorsque le délai imputé au premier processus de la liste devient nul celui-ci est réveillé.

Une opération d'attente sélective sur une ou plusieurs alternatives (instruction "select") est réalisée grâce à la construction d'une table de sélection dont chaque entrée décrit la sémantique d'une alternative relative à la communication entre processus.

Chaque entrée de la table a le format suivant :

F	OF	CONDITION

F : indique la nature de l'alternative/accept, délai, else ou terminate

OF : indique si l'alternative associée à l'entrée de la table est ouverte ou non

CONDITION : désigne la tête de queue du sémaphore rendez-vous du processus si F = A, une durée du délai si F = D, et est non utilisée sinon.

Les informations contenues dans la table de sélection permettent de choisir une alternative lors de la réalisation du rendez-vous.

4.1.3.3 Mécanismes de traitements d'exceptions

Ces mécanismes ont été conçus pour être implantés au moindre coût temps/espace lorsque l'utilisateur a prévu un traitement d'exception, et à un coup, si possible nul, lorsqu'aucun traitement d'exception n'est fourni.

Le BCP de chaque tâche contient trois champs :

- PRESENCE EXCEPTION : indique si une exception est à traiter quand le processus est réveillé ;
- VALEUR EXCEPTION : indique la valeur de l'exception à traiter
- ADRESSE TRAITEMENT-EXCEPTION : ce champ est utilisé lorsqu'une exception se produit dans le camps de la tâche, il est rempli lorsque le processus devient "activé".

Lorsqu'une exception est levée la valeur de l'exception est rangée dans le champ VALEUR EXCEPTION. La valeur de l'exception permet de sélectionner une entrée dans une table de branchement. Lorsque l'exception peut être traitée cette table fournit un déplacement afin d'accéder au traitement d'exception sélectionné, sinon il y a propagation.

La propagation des exceptions nécessite un descripteur DTE associé à chaque unité. Il est placé devant le code d'élaboration de chacune d'elle.

DTE contient deux champs :

- DEBUT : contient la position de la première instruction générée pour la partie "BEGIN" de l'unité ;

- TRAITEMENT EXCEPTION : contient la position du traitement des exceptions s'il existe, o sinon.

DEBUT permet de savoir si l'instruction, où a été levée l'exception, se trouve dans la partie élaboration (il y a alors propagation) ou dans la partie traitement de l'unité (il y a alors propagation si il n'y a pas de traitement correspondant à l'exception).

Le DTE d'une unité est retrouvée à partir de l'instruction d'appel correspondant. Cette instruction est retrouvée à partir de l'adresse retour (située dans la SLCL) et de la taille de l'instruction d'appel déduite du code opération. L'instruction d'appel permet de connaitre l'adresse de la première instruction du sous programme et donc d'accéder aux informations contenues dans DTE.

Lorsqu'une exception est propagée dans le corps d'une tâche alors l'adresse du traitement des exceptions est recherchée dans le champ ADRESSE TRAITEMENT EXCEPTION du BCP, si elle ne peut être traitée, la propagation est arrêtté sauf si le processus est "en rendez-vous", auquel cas l'exception est propagée chez l'appelant.

Si le processus est dans l'état "élaboré" ou "activable", l'exception est propagée au processus créateur.

4.1.4 Architecture matérielle de MLI

Cette section a pour but de donner une vision globale de architecture du processeur langage intermédiaire.

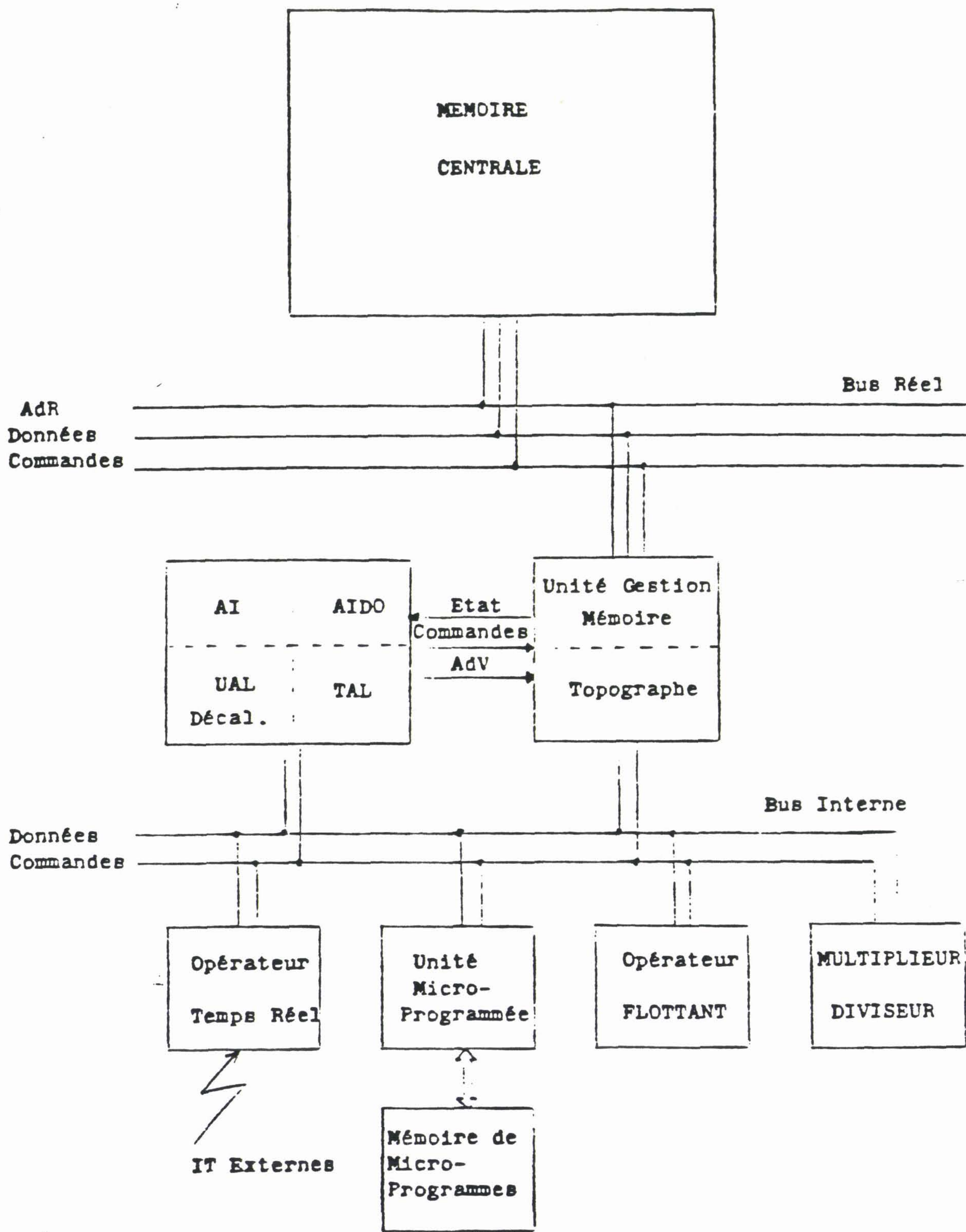
MLI se compose principalement de deux parties :

- la mémoire centrale MC

- le processeur centrale PC

Ces deux éléments sont en communication grâce à un bus principal appelé Bus Réel en raison des adresses qu'il véhicule.

Le PC, quant à lui, est composé de plusieurs modules interconnectés soit directement pour certains d'entre eux, soit par l'intermédiaire d'un autre bus appelé Bus Interne.



AdR : Adresses Réelles
AdV : Adresses Virtuelles

fig 3 : Architecture matérielle de MLI

Le processeur langage est l'élément principal du PC, c'est lui qui prend en charge l'excécution des instructions qu'il réalise lui même ou qu'il fait réaliser par les autres composantes du PC.

Il est constitué de trois composantes :

- l'approvisionneur d'instructions (AI)

- l'analyseur d'instructions/décodeur (AIDO) incluant une U.A.L.

- le transformateur d'adresses lexicales (TAL)

. AIDO est l'élément central du processeur langage. Il obtient de AI les instructions que ce dernier extrait d'une manière indépendante et si possible avec anticipation de la mémoire, utilise le TAL pour générer les adresses virtuelles des opérandes, des instructions et les transmettre au topographe qui les transforme en adresses réelles que l'UGM utilise pour effectuer les accès mémoire.

Ces trois composants ont un fonctionnement asynchrone autorisant ainsi un fonctionnement pipe-line qui diminue le temps d'excécution des instructions. Plus précisément, AI dispose d'une logique permettant d'extraire par anticipation les instructions de la mémoire, AIDO analyse le code opération de l'instruction puis décode le premier opérande, celui-ci demande au TAL de convertir son adresse lexicale en adresse virtuelle puis d'émettre la requête mémoire nécessaire pendant qu'il décode le second opérande, ...

. Le T.A.L. est un élément original du processeur langage il transforme une adresse lexicale (T,i,c,d) en adresse virtuelle permettant d'accélérer au moyen d'une mémoire associative, le temps d'accès aux informations qui serait sinon prohibitif.

Les ressources du processeur langage sont de type registre et peuvent être réparties en deux catégories :

- les registres généraux non programmables utilisés pour la réalisation de chaque instruction et non la communication entre instruction.

- les registres de services parmi lesquels ceux marquant l'état du programme et ceux utilisés pour les adresses.

. Le processeur langage est l'élément orchestre du processeur central en ce sens qu'il commande l'exécution des instructions, ainsi certaines d'entre elles peuvent être exécutées par des fonctionnalités connectées sur le "bus interne". Ces unités opérant comme des esclaves sont :

- le topographe et l'UGM qui assurent l'interface entre le processeur langage et son environnement

- l'unité microprogrammée qui complète le processeur langage dans le support du langage

- l'opérateur temps réel qui contribue à la fois à l'interface avec l'environnement et au support du langage

- les opérateurs flottants et MUL/DIV qui servent à améliorer les performances à l'exécution.

. L'unité microprogrammée déroule des micro-programmes nécessaires à l'exécution des instructions complexes telles que les instructions liées aux tâches au sens Ada (création, élaboration, ..., terminaison) ainsi que d'autres fonctionnalités systèmes. Cette unité est l'élément qui donne au processeur sa puissance pour résoudre des problèmes complexes (rendez-vous), sa capacité d'adaptation et sa souplesse d'utilisation.

. L'opérateur temps réel a pour fonction de gérer l'horloge et les délais qui interviennent dans l'exécution des applications à la charge du processeur central. En particulier, il gère les délais d'attente des processus en rendez-vous.

. Les opérateurs arithmétiques assurent des performances satisfaisantes à la machine. L'opérateur Multiplication/Division doit réaliser la multiplication et la division de nombre entiers sur 8,16 et 32 bits. L'opérateur Flottant doit réaliser des opérations arithmétiques sur des nombres flottants de 32 bits et 64 bits.

L'architecture de MLI se veut extensible et autorise la multiplication :

- d'une part, des modules mémoires afin que la mémoire centrale puisse être organisée en plusieurs bancs ;

- d'autre part, des processeurs centraux qui réaliseraient ainsi une machine multiprocesseurs configurable suivant le type d'applications demandées.

MLI apparaît donc comme un bon support pour l'étude d'une architecture adapté au Data Flow Macroscopique appliquée au langage Ada.

4.2 LES RESEAUX D'INTERCONNEXION

Une bonne communication des informations entre les différentes unités d'un multi-processeur et l'une des clefs de la réussite de telles architectures.

En effet un réseau de communication ou interconnexion mal adapté aux flux des informations en transit devient très vite un "goulot d'étranglement" diminuant ainsi les performances du système. Par ailleurs les propriétés d'une machine découlent généralement de celles de son réseaux d'interconnexion. C'est pourquoi une étude bibliographique a été faite par D. Rocacher donnant lieu à un rapport (Roc 83) afin d'appréhender l'état de l'art en matière de réseaux d'interconnexion pour des machines fortement couplées. Seules quelques architectures et principes de fonctionnement seront rappelés dans cette thèse afin de dégager quelques idées principales qui conduiront à une proposition d'architecture de réseau pour AMOS.

De nombreuses structures de réseaux ont été proposées avec leurs qualités et leurs défauts inhérents. Pour construire un multiprocesseur il faut donc choisir un réseau suivant des critères de performance, coût, modularité, tolérance aux pannes, fiabilité, spécialisation, ..., qui satisfassent au mieux les objectifs fixés. Les réseaux sont étudiés ici, dans le but d'une utilisation dans une machine MIMD.

4.2.1 Quelques architectures

Généralement deux classes de réseaux sont distinguées (Fen 81) :

- les réseaux à topologie statique qui se caractérisent par des liaisons fixes entre les unités connectées

- les réseaux à topologie dynamique disposant de liaisons variables entre les unités connectées.

Les réseaux à topologie dynamique sont mieux adaptés à la construction de multiprocesseurs parce qu'ils sont plus performants, moins bloquants, plus extensibles, plus modulaires. Mais ils sont plus coûteux et souvent moins fiables que les réseaux statiques.

C'est pourquoi, ils seront étudiés ici car leurs caractéristiques sont mieux adaptées au type de machine visée. Cependant il ne faut pas négliger l'intérêt des réseaux statiques. Ainsi, par exemple, les bus sont des outils parfaitement maitrisés, simple, dont l'utilisation est envisageable lorsque le nombre (et/ou le débit) des unités à connecter est faible. Ainsi, ils peuvent être utilisés pour former des grappes de processeurs ou de bancs mémoire, elles-

mêmes reliées par un réseau d'interconnexion, ce réseau pouvant être également un bus : CM* (Swa 78) est l'exemple le plus connu de machine fonctionnant sur ce principe.

4.2.1 Réseaux Crossbars

Les réseaux crossbars sont des réseaux permettant de relier n-entrées à n-sorties. Ils sont représentés généralement sous forme matricielle (fig 4).

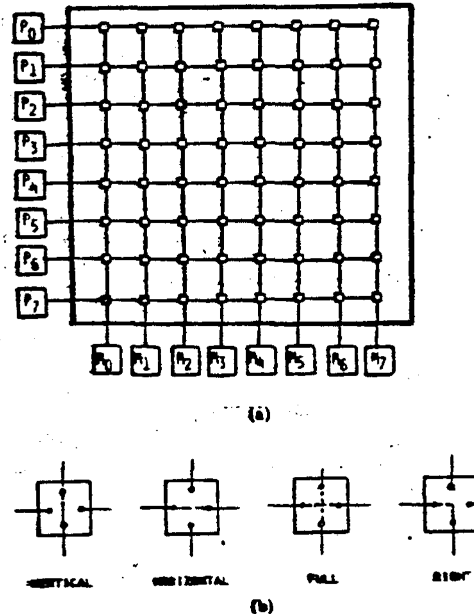


fig 4 : (a) crossbar system (b) crossbar switch positions

Leur degré de complexité, c'est à dire leur nombre de points d'interconnexions varie en N2 (pour un crossbar NxN), ce qui rend difficile la construction de réseaux de taille importante. Des réseaux 4x4, 8x8, 16x16 sont couramment utilisés (Sni 80) (Fr 81). Le multi-mini-ordinateur CMMP est construit autour d'un crossbar 16x16, (WB 72).

Les réseaux crossbars se caractérisent par un temps de traversée rapide (O/N). Ils sont non bloquants.

4.2.1.2 Réseaux multi-étages

Les réseaux multi-étages sont construits pour diminuer le nombre de connecteurs et donc de points de connexions par rapport aux crossbars. Leur complexité moins grande les rend moins cher et permet la réalisation de réseaux de plus grandes tailles.
Ces réseaux sont généralement construits à l'aide de connecteurs 2x2 permettant des connexions croisées ou parallèles.

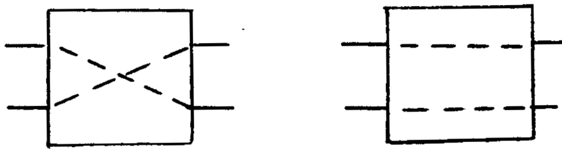


fig 5 : Les états d'un connecteurs 2x2

Certains auteurs proposent des connecteurs 4x4 comme cellules élémentaires. Le principal avantage de telle réalisation est la diminution du nombre d'étages du réseau puisque celui-ci est une fonction en lognN ou n est la taille des cellules élémentaires et N la taille du réseau.

Le type de liaison à la base des systèmes de connexion reliant chaque étage est appelé mélange parfait (perfect shuffle), il permet de mélanger deux ensembles de même dimension ainsi que le décrit la figure 6.

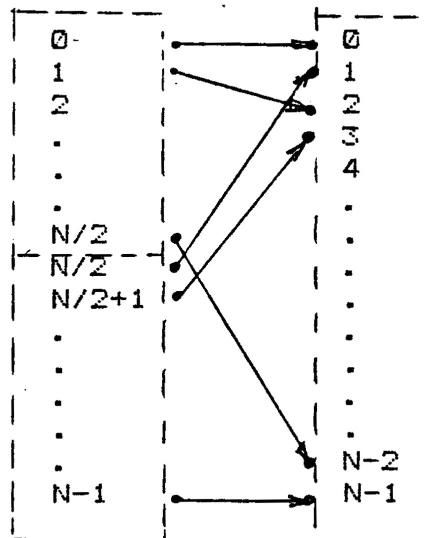


fig 6 : Perfect shuffle

- Réseaux de Bénés

Les réseaux de Bénés (Ben 68) permettent de relier 2n entrées à 2n sorties. Ils sont obtenus grâce à une construction récursive (fig 7)

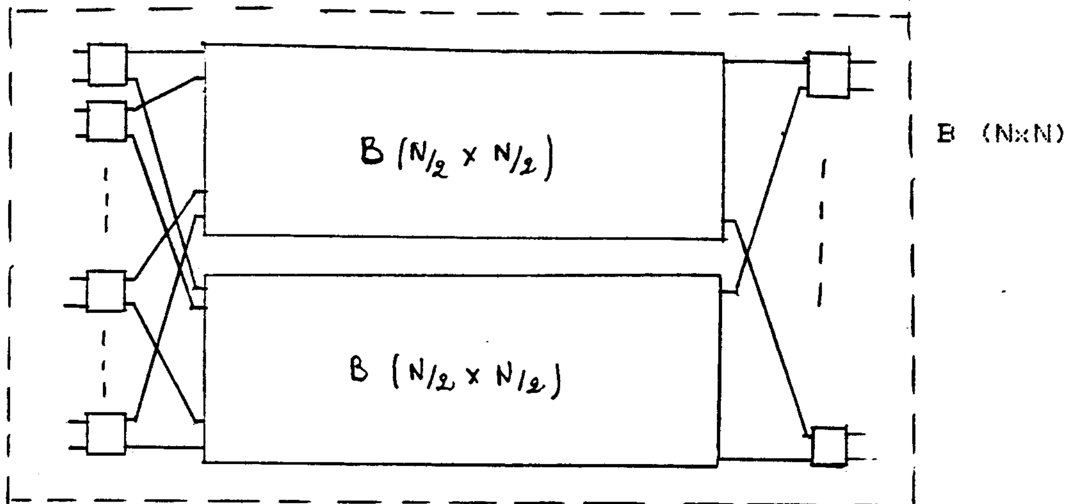


fig 7 : procédé récursif de construction d'un réseau de Bénés : La décomposition de B(NxN) fait apparaître deux blocs $B(\frac{N}{2}X\frac{N}{2})$, le procédé de décomposition est itéré sur ces blocs jusqu'à l'obtention de cellules élémentaires 2x2

Les réseaux de Bénés comportent (2 log2 N)-1 étage. Ils sont réarrangeables (c'est à dire qu'ils peuvent réaliser toutes les permutations possibles entre N entrée (N!).

- Réseaux à Log2 N étages

Ce type de réseaux est décrit dans un article de Chuan-Lin et Tse-Yung Feng (WF 80) où il est démontré que la plupart des réseaux à Log2 N étages sont topologiquement équivalents au réseau "Baseline" pris comme réseau de référence.

Le réseau Baseline NxN (Nentrées, Nsorties) est construit de façon récursive (fig 8) :

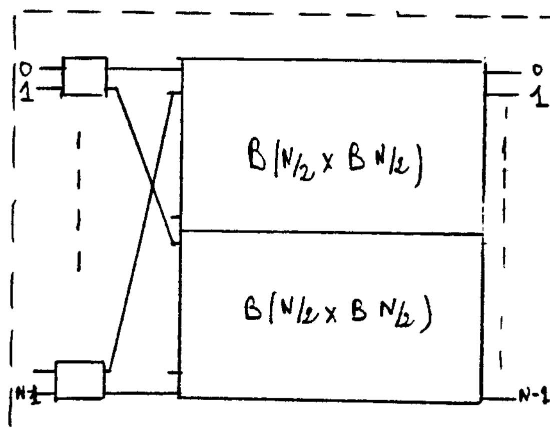


fig 8 : Procédé récursif de construction d'un réseau Baseline La décomposition de B(NxN) fait apparaitre deux blocs $B(\frac{N}{2}x\frac{N}{2})$, le procédé de décomposition est itéré sur ces blocs jusqu'à l'obtention de cellules élémentaires 2x2

Les réseaux Baseline comportent Log2 N étages, ils ne sont pas réarrangeables mais ils sont partitionnables, c'est à dire qu'ils peuvent se décomposer en sous-réseaux indépendants.

Les réseaux "Oméga" (Gov82), "flip" (Bat 76), "n-cribe" (Fea 77), "regular su Banyan" (GL 73), "modified data manipulator" (Fen 74), ..., font partie de cette classe de réseaux.

- <u>Réseaux à chemins redondants</u>

Ces réseaux sont mis en oeuvre pour accroitre la fiabilité et diminuer les risques de blocages en permettant plusieurs chemins possibles entre une entrée et une sortie. Les réseaux "gamma" (PR 82), ADM et IADM (MS 82) font partie de ce type de réseaux.

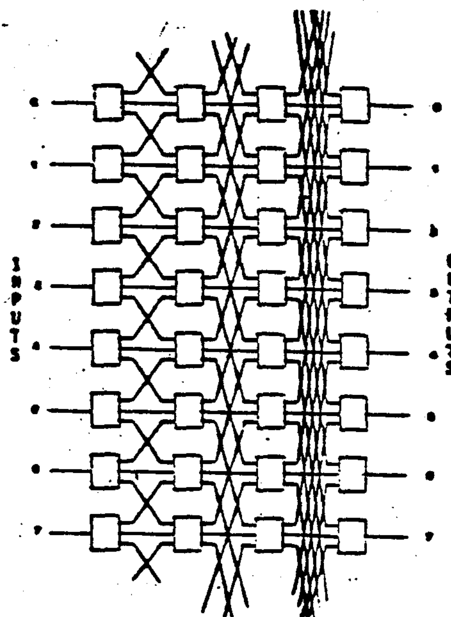


fig 9 : Le réseau gamma

4.2.1.3 <u>Autres réseaux</u>

Outre les types d'architectures classiques de réseaux décrits précédemment il semble intéressant de signaler trois architectures de conception originales :

- <u>Le réseau de la machine CHIP</u> :

CHIP (Configurable Highly Parallel) (Sny 82) est un multiprocesseur basé sur un réseau programmable capable de prendre différentes configurations (systolique, arbre, anneau,...). Les réseau de CHIP (fig 10) est une structure régulière formée de processeurs connectés les uns aux autres. Ces connexions ne sont pas statiques et passent par des connecteurs programmables permettant de configurer la machine.

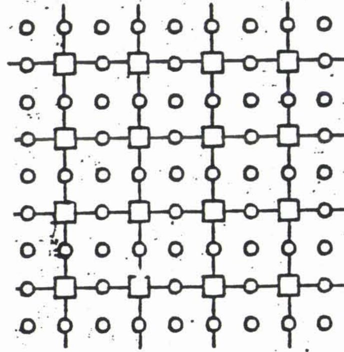


fig 10 : Principe de liaison inter processeurs dans CHIP

- Le réseau de la machine HEP

Le réseau d'interconnexion de la machine MIMD HEP est particulièrement intéressant parce qu'il est composé de connecteurs ayant chacun 3 ports d'entrée-sortie. Ceci donne à la machine de nombreuses possibilités d'extension et de configurations dans la mesure où pour ajouter une unité dans une configuration, il "suffit" de "couper" une liaison et d'y connecter un noeud connecteur sur deux ports, le troisième étant relié à la nouvelle unité.

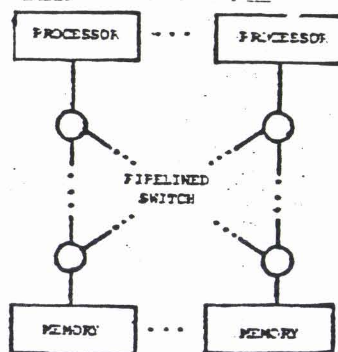


fig 11 : le réseau d'interconnexion de HEP

- Les mémoires circulantes

Une mémoire circulante est un ensemble de cellules, ou registres à décalage, connectées les unes aux autres, la sortie de la dernière étant rebouclée sur l'entrée de la première, les informations circulant de cellules en cellules.

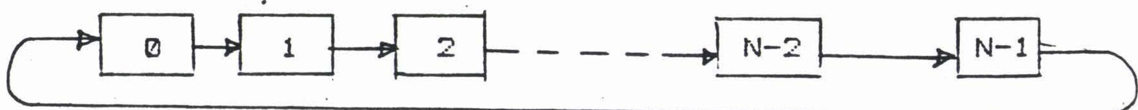


fig 12 : mémoire circulante

Les machines MAUD (Lec 79) et ZMOB (Rie 80) disposent de ce moyen de communication entre processeurs.

Les mémoires circulantes sont un moyen de communication très parallèle et sans conflit, cependant leur lenteur fait que leurs avantages seraient mieux exploités pour le transport d'un nombre d'information réduit. Ainsi par exemple, Dennis et Denning (DD 81) propose l'utilisation des mémoires circulantes comme moyen de distribution des travaux prêts à être exécutés, permettant ainsi de résoudre de nombreux problèmes d'exclusion mutuelle.

4.2.2 Principes de commandes

Dans ce paragraphe le principe de fonctionnement des réseaux à contrôle centralisé, c'est à dire où l'ensemble des connecteurs est commandé de façon synchrone par une unité de contrôle, n'est pas abordé car cette technique est adaptée à un fonctionnement SIMD.

Les réseaux à contrôle distribué ont une unité de contrôle associée à chaque connecteur. Chacun d'entre-eux réalise ses connexions en fonction de bits de routage établis à l'aide d'adresses source et destination.

Trois méthodes de commutation sont couramment utilisées :

- les commutations en ligne ("circuit switching") :

Dans ce type de commutation une ligne physique est créée entre la source et la destination. La liaison se fait de proche en proche : successivement, à chaque étage, le connecteur se trouvant en sortie de l'étage précédent détermine son état en fonction des bits de routage du message qu'il reçoit. Le chemin entre une entrée et une sortie, ainsi réalisé, permet la circulation des informations.

- les commutations par paquets ("packet switching") :

Cette méthode n'entraine pas la création de ligne physique entre une entrée et une sortie. Les messages traversant le réseau contiennent d'une part les bits de routage, d'autre part les informations à transmettre et forment un "paquet" trouvant son chemin de connecteur en connecteur.

- les commutations mixtes ("integrated switching") : elles cumulent les capacités des deux méthodes précédentes. Le réseau de la machine TRAC (JB 82) fonctionne de cette façon.

Les techniques de routage se font généralement sur l'interprétation des adresses source S et destination D des messages. Ainsi, par exemple, soit Sn-1 Sn-2 ... So et Dn-1 Dn-2 ... Do les représentations binaires de S et D. Dans un réseau Baseline, le connecteur suivant la source S prend l'état haut (⊟ ou ⊠) si Dn-1=1, ou bas (⊠ ou ⊟), si Dn-1 = 0. L'état du connecteur suivant est déterminé par le digit Dn-2. Le procédé est itéré jusqu'à la destination. Inversement la connexion de D à S se fait en utilisant les digit Si (WF 80).

Si une commande de type commutation de ligne est couramment utilisée, et plus simple à mettre en oeuvre, il semble que la commutation par paquet permette d'obtenir de meilleurs résultats. En effet, la commutation de ligne, en maintenant une ligne physique entre chaque source et destination durant le temps des échanges, engendre des blocages sur des chemins entiers puisque les réseaux à log2N étages sont bloquants.

Par contre la commutation par paquet permet l'utilisation de files d'attente (ou tampons) devant chaque entrée de connecteur. Ainsi lorsqu'un conflit intervient au niveau d'un connecteur, son unité de contrôle résoud le problème grâce à un système de logique prioritaire permettant le passage du message de plus haute priorité et la mise en attente du second dans le tampon de l'une de ses entrées. De plus cette technique permet une traversée "pipeline" du réseau, c'est à dire que des informations sont propagées simultanément au niveau de chaque étage : il y a recouvrement des traversées de chaque étage (fig 13).

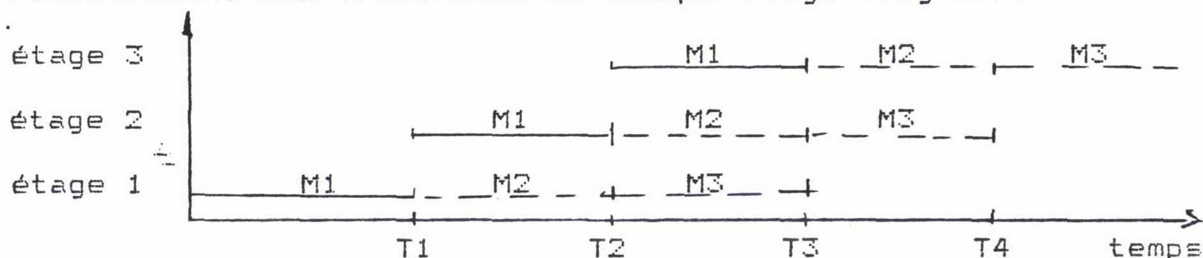


fig 13 : traversée "pipeline" d'un réseau baseline 8x8 disposant de file d'attente sur les entrées de ses connecteurs

En supposant que la traversée du réseau s'effectue sans conflit lorsque l'ensemble des messages Mi traversent l'étage 3, alors simultanément, Mi-1 traverse l'étage 2 et Mi-2 traverse l'étage 1. Le temps de traversée apparant du réseau est égal au temps de traversée d'un étage (s'il n'y a de blocage et si il n'y a pas de rupture du pipeline).

Cependant, la réalisation par paquet pose quelques problèmes pour l'exécution de certaines commandes. En particulier les opérations en exclusion mutuelle nécessitent la mise en place d'interfaces spécifiques au niveau des ressources. Celles-ci ne sont pas nécessaires pour la commutation de ligne, car la

création d'une ligne physique entre la sourcee et la destination exclue tout autre accès et permet de réaliser matériellement l'exclusion mutuelle.

4.2.3 Propriétés

Une partie importante des qualités d'une machine parallèle découle des propriétés de son organe principal : le réseau d'interconnexion. Quelques unes de ces caractéristiques sont maintenant décrites.

4.2.3.1 Modularité et extensibilité

Les réseaux multi étages ont de part leur construction de bonnes propriétés de modularité et d'extensibilité.

En effet, le caractère récursif de leur architecture permet d'obtenir, à partir de réseaux de petite taille, des réseaux de taille supérieure. Ainsi, comme cela a été montré au paragraphe 4.2.1.2., un réseau baseline 2n+1 x 2n+1 est construit à l'aide de deux réseaux 2n x 2n, à n étages, plus un étage de 2n-1 connecteurs.

Cependant, l'augmentation de taille de tel réseau, construit à l'aide de connecteurs xXx ne peut se faire que par puissance de x : ils sont localement extensibles.

Le réseau d'interconnexion de la machine HEP à la particularité d'être parfaitement extensible, c'est à dire que la taille du réseau peut être augmentée par pas de un connecteur.

4.2.3.2 Partitionnement

Le partitionnement est la propriété d'un réseau à se décomposer en sous réseaux indépendants ayant les mêmes propriétés que le réseau initial. Ainsi, les réseaux de type baseline, construit à l'aide de connecteur 2x2, sont partitionnables en 2, 4, 8, ... sous réseaux suivant.

Certains auteurs comme Siegel (Si 80) pensent qu'une machine disposant d'un tel réseau doit être reconfigurable dynamiquement et peut fonctionner en mode MIMD/MSIMD (cf §1.2.1.). Il faut remarquer que le problème n'est pas simple et que, outre la configuration de la machine, de nombreux problèmes d'adressage et de configuration de la mémoire se posent. Par contre, cette propriété peut permettre, en cas de panne d'une partie des unités de la machine ou du réseau, de fonctionner en mode dégradé sur une sous machine.

4.2.3.3 Tolérance aux pannes

Les réseaux d'interconnexions multi-étages comportent un grand nombre de points de connexion. La probabilité de panne est d'autant plus importante que ce nombre est élevé. Aussi le problème des réseaux de grande taille est leur fiabilité. La complexité des réseaux s'exprime en fonction du nombre de points de connexion, ainsi les crossbars ont une complexité de l'ordre de N2 (O(N2)), les réseaux de Bénés de O(Nlog2N), les réseaux de type baseline de O(N/2log2N). Bien entendu le coût d'un réseau est fonction de sa complexité.

Il est donc important d'envisager des fonctionnements en mode dégradé. De tels réseaux sont dit tolérants aux pannes.

Plusieurs méthodes de fonctionnement en mode dégradé sont possibles. Par exemple, la largeur des chemins de données peut être plus grande que celle nécessaire afin de disposer d'un certain nombre de bits de "secours", un étage supplémentaire permet d'augmenter le nombre de chemins possibles, les messages peuvent être sérialisés lorsque la largeur du chemin de données a été réduite du fait d'une panne, certains réseaux, comme celui de la machine TRAC (TL 79), utilisent des systèmes de routage adaptatif.

4.2.3.4 Performances

Si les réseaux crossbars sont les réseaux les plus complexes, et les plus chers, ils sont, par contre, les plus efficaces et sont souvent pris comme référence. Les autres réseaux sont généralement comparés à eux.

De nombreux critères permettent de juger de l'efficacité d'un réseau : temps de traversée, probabilité qu'une requête soit acceptée, temps moyen d'attente avant qu'une requête mémoire soit acceptée, nombre de requêtes mémoire acceptées par cycle, le rapport performance sur coût...

Une étude (MM 81) des réseaux crossbars montre que lorsque chaque processeur émet une requête à un banc mémoire, de façon uniforme sur l'ensemble de la mémoire, environ 70% des requêtes sont acceptées.

Des études réalisées par Patel et Dias (Pa 81)(DJ 81a) (DJ 81b), basées sur de nombreuses simulations d'un réseau de type baseline, indiquent que le nombre de requêtes acceptées par cycle d'un réseau construit à l'aide de connecteurs 4x4 est sensiblement meilleur que lorsque les connecteurs sont 2x2. Il ressort de ces études que les performances d'un réseau multi étages sont nettement améliorées par l'adjonction de tampons devant les entrées de chaque connecteur, ainsi les performances du réseau à log2N étages et d'un crossbar de même taille deviennent comparables. En outre, l'analyse et la simulation ont montré que

le meilleur rendement d'un tel réseau est obtenu avec des tampons pouvant contenir deux à cinq messages.

4.2.4 Conclusion

Dans une architecture parallèle où plusieurs dizaines de microprocesseurs coopèrent ou travaillent indépendamment le système de communication entre les différentes unités est un élément particulièrement important car une grande part des performances et des propriétés de la machine en découle.

Ce réseau d'interconnexion doit pouvoir faire communiquer N unités d'exécution à N unités mémoire, en offrant de bonnes performances, c'est à dire un temps de traversée faible et un nombre de requêtes acceptées par temps de cycle grand, il doit être extensible, tolérant aux pannes et d'un coût raisonnable.

De l'étude bibliographique réalisée il ressort que l'architecture modulaire des réseaux multi étages de type baseline correspond le mieux aux objectifs fixés.

Ainsi l'utilisation d'un réseau baseline construit à l'aide de connecteurs 4x4, afin de diminuer le nombre d'étages (log4N), et donc le nombre de modules élémentaires, est un moyen de communication extensible et d'une complexité raisonnable.

Un contrôle distribué et un type de communication par paquet asso
é à un réseau comportent des
files d'attente, permet de résoudre, au niveau de chaque connecteur, les problèmes de conflits, d'implanter des techniques de t
érance aux
pannes et d'obtenir des performances qui deviennent comparables à celles de crossbars.

4.3. ELEMENTS D'UNE ARCHITECTURE MULTIPROCESSEUR

Parallèlement aux travaux sur le Data Flow Macroscopique nous avons mené quelques réflexions sur la conception d'une architecture parallèle, adaptée à l'exécution de macro instruction, appelée AMOS/Architecture Multi Opérateurs Séquentiels). Suite à une étude approfondie des machines parallèles existantes (DR 83), quelques éléments d'architecture ont été proposés et ont permis de bâtir une première ébauche de machine décrite par C. Renvoise (Ren 83). Bien entendu, cette approche est une base devant conduire à de nombreuses autres réflexions et simulations pouvant apporter de profondes modifications. Il nous a semblé qu'il était important que cette thèse fasse état de ces études mais, étant donné le caractère non établi des solutions proposées, cette partie sera ici peu développée.

4.3.1 Caractéristiques d'une architecture adaptée au modèle

AMOS est une machine adaptée à l'exploitation du parallélisme inhérent aux logiciels classiques et mis en évidence par transformation programme.

Nos objectifs sont de proposer une architecture qui soit :

- peu coûteuse grâce à l'utilisation d'éléments déjà existants ;

- extensible, c'est à dire permettant la réalisation d'une gamme de machines allant de machines faibles puissances à des machines hautes performances.

Un programme "Data Flow Macroscopique" est caractérisé par un ensemble de blocs d'instructions (traitements effectifs) dont les dépendances sont traduites par des compteurs de dépendance mis à jour par des traitements de contrôle.

Ainsi deux solutions sont envisageables pour l'exécution de chaque macro instruction :

1 - les traitements effectifs sont exécutés par des processeurs de traitement et les traitements de contrôle, assurant en particulier la mise à jour des compteurs de dépendance, sont exécutés par un (ou des) processeur(s) de mise à jour.

2 - le traitement effectif et le traitement de contrôle d'une macro instruction sont exécutés par un processeur.

La première solution offre des particularités intéressantes de fonctionnement en pipeline, cependant elle nécessite la distribution aux différents processeurs d'une part des traitements effectifs et d'autre part des traitements de contrôle, augmentant ainsi les échanges d'informations dans la machine. En outre, cette solution exige de pouvoir faire exactement la distinction entre le code du traitement effectif et le code du traitement de contrôle, comme dans le modèle théorique, mais pratiquement cela est difficilement réalisable, comme dans le cas de la prise en compte de rendez-vous. C'est pourquoi il a semblé préférable d'exécuter une macro instruction sur un processeur physique. Cette solution simple peut être améliorée en associant, à chaque processeur physique, une unité de contrôle permettant de réaliser une partie des traitements de contrôle. Ainsi un recouvrement dans l'exécution des macro instructions sur chaque processeur est possible.

Le Data Flow Macroscopique fait apparaître la notion de bloc d'instructions qui peut être exploitée grâce à l'implémentation de mécanismes relativement simples permettant une accélération de l'exécution des macro instructions. Ainsi par exemple, sur une requête d'un processeur d'exécution à une unité de contrôle mémoire, un bloc d'instructions peut être chargé dans une zone propre au PE (mémoire locale ou mémoire cache). Lors de l'exécution, les instructions seront recherchées dans cette mémoire à accès rapide.

En ce qui concerne le stockage des données manipulées par chaque macro instruction deux solutions sont envisageables :

1 - les données sont réparties sur l'ensemble de la mémoire et sont accédées par chaque processeur durant l'exécution

2 - les blocs des données manipulées par chaque macro instruction sont distribués sur un système de mémoires réparties.

La deuxième solution permet une exécution des macro instructions qui soit locale à chaque processeur, en chargeant un bloc d'instruction et son bloc de données associées dans la mémoire locale du processeur. Cette solution semble satisfaisante lorsque des mécanismes associatifs de communication, comme une mémoire circulante (Lec 79), facilitent la création de ces blocs. Dans une architecture plus classique la création de blocs de données augmente les traitements de contrôle, augmente la quantité d'informations échangée, et posent de nombreux problèmes comme ceux engendrés par les variables définies dynamiquement. La première solution a donc été retenue. Elle n'exclut pas l'utilisation d'une mémoire locale propre au processeur contenant d'une part les instructions, d'autre part certaines variables locales aux macro instructions (variables temporaires) permettant d'optimiser les accès mémoire.

Ainsi, une architecture adaptée au Data Flow Macroscopique est constituée d'un ensemble de processeurs d'exécution, chacun exécutant une macro instruction, d'un réseau d'interconnexion et d'une mémoire commune contenant l'ensemble des données et instructions.

Les processeurs peuvent être plus ou moins performants et permettre des recouvrements dans l'exécution des macro instructions. Ainsi un processeur performant doté d'une mémoire propre peut être constitué de 3 unités pipeline : Une unité de recherche de macro instruction exécutable et de préchargement des instructions en mémoire locale, une unité d'exécution de traitement effectif, une unité d'exécution de traitement de contrôle.

4.3.2 Eléments d'architecture matérielle de AMOS

L'architecture générale de AMOS s'organise autour d'un réseau d'interconnexion performant et est shématiquement représentée de la manière suivante :

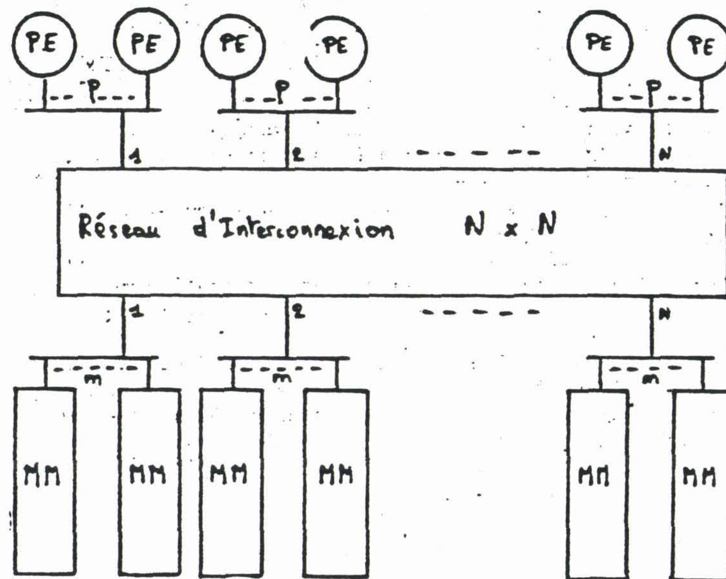


fig 14 : représentation shématique de AMOS

4.3.2.1 Le réseau d'interconnexion

L'architecture du réseau d'interconnexion est du type baseline (cf § 4.2.1.2.). C'est un réseau multi étages dont les éléments de base sont des crossbars 4x4. Le choix de ce type de connecteur permet de réduire à log4N le nombre d'étages d'un réseau N x N (au lieu de log2N avec des connecteurs 2x2). Cette approche permet d'obtenir de meilleurs résultats tant en fonctionnalités (conflits, temps de propagation) qu'en coût matériel, puisque le nombre de composants est nettement réduit par rapport à une approche utilisant des noeuds de communication 2x2. De plus cette configuration permet au réseau de conserver une bonne modularité.

Un réseau 16x16 est réalisable avec 2 étages de 4 crossbars 4x4 :

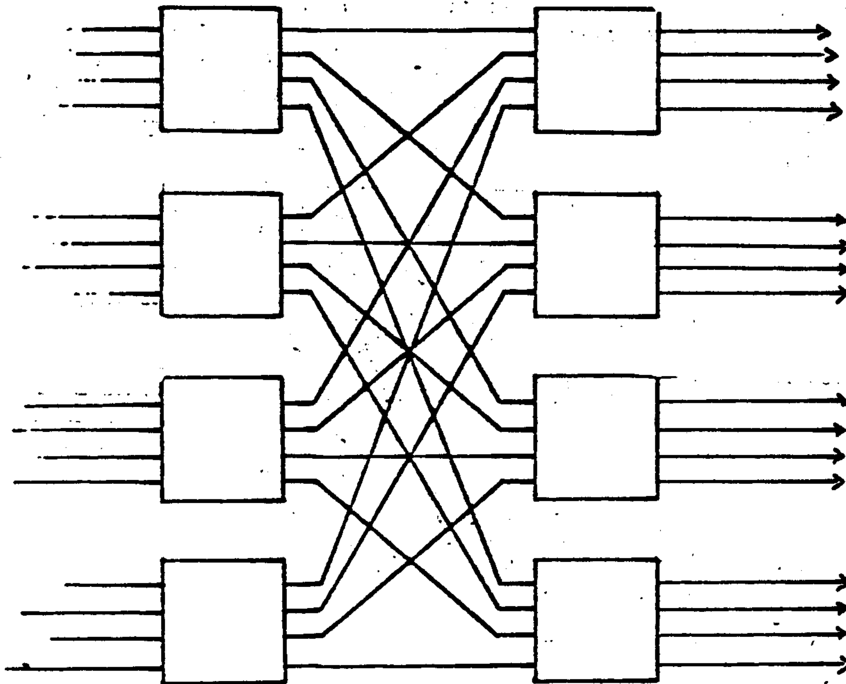


fig 15 : réseau 16x16

La commande du réseau est effectuée selon un principe de communication par paquet. Les messages traversant le réseau sont donc constitués d'une partie nécessaire au routage dans le réseau et d'une partie contenant les informations à échanger. L'état de chaque crossbar est déterminé par leur unité logique d'arbitrage et de commande en fonction des bits de routage de chaque message et de leur priorité respective en cas de conflits.

Chaque entrée de chaque crossbar dispose d'une file d'attente pouvant contenir les messages en cours de transfert, de les stocker durant le temps nécessaire à la commande des connecteurs et lorsqu'un conflit bloque un ou plusieurs d'entre eux.

Cette technique permet d'augmenter de façon importante les performances du réseau grâce à un pipeline entre les traversées de chaque étage afin que le temps apparant de traversée du réseau soit égal au temps de traversée d'un étage. De plus la vitesse de traversée d'un étage est accélérée grâce à un pipeline entre la commande d'un crossbar et le transfert des informations qui le traversent.

Pour des raisons d'efficacité et de diminution de charge, les réseaux utilisés sont unidirectionnels, ainsi deux réseaux sont nécessaires pour assurer les échanges, l'un achemine les informations des processeurs à la mémoire et l'autre de la mémoire aux processeurs.

Le temps de traversée d'un étage varie de 100 à 40 ns, selon les méthodes de commande et les technologies utilisées.

4.3.2.2 <u>La mémoire principale</u>

La mémoire principale est décomposée en N * m modules physiques indépendants répartis en N grappes de m modules chaque. Les modules d'une grappe sont sollicités successivement en lecture ou en écriture, un banc ne pouvant être resollicité qu'après un temps égal à son temps de cycle. Le nombre m est choisi pour permettre qu'un module puisse être accédé dès que possible, c'est à dire dès la fin de son temps de cycle. Ainsi, il n'y a pas de retard dans l'accès mémoire. Chaque grappe peut recevoir ou émettre un message à la fréquence du réseau d'interconnexion ce qui permet d'atteindre des hauts débits mémoire si nécessaire.

Ainsi, par exemple, si le temps cycle d'un banc mémoire est de 200 ns et que le débit de chaque sortie du réseau est d'une information toutes les 50 ns, les grappes comporteront 4 bancs mémoire.

Il est prévu de disposer des files d'attente devant chaque grappe mémoire (ou devant chaque banc mémoire si nécessaire) afin de stocker les requêtes qui ne peuvent pas être satisfaites immédiatement.

Le principe de la communication par paquet entraine des problèmes supplémentaires lors de la modification en mutuelle exclusion de certaines variables stockées en mémoire principale.

Une méthode décrite par Weathely et Leathrum (XL 82) permet de résoudre ce problème. Elle consiste en l'utilisation d'unité de contrôle devant les modules (ou les grappes) afin de pouvoir réaliser la fonction "lecture, modification, écriture" en un seul cycle instruction.

De plus, des mécanismes d'accélération d'accès aux données peuvent être implémentés afin de rendre le système plus performant. Ainsi, une mémoire associative associée à chaque unité de contrôle mémoire permet de stocker momentanément certains types d'informations accédées fréquemment et de les retrouver rapidement lorsqu'elles sont lues ou modifiées.

Ces techniques sont particulièrement adaptées à AMOS où de nombreuses opérations de décrémentation de compteurs de dépendance, ressources àccessibles simultanément par plusieurs processeurs, doivent être exécutées en mutuelle exclusion.

Ainsi, par exemple, lorsqu'un processeur décrémente un compteur de dépendance il envoie à une unité de contrôle mémoire une commande "lire, décrémenter, écrire". A la réception du message, celle-ci teste si ce compteur se trouve dans sa mémoire associative, si oui, elle le modifie, si non, elle lit en mémoire la valeur du compteur, le modifie et le range en mémoire associative en appliquant une méthode de gestion de type FIFO ou LRU. Ce principe de fonctionnement permet de diviser par quatre le temps de modification d'un compteur de dépendance.

D'une manière générale de nombreux problèmes peuvent être résolus efficacement en donnant plus "d'intelligence" à la fonction de stockage mais en contrepartie, de telles solutions coûtent en matériel. Si ces solutions sont selon nous souhaitables, il ne faut cependant pas tomber dans un excés de matériel qui engendrerait un coût de conception important et diminuerait la fiabilité de la machine. Ainsi, par exemple, quel sera le coût et la fiabilité d'un réseau tel celui proposé par Gottlieb (Go 83) pour sa machine NYU. Ultracomputer (4096 PE(s)), où chaque connecteur dispose d'une logique suffisante pour réaliser des primitives "fetch-and-add", relativement complexe, afin de réduire les problèmes de conflits?

4.3.2.3 Les processeurs d'exécution

Les processeurs d'exécution sont organisés en grappe, chacune des grappes étant connectée à une des entrées du réseau. Le principe de grappes de processeurs permet d'utiliser au mieux les capacités du réseau d'interconnexion en fonction des fréquences caractéristiques des éléments de l'architecture. Ainsi, en supposant qu'un processeur puisse émettre une requête mémoire en moyenne toutes le 1,8 us et que chaque entrée du réseau d'interconnexion soit capable de prendre en compte une requête toutes les 100 ns (temps de commande et de traversée d'un étage), alors 18 processeurs sont nécessaires pour saturer l'accès mémoire au travers d'un point d'accès du réseau.

La notion de grappe permet l'utilisation de micro processeur du commerce comme processeur élémentaire (PE) d'une machine parallèle, tout en gardant de bonne performance. En effet, il suffit d'adapter le nombre de micro processeurs par grappe au débit de chaque accès du réseau pour obtenir une configuration permettant une utilisation optimum du système de communication.

Au delà de l'incidence sur les performances, cela a une portée sur la fonctionnalité système de la machine. Ainsi un processeur de gestion de grappe peut être associé à chaque grappe de PE. Celui-ci est alors un élément priviligié pour l'intégration de la grappe de PE sur la machine, et exécute des fonctions du système de base pour les PE de la grappe. A titre d'exemple, il tiendrait à jour l'état d'activité des processeurs de la grappe (opérationnels, actifs, suspendus, ...), serait responsable de l'attribution des travaux aux PE de la grappe (distribution préemption éventuelle ...). Il assurerait une transparence de la nature des PE pour le reste du système.

La notion de grappe apparait intéressante pour réaliser une machine plus tolérante aux pannes et plus fiable. Ainsi, sur une grappe des techniques de contrôle mutuel peuvent être implantées. Le processeur de grappe peut être l'un des processeurs de la grappe de façon à ce que, en cas de défaillance, l'un des processeurs de la grappe vienne se substituer à ce dernier.

CHAPITRE V

MISE EN OEUVRE DU DATA FLOW MACROSCOPIQUE SUR MLI

5.0 INTRODUCTION

Il est désormais classique :

- que les machines modernes soient dotées d'une mémoire virtuelle segmentée ;

- qu'un programme ait accès à un ou plusieurs segments dont les références sont regroupées dans une ou des tables qui sont utilisées pour l'interprétation des adresses vues du programme.

Ainsi dans l'IAPX 432 la traduction d'une adresse logique en adresse physique est faite en deux étapes, sélection d'un descripteur d'accès dans un segment d'accès, puis sélection d'un descripteur de segment dans une table de segments.

Ces principes restent applicables et satisfaisants pour notre machine où un programme est décomposé en macro-instructions. Ainsi, dans ce chapitre il est montré la compatibilité entre les structures systèmes adoptées dans une machine comme MLI (§ 4.1) et la structure des programmes parallélisés selon le modèle Data Flow Macroscopique.

En particulier, les structures d'adressage, la création et la désignation des vecteurs d'état, les structures de données nécessaires aux rendez-vous et aux exceptions, sont examinées.

5.1. Data Flow Macroscopique et Unité de Compilation

Les langages à structures de blocs comme Ada, permettent la conception de systèmes modulaires entrainant le développement de programmes industriels d'une plus grande fiabilité.

Ainsi le développement d'un programme Ada en sous unités (corps d'un sous programme, d'un package ou d'une tâche) permet un découpage du programme en entités de tailles raisonnables pour permettre :

- une compréhension plus facile du programme

- une compilation ou recompilation plus rapide limitée seulement aux sous unités affectées par une modification de programme.

La modularité des programmes Ada, la structure de blocs et les règles de visibilité sont des notions contribuant à la fiabilité des programmes. Elles doivent être respectées par toute transformation de programme ou parallélisation.

La notion d'unité de compilation séparée entraine quelques restriction au niveau de l'exploitation du parallélisme.

En effet, il n'est pas possible de déterminer les relations de dépendance entre les traitements d'une unité de compilation séparée et les autres unités du programme la nature de ces traitements leur étant inconnue du fait de la compilation séparée. Par conséquent, les unités de compilation d'un programme doivent être exécutées suivant l'ordre canonique, c'est à dire l'ordre défini par les règles du langage, et seul le parallélisme interne à chaque unité de compilation peut être exploité. Il en est de même pour les unités de librairie, sauf si elles ne partagent pas d'objet avec le programme (comme lorsqu'elles sont des sous programmes), auquel cas les dépendances ne sont liées qu'aux paramètres de communication.

Toute unité de compilation à l'intérieur d'une "unité de librairie" est parallélisable. Il faut lors d'une parallélisation seulement prendre en compte les variables visibles, lors d'une parallélisation, en dehors de l'unité de librairie.

Une unité de compilation est donc un "sous système" et est représentée dans le système macroscopique par une macro-instruction. Lorsque cette macro instruction est suffisamment complexe et parallélisable, elle peut être décomposée en un système de macro instructions dont les enchainements sont décrits par un graphe de dépendance. A une unité de compilation il correspond alors un vecteur d'état constitué des mots d'états des macro instructions qui la composent.

Lorsqu'une unité de compilation appelle une autre unité de compilation, celle-ci est représentée dans le vecteur d'état de l'unité appelante par une macro instruction molécule qui, lorsqu'elle est exécutée, crée le vecteur d'état de l'unité appelé correspondante est rend exécutables ses macro instructions initiales. Par la suite, les macro instructions terminales de cette unité décrémentent le compteur de dépendance de la macro instruction molécule de façon à la réactivée.Cette technique est semblable à celle utilisée pour l'appel de sous programme (§ 2.3.3.).

Il se peut que la décomposition macroscopique ne fasse pas apparaître une unité de compilation, son appel est alors "caché" dans le traitement effectif d'un atome de l'unité appelante.

Ainsi, une unité de compilation possède en général un vecteur d'état qui lui est propre et a visibilité sur les objets des unités englobantes, en particulier sur les mots de leur vecteur d'état. Les mécanismes de décomposition macroscopique sont donc compatibles avec ceux de l'adressage lexical (§ 4.1) et il est possible de représenter un vecteur d'état d'une unité par un segment logique Vecteur d'état (noté SLVE) contenu dans un segment virtuel de vecteurs d'état.

L'adresse d'un mot d'état est du type i, c, d, où :

- i représente le niveau lexical et désigne un descripteur de la table des niveaux lexicaux (TNL)

- c représente la catégorie d'objets mots d'état et désigne un descripteur du segment logique de contexte lexical (SLCL)

- d représente le déplacement dans le SLVE et désigne le mot d'état à accéder.

Par exemple, soit l'exécution d'un sous programme P de niveau lexical i pour une tâche T. Le système d'adressage est caractérisé par un BCP pour la tâche T et sa TNL associée, qui permet d'accéder à la SLCL du sous programme.

Les descripteurs de cette table désignent les ensembles d'objets propres au sous programme, en particulier son SLVE.

BCP de la
Tâche T

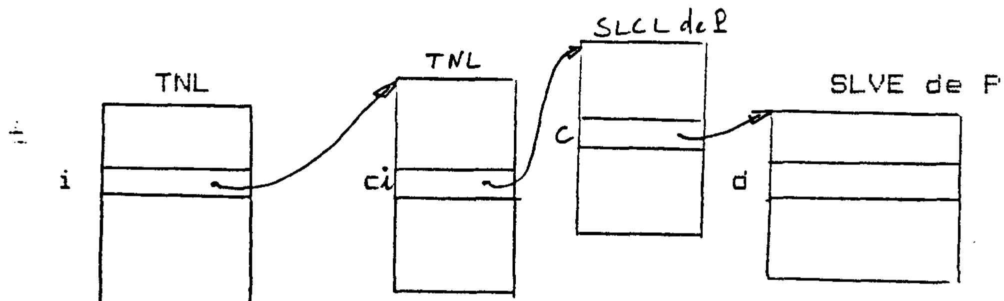


fig 1 : adressage des mots d'états d'un sous programme de niveau lexical i dans une tâche T

L'activation d'une unité de compilation de niveau lexical i+1 (UC i+1) a pour effet de créer un segment logique vecteur d'état si cela est nécessaire. Le shéma d'adressage devient :

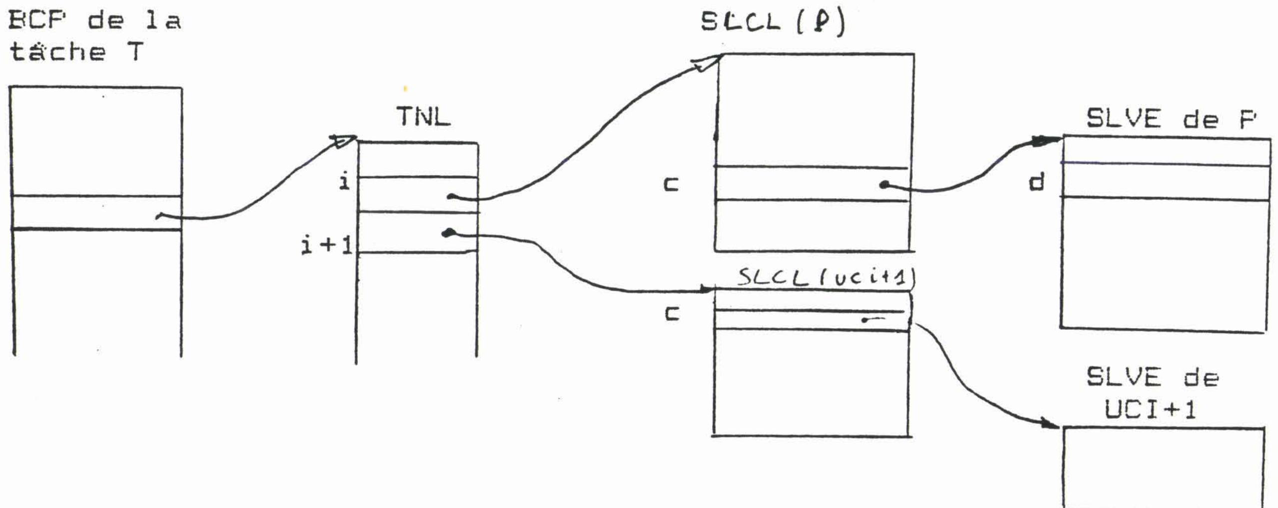


fig 2 : adressage des mots d'état d'une unité de compilation de niveau lexical i+1

5.1.2. <u>Exploitation du parallélisme dans une unité de compilation</u>

Une unité de compilation est donc un sous système parallélisable par décomposition macroscopique. L'enchainement de l'ensemble des macro instructions obtenues et décrit par un graphe de dépendances et permet l'exécution parallèle de sous programmes appartenant à des niveaux lexicaux différents. Il faut cependant vérifier que leur exécution reste possible étant donné le système d'adressage choisi.

Initialement, l'approche proposée pour MLI était basée sur la création de segments logiques propre à chaque sous programme, c'est à dire la création d'une SLCL propre, accessible par une entrée de la TNL et divers segments logiques de "données". Cependant, à des fins d'optimisations, certains sous programmes lexicaux de même confidentialité (c'est à dire qui ne sont pas appelés en dehors de l'unité qui la contient, qui sont non récursifs ou activent n'admet pas de tâche) sont regroupés et partages un même segment logique de contexte lexical et les mêmes segments logiques de données. Un tel regroupement est appelé "unité de programmation". Chaque unité de programmation possède un niveau lexical qui est le niveau lexical de référence des sous programmes qu'elle regroupe.

Par la suite nous appellerons "unité de programmation" toute unité créant ses propres segments logiques.

En particulier, une unité de programmation dispose de son propre segment logique vecteur d'état. Ainsi une unité de programmation est représentée dans le vecteur d'état de l'unité appelante par une molécule "unité de programmation" dont l'activation a pour effet, en particulier, de créer les segments logiques de l'unité correspondante, dont le segment logique vecteur d'état. Il arrive fréquemment qu'une unité de programmation soit une unité de compilation.

Pour respecter les mécanismes de construction du système d'adressage les appels et retours de sous programmes se font en respectant l'ordre canonique, cet ordre doit être respecté par le système de macro instruction du Data Flow Macroscopique.

5.1.3. Exécution parallèle d'unité de programmation non-imbriquées

Il est particulièrement complexe et coûteux de chercher à rendre parallèle l'exécution de sous programmes (ou unité de programmation) non imbriqués, car cela engendre des problèmes d'ambiguité sur les niveaux lexicaux.

Ainsi par exemple, soit une unité de programmation A de niveau lexical i-1 appelant deux sous programmes B et C de niveau lexical i.

Les sous-programmes B et C ont la même visibilité sur les données de A et ses unités englobantes. En supposant que les écritures et les lectures de B n'interfèrent pas sur celles de C, et réciproquement, alors les exécutions de ces deux sous programmes sont potentiellement réalisables en parallèle.

Lors d'une exécution séquentielle les données locales de B sont accédées grâce à une adresse de type i, c, d, qui détermine un chemin d'accès au travers des tables TNL, SLCL et SL de données. la fin d'exécution de B entraine la suppression de l'entrée i de la TNL et des tables SLCL et SLD. Lors de l'appel de c, une entrée i est créée dans la TNL permettant l'accès à la SLCL de c.

Ainsi, une exécution parallèle de B et C engendre une ambiguîté dans l'accès à leurs données locales puisqu'elles sont toutes les deux de la formes (i, c, d). Pour réaliser un tel système, il faut donc que l'adressage des données de A et B soit indépendant à partir du niveau lexical i.

Une solution consiste à associer à chaque unité de programmation parallèle une TNL dont les entrées sont identiques, jusqu'au niveau i-1, à celles de l'unité appelante.

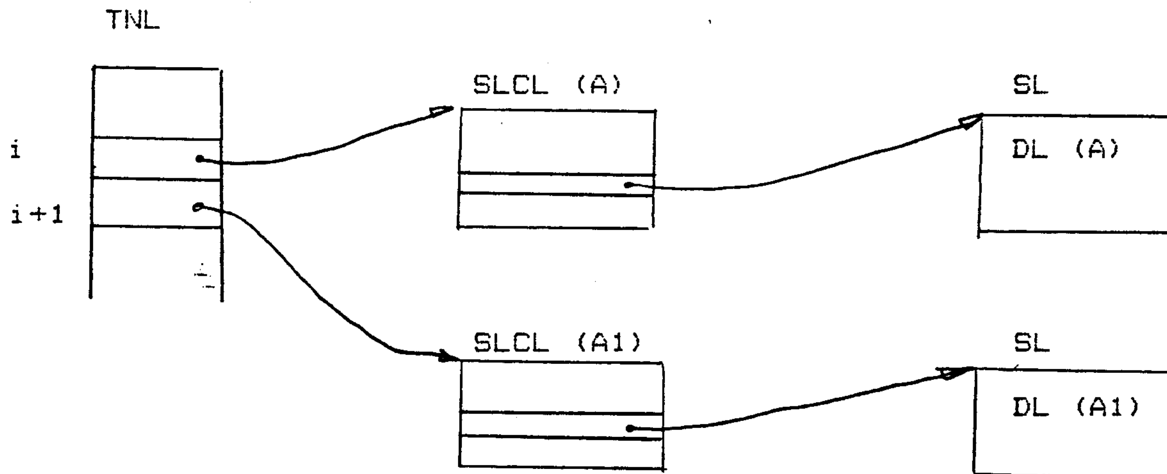
L'entrée i de la TNL de ces unités permet d'accéder à leur SLCL et leurs segments logiques. Il faut, de plus, se doter d'un mécanisme permettant d'accéder à la bonne TNL pour chaque unité exécutée en parallèle.

La réalisation d'un tel niveau de parallélisme, quoique simple en principe, est coûteux à réaliser aussi bien en temps qu'en espace mémoire, puisqu'il faut recopier la TNL (c'est à dire le contexte). Les sous programmes non imbriqués ne seront donc pas rendus parallèles du fait du mode d'adressage de la machine.

5.1.4. Exécution parallèle d'unités de programmation imbriquées

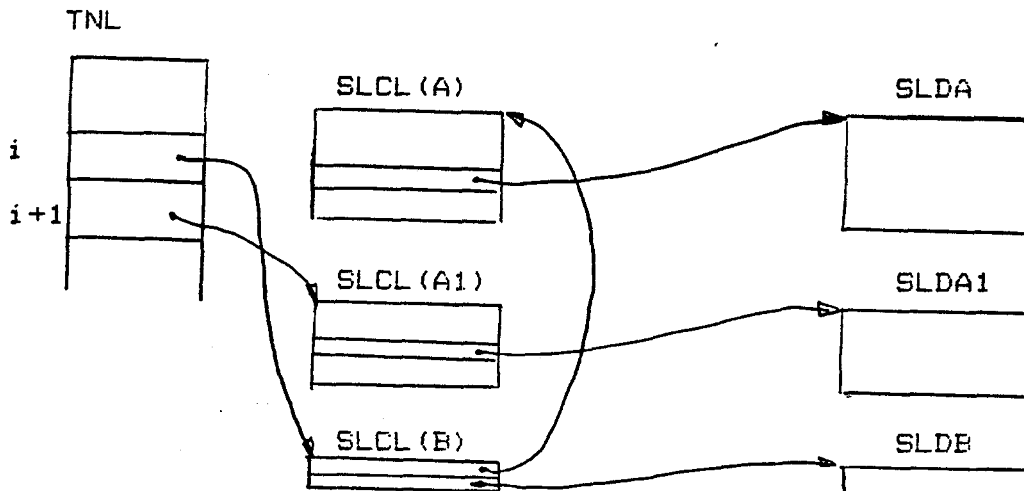
Pour la machine langage, l'appel à une unité de compilation se traduit par la création d'un segment logique de contexte lexical (SLCL) et la création de segments logiques propres (segment de données).

Ainsi, soit une unité de programmation A de niveau lexical i appelant une unité de programmation A1 de niveau lexical i+1. L'état de référence de la machine est :



Dans ce cas, l'exécution parallèle des unités appelante et appelée ne pose pas de problème d'accès aux données puisqu'il y a bijection entre les niveaux lexicaux i, i+1 et les SLCL(A), et SLCL(A1). Par conséquent l'adressage (i,c,d) est sans ambiguïté.

Cependant, l'unité A1 a visibilité sur tous les niveaux inférieurs ou égals à i+1 et peut faire appel a une unité B de niveau lexical i. L'état de référence devient :



Dans ce cas, l'exécution parallèle de l'appelant A1 et de l'appelé n'est plus possible, car le niveau lexical i correspond à deux unités (A et B). L'entrée i de la TNL pointe sur la SLCL de l'unité B et la SLCL de A n'est plus directement accessible à partir de la TNL.

Si l'exécution parallèle de A1 et de B est possible, alors, quand A1 accède à une donnée d'adresse (i,c,d), le shéma est tel que c'est une donnée d'un segment logique propre à B qui est atteinte.

Pour résoudre ce genre de problème trois solutions peuvent être envisagée.

- <u>Solution 1</u>

Soit i le niveau lexcial de l'unité de programmation appelante(A). Si le niveau lexical de l'unité appelé(B) est j, l'exécution parallèle des sous programmes de niveau lexical $\geqslant$ j n'est pas autorisée.

En effet, lorsque A appel B l'exécution parallèle avec B des sous programmes de niveau lexical inférieur à j (c'est à dire visible par B) est possible puisque les SLCL de ces unités sont directement accessibles à partir des entrées de la TNL.

Ainsi, pour l'exemple précédent, lorsque l'appel de B est réalisé alors l'exécution de A1 et de A doit être interrompue car les données locales de A, visible par A1, ne sont plus accessibles.

La décomposition macroscopique de l'unité de compilation et le graphe de dépendance du système de macro instructions correspondant peuvent traduire ce type de parallélisme.

- <u>Solution 2</u>

La solution précédente est restrictive et ne permet pas, en particulier, l'exécution parallèle de sous programmes récursifs. La méthode proposée ici offre une plus grande liberté d'exécution parallèle, moyennant l'utilisation de mécanismes associatifs par chaque processeur d'exécution.

En outre, une technique d'adressage utilisant des tables d'accessibilité doit être adoptée.

Chaque unité de programmation dispose, dans sa SLCL, d'une table de référence aux segments logiques de contexte lexical qu'elle accède (table d'accessibilité). Cette table est construite au moment de l'appel car les appels (et les retours) se font suivant l'ordre canonnique.

Aussi, les informations contenues dans la TNL au moment d'un appel permettent d'accéder aux SLCL visibles par l'unité appelée et par conséquent de construire une table de référence aux segments de contexte lexicaux accessibles.

Chaque processeur d'exécution possède un T.A.L. (§4.1.4) (Transformateur d'Adresse Lexical) dont le but est d'accélérer le calcul d'adresse au moyen de mécanismes associatifs. Ainsi, lorsqu'un processeur fait référence à un segment logique d'une unité de niveau i, le T.A.L. cherche dans sa mémoire associative si une référence au niveau lexical i a déjà été effectuée et si oui alors il associe la valeur de l'adresse du SLCL correspondant.

L'emploi du T.A.L. permet donc de ne pas utiliser les informations de la TNL. Quand le T.A.L. est chargé, il se comporte comme une table d'accessibilité locale au processeur d'exécution.

Ainsi, lorsqu'une unité est exécutée par un processeur, il suffit de charger la mémoire associative de son T.A.L. avec les informations contenues dans la table d'accessibilité du SLCL de cette unité. Dès los, aux différents niveaux lexicaux sont associés, par le T.A.L., les adresses du segment logique de contexte lexical des unités accédées par l'unité exécutée.

Les mécanismes de gestion des informations du T.A.L. devront être adaptés à ce type d'utilisation. En particulier, lorsqu'il n'y a plus de place dans les entrées du T.A.L. il ne faut pas écraser, par l'introduction de nouvelles informations, les entrées associées à la table d'accessibilité.

L'emploi du T.A.L. permet de ne pas utiliser la TNL qui était une source de conflits d'adressage lors de l'exécution parallèle de sous programmes.

Cependant, lors de l'activation d'une macro instruction, il faut pouvoir retrouver l'adresse du SLCL qui lui est associée afin de pouvoir lire les informations nécessaires à l'initialisation du T.A.L. du processeur d'exécution. Pour cela, chaque segment logique vecteur d'état contiendra dans une de ses entrées l'adresse du SLCL auquel il appartient. Ce champ sera initialisé au moment de la création du vecteur d'état (c'est à dire de l'appel de l'unité).

Par la suite, lorsqu'une macro instruction deviendra exécutable, il lui sera associée l'adresse du SLCL de son vecteur d'état. Cette adresse sera utilisée par le processeur qui aura la charge d'exécuter cette macro instruction afin d'initialiser son T.A.L..

Mais cette solution n'est pas facile à mettre en oeuvre et une troisième solution intermédiaire aux deux première peut être envisagée.

- Solution 3

Cette solution est une partie de la solution 1 dans la mesure où si une unité de programmation de niveau lexical i est appelée alors l'exécution parallèle de toutes les unités de niveau lexical strictement supérieur est interdite. C'est une partie de la solution 2 dans la mesure où l'exécution simultanée de plusieurs appels d'un programme récursif est autorisée, grâce à l'utilisation es T.A.L.

L'appel d'un programme récursif correspond à l'appel d'une unité de programmation et se traduit par la création d'un SLCL et de segments logiques propres, en particulier d'un segment logique vecteur d'état. Une entrée de ce vecteur d'état est initialisée au moment de l'appel, avecl'adresse du SLCL auquel il appartient.

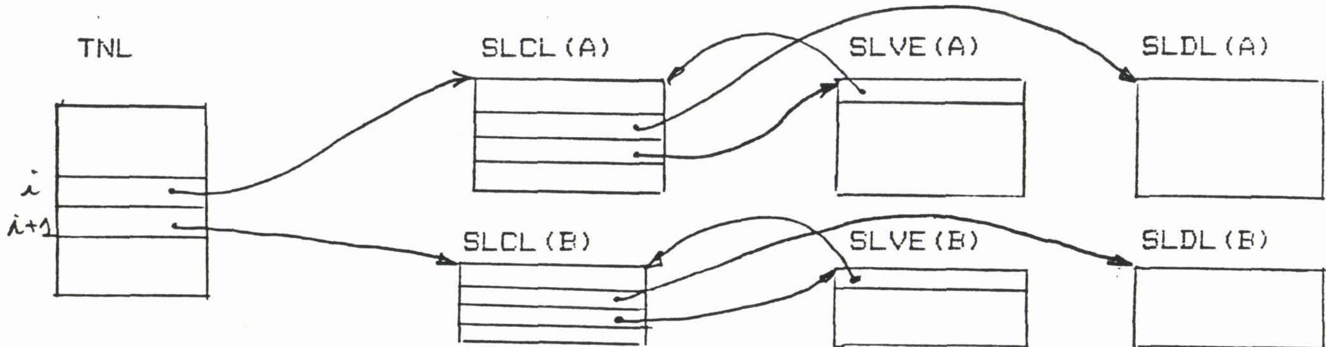
Lorsqu'une macro instruction devient exécutable cette adresse lui est associée. Ainsi le processeur, qui aura la charge d'exécuter cette macro-instruction, utilisera cette adresse pour initialiser son T.A.L.. Celui-ci associera au niveau i l'adresse de la SLCL du programme récursif correspondat.

Cette adresse ne devra pas être détruite durant toute l'exécution de la macro-instruction.

Cette technique permet une exécution parallèle de plusieurs appels de sous programmes récursifs en rendant accessible leurs données locales à chaque processeur d'exécution. En effet, il n'y a pas de conflit dans l'accès des données locales de chaque instanciation du sous-programme puisque la TNL n'intervient pas dans ce mécanisme d'adressage.

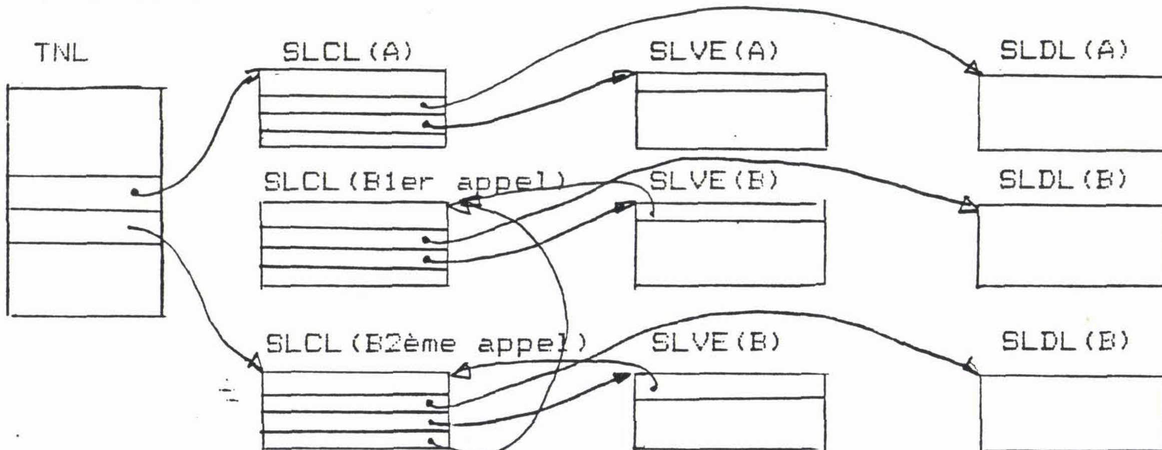
L'exemple suivant montre comment un sous programme A de niveau i peut s'exécuter en parallèle avec les instanciations d'un sous programme récursif de niveau i+1 qu'il appelle.

Lors de la première instanciation de B, l'état de référence de la machine est :



où : SLVE signifie segment logique vecteur d'état et SLDL signifie segment logique de données locales.

Lors de la deuxième instanciation de B l'état de la machine devient :



L'exécution de toute macro instruction du premier appel de B a entrainé la précharge de la mémoire associative du TAL du processeur d'exécution avec l'adresse du SLCL (B 1er appel). Toute référence au niveau i+1 pointe alors sur le SLCL B 1er appel grâce au TAL.

L'exécution de toute macro instruction du deuxième appel de B a entrainé la précharge de la mémoire associative du T.A.L. du processeur d'exécution avec l'adresse du SLCL B 2ème appel. Toute référence au niveau i+1 pointe sur le SLCL B deuxième appel grâce au T.A.L..

5.1.5. Exécution parallèle des "unités de programmation"

Dans le cas où un sous programme est local a une unité de programmation, son niveau lexical est celui de l'unité de programmation et les informations concernant son contexte sont incluses dans le SLCL de celle-ci.

Du point de vue de l'accès aux données tout se passe comme si la notion de sous programme n'apparaissait pas et le sous programme semble inclu à l'intérieur de l'appelant. Ainsi l'adresse d'une donnée à l'intérieur d'une unité de programmation est de la forme (i,c,d) où i est le niveau lexical de référence, c un déplacement dans la SLCL de l'unité et d'un déplacement dans un segment logique de donnée.

La SLCL d'une unité de programmation contient une zone mémoire de taille variable dans laquelle sont sauvegardées les adresses de retour de sous programmes.

Cette zone mémoire est gérée comme une pile au moyen d'un registre qui a tout moment désigne, parmi tous les sous programmes du niveau lexical courant, celui qui s'exécute. Cette structure impose que l'ordre des appels et retours de sous programmes dans un programme parallélisé soit l'ordre canonique. Les informations contenues dans la SLCL ne sont alors pas modifiées.

Bien entendu, cette contrainte disparait si une technique d'optimisation de type insertion en ligne est utilisée. Elle vise à faire disparaitre la notion de sous programme en insérant le code du sous programme appelé dans celui du sous programme appelant. Dans ce cas, les macro instructions du sous programme inséré font partie du graphe de dépendance du sous programme.

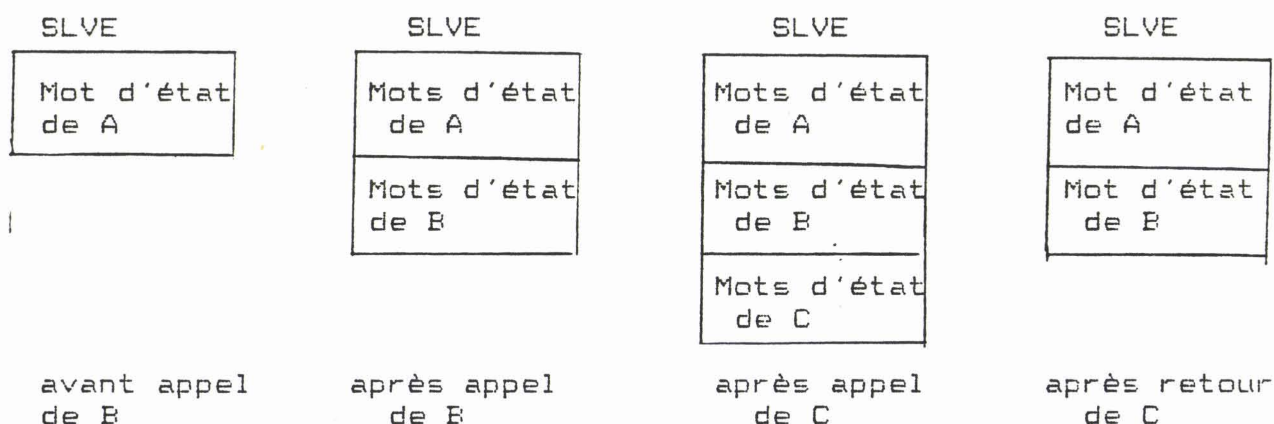
Puisqu'il n'y a pas de recouvrement des informations d'accès des sous programmes appelé et appelant, alors il n'y a pas d'ambiguité dans l'adressage de leurs données. Ainsi l'exécution parallèle (partielle ou totale) d'un sous programme appelant avec un sous programme appelé peut être réalisée sans modification du système d'adressage, si les relations de dépendances entre leurs différentes macro instructions, mises en évidence par l'outil de parallélisation automatique des programmes, le permettent.

Une unité de programmation est donc un regroupement logique d'unité de même confidentialité disposant d'un segment logique de données et d'un segment logique vecteur d'état. L'adresse des mots d'état d'une unité de programmation est de la forme (i,c,d) ou i est le niveau lexical de référence.

La gestion du segment logique vecteur d'état d'une unité de programmation peut se faire dynamiquement.

Dans ce cas, lorsqu'une molécule sous programme est activée elle écrit, dans le segment logique vecteur d'état de son unité de programmation, le vecteur d'état local qui lui correspond, c'est à dire l'ensemble des mots d'état asssociés au graphe de dépendance du sous programme. Au retour du sous programme, c'est à dire lorsque la molécule sous programme est réactivée, le vecteur d'état local associé est retiré du SLVE. L'ordre des appels et des retours de sous programmes étant l'ordre canonique le SLVE se comporte comme une pile des "vecteurs d'état locaux" des sous programmes locaux à l'unité de programmation.

Ainsi par exemple, si une unité de programmation A appelle un sous programme B, qui appelle un sous programme C, alors le vecteur d'état de l'unité de programmation aura les différentes formes suivantes selon les appels et retours de sous programmes effectués :



avant appel de B — après appel de B — après appel de C — après retour de C

Cependant, toute gestion dynamique demande du temps de calcul lors de l'exécution du programme, il peut être préférable, lorsque cela est possible, d'exécuter la plupart des contrôles de façon statique, c'est à dire lors de la compilation.

Ainsi l'application d'une méthode d'inclusion textuelle, permettant de définir statiquement (complètement ou partiellement) le vecteur d'état d'une unité de programmation en insérant les mots d'état de certains sous programmes dans le vecteur de leur unité appelante, présente un "overhead" temps inférieur, mais entraine un encombrement mémoire supérieur, dans la mesure où les représentations des macro instructions d'un sous programme appelé en plusieurs endroits sont dupliquées.

5.1.6. Exécution parallèle de boucles dynamiques

Tant que la décomposition d'un programme est statique, la structure des vecteurs d'états des différentes unités exécutées est connue à la compilation et ne pose pas de problème d'implémentation, puisqu'elle est bien définie. Par contre, lorsque la décomposition introduit des structures permettant la création dynamique de mots d'état, comme pour les sous programmes ou les boucles parallèles dont le coefficient d'itération est calculé dynamiquement, alors la structure des vecteurs d'état n'est pas entièrement définie à la compilation, leur taille est variable, et leur implémentation plus difficile à réaliser.

En ce qui concerne les appels de sous programmes des solutions ont été apportées dans les sections précédentes. Dans ce paragraphe les problèmes d'implémentation du vecteur d'état d'une unité contenant des boucles dynamiques est étudié.

Une boucle dynamique est caractérisée par une molécule dont le rôle est de générer, dans le vecteur d'état de l'unité propriétaire, les n-uplets décrivant les itérations à exécuter. Plusieurs solutions de gestion du vecteur d'état sont envisageables.

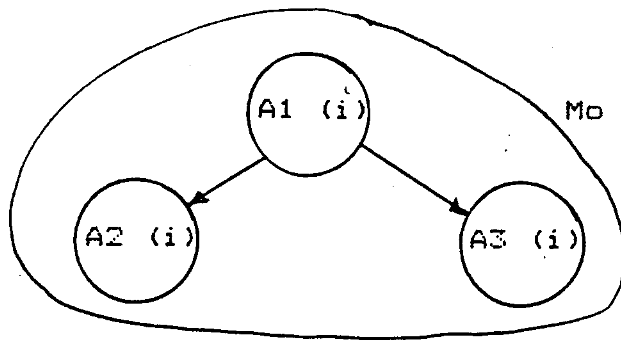
- Solution 1

La première solution consiste à créer un segment logique vecteur d'état propre à la boucle dynamique dans le segment virtuel vecteur d'état.

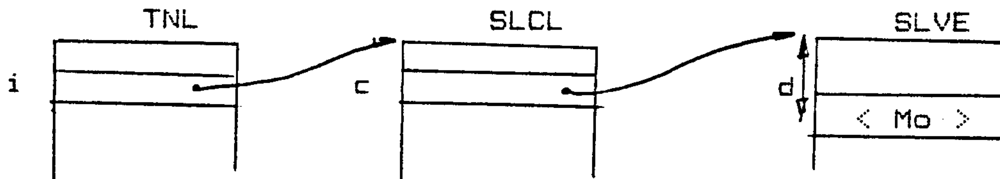
Lorsqu'une molécule boucle est activée, elle crée un segment logique vecteur d'état qu'elle initialise avec l'ensemble des n-uplets correspondants aux mots d'états des N itérations de la boucle à exécuter. Les ensembles de mots d'état de chaque itération sont rangés successivement dans le segment logique en suivant leur numéro d'ordre d'itération. De plus une catégorie, de la zone de la SLCL réservée aux capacités des segments logiques de l'unité contenant la boucle, est initialisée avec l'adresse de base du nouveau segment logique.

Ainsi, l'adresse d'un mot d'état de la boucle dynamique est de la forme (i,c,d) où i est le nouveau lexical de référence l'unité propriétaire, c une catégorie de la SLCL associée permettant de désigner l'adresse de base du segment logique vecteur d'état de la boucle.

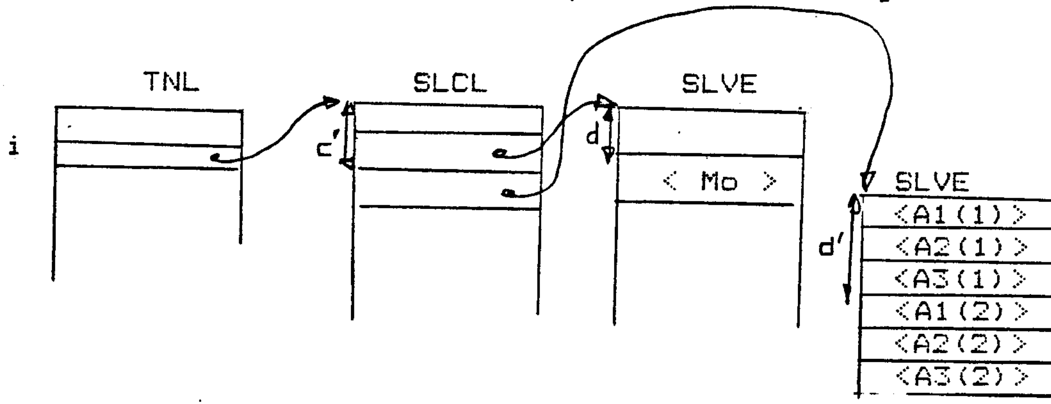
Par exemple, soit une boucle dynamique correspondant a une molécule composée de trois atomes :



Le système d'adressage avant l'activation de Mo est :



Lorsque la molécule Mo est activée, un nouveau segment logique vecteur d'état est créé. En supposant que la boucle comporte deux itérations le système d'adressage devient :



Les mots d'états des différentes macro instructions de chaque itération ont pour adresse (i,c',d') où d est un déplacement, dans un segment logique vecteur d'état, calculé en fonction du numéro d'itération dans la boucle.

- Solution 2

La méthode précédente propose la création, lors de l'activation d'une boucle dynamique, d'un segment logique vecteur d'état. Cette solution comme toute méthode dynamique, nécessite du temps de calcul durant l'exécution du programme. Dans le cas particulier où le nombre d'itérations de la boucle est borné, et que cette borne est connue à la compilation, alors il est possible d'éviter la création d'un nouveau segment logique vecteur d'état, en réservant, dans le vecteur d'état de l'unité propriétaire, un nombre d'emplacements égal au nombre de macro instructions de l'ensemble des itérations exécutées en parallèle.

Cette méthode est réalisable parce qu'il est possible de déterminer, à la compilation, la taille mémoire maximum nécessair au stockage des mots d'état qui seront générés.

Cette solution est également bien adaptée à l'exécution parallèle des itérations d'une boucle dans un mode constant, c'est à dire que le nombre d'itérations exécutables en parallèle est fixe tout au long de l'exécution de la boucle, et que la réactivation d'un nouvel ensemble d'itérations est conditionnée par la terminaison de l'ensemble des itérations activées au coup précédent (Ren 83).

5.2. DATA FLOW MACROSCOPIQUE ET MECANISMES DE RENDEZ-VOUS

L'implémenation du Data Flow Macroscopique sur une machine telle que MLI (§ 5.1) n'entraine que peut de modifications des structures systèmes utilisées dans ce système de base. En particulier, les structures de données nécessaires pour mettre en oeuvre les mécanismes de rendez-vous sont peu différentes.

Cette simplicitée est principalement dûe aux règles de décomposition macroscopique qui assurent que pour chaque tâche l'ordre d'exécution des points de synchronisation est identique à l'ordre canonique. Le découpage des tâches en macro instructions et leurs relations de dépendance, décrite par un graphe de dépendance, garantissent l'exécution séquentielle des points de rendez-vous dans l'ordre du programme d'origine.

Les accès tables systèmes contenant les variables d'état caractérisant tous les rendez-vous sont des ressources partagées entre plusieurs processus et doivent être, par conséquent, accédées en mutuelle exclusion.

Ces informations sont contenues dans le bloc de contrôle de processus (BCP) de chaque tâche. Il a été montré au paragraphe 4.1.3.2. que dans MLI les points de communication entre processus sont matérialisés, dans chaque BCP, par des têtes de queues correspondant aux entrées propres à une tâche.

Chaque tête de queue permet une mémorisation, sous la forme d'une liste, des processus qui veulent communiquer sur l'entrée associée. Elle contient des variables caractérisant l'état du processus appelé (ou propriétaire) pour cette entrée. Lorsque la décomposition macroscopique prend en compte la structure de rendez-vous d'une tâche appelé deux paramètres supplémentaires, I et @MI, sont nécessaires. I indique si la décomposition macroscopique prend en compte la structure de rendez-vous de l'appelant, auquel cas @MI est l'adresse du mot d'état de la macro instruction à rendre exécutable, c'est à dire, celle associée au traitement de rendez-vous.

La nouvelle tête de queue à la structure suivante :

OF	COMPTEUR	DEPL
I	@ MI	
PREMIER		
DERNIER		

Les maillons associés à chaque tête de queue contiennent les informations caractéristiques des processus qui veulent communiquer. Lorsque le Data Flow Macroscopique est mis en oeuvre, ils doivent contenir un indicateur I, signalant si l'appel associé est caché ou non dans un atome, et l'adresse, @ MI, du mot d'état de la macro instrúction de la tâche appelante à activer après le rendez-vous.

Les maillons ont la structure suivante :

I D - PROCESSUS			
P	RV	PR	SUIVANT
I	@ MI		
PARAMETRES			

De même les maillons du sémaphore délai contiendront un indicateur de décomposition I et l'adresse du mot d'état d'une macro instruction à rendre exécutable.

Ainsi, lorsque le délai associé à un maillon deviendra nul, alors, selon la valeur de I, le processus désigné dans le maillon est réveillé ou le compteur de dépendance de la macro instruction MI est décrémenté.

Enfin, pour réaliser les opérations de sélection de rendez-vous, une table de sélection (TS) est construite afin de faciliter cette opération complexe. Chaque entrée décrit la sémantique d'une alternative relative à la communication, et contiendra une adresse de mot d'état d'une macro instruction, @MI. L'indicateur I est une information générale à l'ensemble de la table caractérisant la décomposition macroscopique de l'instruction "select".

Lorsque la décomposition macroscopique prend en compte les mécanismes de rendez-vous, les mécanismes de blocage et de réveil des processus sont réalisés grâce au compteur de dépendance de chaque macro instruction, et non par des opérations P et V sur des sémaphores privés.

La valeur positive d'un compteur de dépendance "bloque" une macro instruction et son passage à 0 la rend exécutable.

Le réveil ou non d'un processus, ou la décrémentation ou non du compteur de dépendance d'une macro instruction lors d'un rendez-vous est déterminé par la même analyse des variables d'état associées au rendez-vous et contenues dans le BCP des différentes tâches.

Les indicateurs de décomposition macroscopique I font partie de ce type de variables manipulées en exclusion mutuelle et précisent quelles activations il faut réaliser.

Les variables I permettent de choisir les traitements de contrôle adaptés à la décomposition de chaque tâche.

Par exemple la réalisation d'une instruction "Accept E" s'effectue de la manière suivante :

```
P (mutex) ;                -- mutex est un sémaphore de mutuelle
                              exclusion

Modification des variables d'état du rendez-vous ;

if I 1 then                -- le traitement de rendez-vous

  if blocage then          -- est contenu dans une macro

         mémoriser   MI1 dans la          -- instruction MI1

         tête de queue de l'entrée E ;

 else

         (MI 1) := (MI 1)-2 ;

  endif ;

  V(mutex) ;

else .
```

```
if non blocage then
        V (sémaphore de rendez-vous) ;   -- le processus
    endif ;                              -- ne doit pas
    V (mutex) ;                          -- se bloquer en
    P (sémaphore de rendez-vous) ;       -- section critique
endif ;
```

La structure mise en place est souple et permet d'envisager différentes décompositions macroscopiques (cf §3.2.1.c). Ainsi, par exemple, au lieu de mémoriser une adresse @MI dans chacune des différentes structures de données précédentes, plusieurs adresses pourraient être stockées, leur nombre étant caractérisées par un compteur.

5.3. MISE EN OEUVRE DES EXCEPTIONS

Deux mécanismes de prises en compte des exceptions dans un programme parallélisé ont été proposés au paragraphe 3.2.3. La mise en oeuvre de ces mécanismes sur MLI, est présentée dans cette section. Les objectifs visés sont de montrer qu'elle est relativement simple et n'entraine pas de profondes modifications du système de base. Les mécanismes ont été conçus afin de pénaliser le moins possible, en temps d'exécution, un programme où aucune exception n'est levée. Bien entendu, des solutions permettant l'exploitation d'un parallélisme plus important peuvent être envisagées, mais elles engendrent des mises en oeuvres plus complexes modifiant de façon importante le système de base MLI.

La mise en oeuvre des mécanismes d'exception dans un programme parallélisé par décomposition macroscopique est simplifiée du fait du mode de séquencement choisi, (voir 3.2.3.), qui implique que pour une tâche, seules les unités associées à un même traitement d'exception sont exécutables simultanément. Ainsi lorsqu'une exception est levée dans une tâche, tous les processus en exécution ou exécutables pour cette tâche sont touchés par cette exception.

Les deux solutions proposées utilisent le BCP comme moyen de communication des processus actifs pour une tâche. Ainsi, lorsqu'une exception est levée dans une macro instruction en exécution, la présence et les caractéristiques de celle ci sont mémorisées dans une zone particulière du BCP accessible en mutuelle exclusion.

Le BCP de chaque tâche contient les différents champs :

BCP
PRESENCE EXCEPTION
VALEUR EXCEPTION
ADRESSE TRAITEMENT EXCEPTION
ID - INSTRUCTION

PRESENCE-EXCEPTION : indique si une exception est à traiter

VALEUR-EXCEPTION : indique la valeur de l'EXCEPTION à traiter

ADRESSE TRAITEMENT EXCEPTION : ce champ est rempli lorsque la tâche devient activée. Il est utilisé lorsqu'une exception se produit dans le corps de la tâche.

ID-INSTRUCTION : indique la position de l'instruction où s'est produite une exception.

A chaque macro-instruction le système associe un bloc de contrôle contenant des informations caractérisant son état, ses ressources, ses liens avec d'autres processus. Cette table système est appelée Bloc de Contrôle de Macro Instruction

(BC MI). Elle contient en particulier :

BCMI
BCP - TACHE
MOT-D-ETAT
ETAT
PROCESSEUR

BCP-TACHE : permet d'établir le lien entre la macro-instruction et le BCP de la tâche propriétaire

MOT-D-ETAT : adresse du mot d'état de la macro instruction

ETAT : indique l'état de la macro instruction : exécutable, en exécution, suspendu.

PROCESSEUR : indique un identificateur de processeur d'exécution.

Comme dans MLI un descripteur de traitement d'exception (DTE) est associé à chaque unité (sous programme, bloc s'il est matérialisé...) et est placé devant la première instruction de celle-ci.

Chaque DTE à la structure suivante :

DEBUT	@ TE	@ MIE

DEBUT : contient la position de la première instruction générée pour la partie "BEGIN" de l'unité.

@ TE : indique la position du traitement d'exception de l'unité, lorsqu'il existe, dans la macro instruction qu'il le contient, Ø sinon.

@MIE : contient l'adresse de la macro instruction qui contient le traitement d'exception de l'unité, lorsqu'il existe, Ø sinon.

Les mécanismes de gestion (création, destruction) des accès aux différents DTE font que l'ensemble des DTE des unités en exécution apparait être géré comme une pile, comme les appels et de retours des sous programmes. Ce mécanisme est mis en oeuvre dans MLI grâce à certaines ressources permettant de déterminer le niveau lexical de référence, le niveau lexical courant, le niveau lexical de référence de chaque sous programme appelant et leur niveau lexical courant dans le niveau lexical de référence.

Ces mécanismes sont détaillés dans (RD 81b) et ne seront pas rappelés ici. Il suffit de retenir que les appels et retours des unités, et par conséquent les accès aux DTE(s), sont gérés comme dans une pile. Par la suite, nous utiliserons, par abus de langage, les termes de "dépiler" et "empiler" en ce qui concerne la gestion des appels et retours d'unité et de leur DTE.

Puique les appels et retours de sous programmes s'effectuent suivant l'ordre canonique et que seules les unités d'une même "unité logique exception" (ULE) sont exécutées simultanément, lorsqu'une exception est levée, elle correspond à la dernière unité logique exception appelée. Tous les sous programmes appelés dans cette unité doivent être désactivés. L'ensemble de ces sous programmes correspond aux appels et retours mémorisés entre le sommet de pile des appels et le dernier appel d'un sous programme contenant une partie traitement des exceptions. Pour distinguer les sous programmes contenant une partie traitement des exceptions de ceux qui n'en contiennent pas, il suffit d'examiner le champ @ TE de leur DTE respectif.

Pour arrêter les processus en cours d'exécution deux méthodes ont été proposées au chapitre 3.2.3. Pour chacune de ces solutions deux cas sont à envisager :

a- la macro instruction où s'est produite l'exception ne comporte pas de traitement d'exception.

b - la macro instruction où s'est produite l'exception contient la partie traitement des exceptions.

* <u>Solution 1</u>

Dans cette solution lorsqu'une exception est levée dans une tâche alors un appel système est effectué est celui-ci bloque l'ensemble des processus actifs de la tâche. Le processeur où l'exception a été levée a la charge de réaliser le traitement des exceptions (voir § 3.2.3).

Le système, ayant bloqué les processus actifs de la tâche, va déterminer dans quelle macro-instruction se trouve la partie traitement des exceptions de l'unité logique exception suspendue. Pour se faire, il va "dépiler" les unités se trouvant en sommet de "pile" jusqu'à l'unité contenant le traitement des exceptions de l'ULE. Cette unité se caractérise par son DTE dont les champs TE et MIE sont non nuls.

Le traitement suivant est effectué :

Accès au DTE de la dernière unité appelée

<u>tant que</u> TE = Ø <u>faire</u>

Retour ;

Accès au DTE de l'unité appelante ;

<u>fait</u> ;

L'instruction RETOUR est décrite dans (RD 81b), elle permet de supprimer les informations du sous programme appelé et de restituer les accès aux informations du sous programme appelant.

Le dernier DTE lu est de la forme :

BEGIN	@ TE	@ MIE

où @ MIE et @TE ont des valeurs non nulles. Ils indiquent l'adresse de la macro instruction où se trouve le traitement des exceptions et l'adresse de ce traitement à l'intérieur de celle-ci.

Si MIE est la macro instruction où s'est produite l'exception alors le processeur qui l'exécute se branche à l'adresse @TE, pour exécuter le traitement des exceptions.

Si MIE n'est pas la macro instruction où s'est produite l'exception alors MIE est activée et son exécution commence à l'adresse @TE.

Lorsque l'exception a été levée l'identificateur de l'instruction touchée a été rangée dans le champ ID-INSTRUCTION du BCF. Lors de l'exécution du traitement des exceptions, la valeur de ce champ est comparée à la valeur du champ BEGIN du dernier DTE lu afin de déterminer si l'exception a été levée dans la partie déclarative de l'unité contenant la partie traitement des exceptions. Si tel est le cas, il y a propagation de l'exception à l'unité logique exception appelante. Sinon l'exception est traitée par l'alternative qui lui correspond dans le traitement des exceptions. Si celle-ci n'existe pas, l'exception est propagée à l'ULE appelante.

La propagation à l'unité appelante se fait en appliquant à nouveau la méthode qui vient d'être décrite, c'est à dire en exécutant des retours successifs d'unité jusqu'à ce que la valeur du champ @TE, du DTE associé, soit non nulle.

Lorsque l'unité dans laquelle on revient est le corps d'une tâche (identifié par le fait que l'adresse de retour est nulle) les adresses @TE et @MIE se trouve associées au champ ADRESSE TRAITEMENT EXCEPTION. Si ce champ est nul la propagation est arrêtée, sauf si la tâche est en rendez-vous, auquel cas l'erreur est propagée chez l'appelant.

* Solution 2

Pour cette solution, mis à part le principe d'arrêt des unités touchées par la levée d'une exception (exposé dans la section 3.2.3), les principes utilisés sont semblables à ceux de la première solution.

De la même façon, un descripeur DTE est associé à chaque unité. Lorsqu'une exception est levée, le système recherche, par retour successif, le DTE de l'unité contenant la partie traitement des exceptions de la dernière ULE appelée, et mémorise les adresses @TE et @MIE dans un champ réservé du BCF de la tâche.

Lorsque l'exécution de la macro instruction MI doit être suspendue, parce qu'elle lève une exception où parce que la valeur du champ PRESENCE-EXCEPTION de son mot d'état est vrai, alors, si @MI = @MIE, la partie traitement des exceptions, située en @TE, est exécutée. Sinon, l'adresse de la partie traitement de contrôle de la macro instruction est lue dans un champ réservée de son mot d'état et est exécutée.

5.4. Sur la taille des processus parallélisé

Il est maintenant reconnu qu'un parallélisme efficace prend en compte les contraintes, d'une part matérielles et d'autre part logicielles, auxquelles est soumise son exploitation. Ainsi les études sur les systèmes dirigés par les données tendent maintenant à exploi-
ter le parallélisme à des niveaux plus élevés que celui de l'instruction, et réalisent des regroupements d'instructions (cf MAUD et (KM 79). Tout au long de ce travail nous avons souligné l'importance de la "granularité" du parallélisme qui vise à équilibrer la taille des processus exécutables en parallèle par rapport aux traitements de contrôle que leur exploitation parallèle nécessite. Dans le modèle Data Flow Macroscopique la partie traitement effectif a été distinguée de la partie traitement de contrôle qui permet l'enchainement des processus. Il a été montré que ces traitements de contrôle sont relativement simples et permettent d'exprimer de nombreux cas de parallélisme inhérent aux programmes.

Cependant, comme cela a pu apparaitre, dans ce chapitre sur la mise en oeuvre du Data Flow Macroscopique sur le système MLI, les traitements de contrôle, définis ci-dessus, ne sont pas les seuls "contrôles" (au sens général) nécessaires à l'exploitation de ce parallélisme.

Ainsi, de nombreux problèmes systèmes, relatifs à la ges-
processus exploités en parallèle, sont à résoudre, en particulier :

- la distribution des tâches qui doit prendre en compte de nombreux critères de choix, comme la priorité ou la gestion des ressources ;

- la gestion des tables système nécessaires à l'exécution parallèle de processus qui engendre de nombreuses mises à jour, consultations, récupérations ...

De tels contrôles sont propres à l'exploitation du parallélisme et leur importance fait qu'il doit en être tenu compte dans le choix du parallélisme exploité.

Un rapide raisonnement montre l'importance de ces contraintes logicielles dans l'efficacité du parallélisme exploité :

- soit un programme séquentiel, décomposable en M macro instructions (ou processus), exécuté séquentiellement en un temps $Ts = \sum_{i=1}^{n} TEi$, où TEi est le temps d'exécution sur un mono processeur du traitement effectif de chaque macro instruction composant ce programme.

Pour simplifier supposons que ces traitements effectifs sont exécutés en un temps constant TE, alors Ts = M*TE.

Supposons, de plus, que :

- N est le nombre de processeurs d'exécution

- TC est le temps de contrôle nécessaire à l'exploitation du parallélisme

- TR est le temps supplémentaire pour l'exécution du traitement effectif d'une macro instruction sur une machine multi-processeurs. (TR comprend principalement les temps de traversée du réseau et de résolution de conflit d'accès mémoire)

- T// est le temps d'exécution parallèle d'un programme avec les hypothèses précédentes.

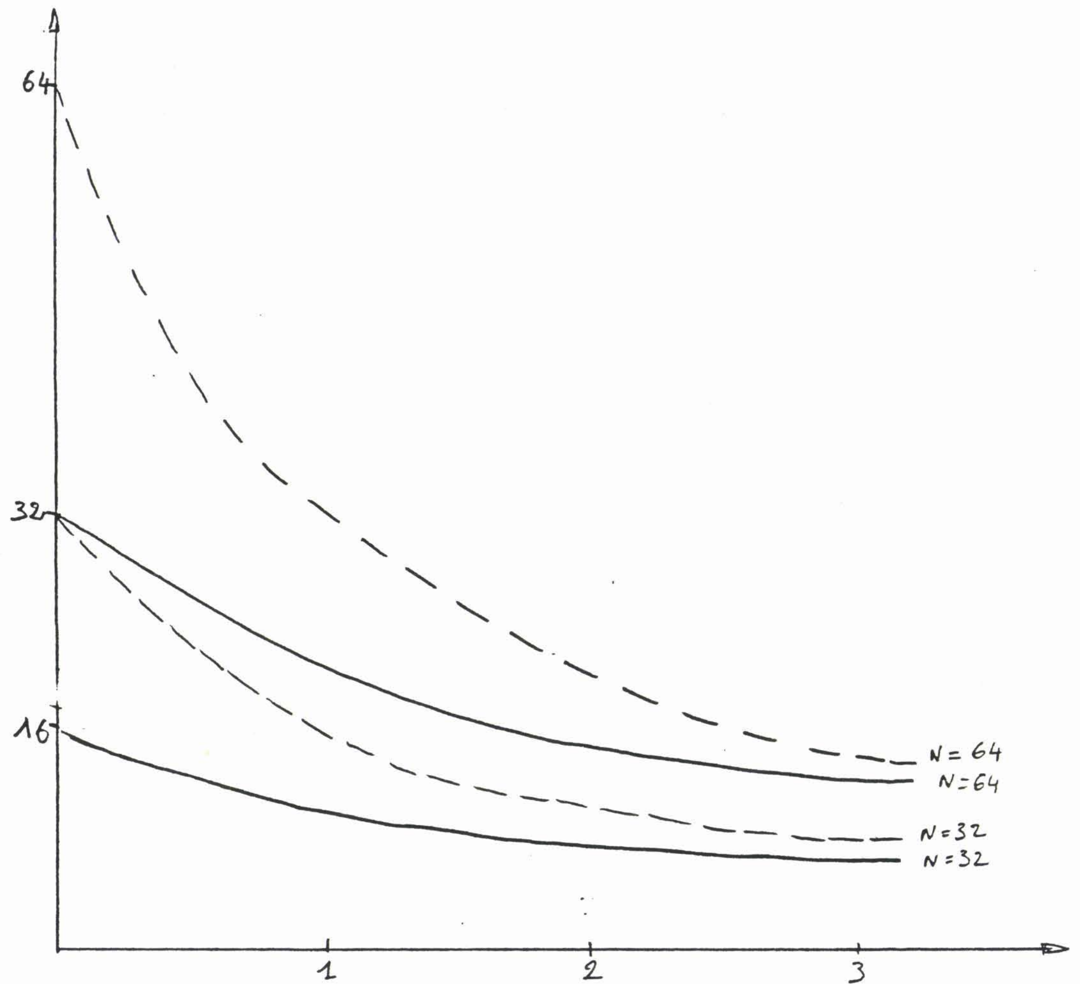
alors : $T// = M/N * (TE + TR + TC)$

ceci permet de déduire le gain :

$$(1) \quad \frac{Ts}{T//} \doteq \frac{N}{1 + \frac{TR}{TE} + \frac{TC}{TE}}$$

Cette expression montre l'importance de la taille des traitements effectifs par rapport aux contraintes matérielles, exprimées par TR, et logicielles, exprimées par le temps TC propre à chaque processus et nécessaire à leur exploitation parallèle.

Les courbes suivantes montrent les variations du gain Ts/T// en fonction de TC/TE pour certaines valeurs de N et de TR/TE supposées constantes.



—— $\frac{TR}{TE} = 1$

--- $\frac{TR}{TE} = 0$

Ces courbes permettent de montrer que pour TC/TE < 1 le gain Ts/T// décroit rapidement. Aussi, pour obtenir une dégradation des performances qui soit raisonnable, il est nécessaire que TC/TE soit inférieur à 1. Cela peut être réalisé soit en diminuant TC soit en augmentant TE.

L'expression (1) permet de déduire la taille minimum du traitement effectif de chaque macro instruction en fonction de TR, TC et de l'accélération Ts/T//.

$$(2) \qquad TE = \frac{TR + TC}{N * \frac{T//}{Ts} - 1}$$

Il faut noter que selon les estimations de Robert Behnard (Ber 82) qui s'appuie sur les travaux de nombreux chercheurs, l'accélération dûe à l'exploitation du parallélisme se situe entre Ø,3 N et N/log2 N.

Bien entendu le raisonnement ci-dessus n'est qu'une première approche du problème visant à montrer l'importance de la "granularité" du paralléllisme. L'estimation de la taille minimum des processus, pour une exploitation raisonnable du parallélisme, nécessite une étude plus approfondie de la structure système et matérielle de la machine.

CHAPITRE 6

C O N C L U S I O N

Cette thèse a présenté une étude d'une architecture dirigée par les données appliquée au langage Ada.

Pour atteindre nos objectifs nous avons d'abord effectué une étude détaillée des architectures multimicroprocesseurs, des réseaux d'interconnexions, d'une machine langage intermédiaire (MLI) et du langage Ada. Nous avons ainsi acquis une connaissance globale sur les architectures de multi microprocesseurs, sur les problèmes de parallélisme et les problèmes de langage. Cette étape nécessaire à la poursuite de nos travaux, a été particulièrement enrichissante étant donné la diversité des domaines abordés liés à l'architecture, au système, au langage et à la conception des ordinateurs. Les chapitre I et 4, en particulier, sont le reflet de ces études.

Ainsi, à l'examen des systèmes dirigées par les données, constatant qu'un parallélisme au niveau de l'instruction est difficile à exploiter, nous avons rappeler les mécanismes d'une technique d'exploitation d'un parallélisme plus global étendu à un groupe d'instructions, appelée "Data Flow Macroscopique". Cette technique permet aussi bien l'exécution parallèle des termes, des expressions arithmétiques ou booléennes, que les instructions du programme isolées ou regroupées, ou encore des tâches au sens des systèmes actuels. Grâce aux possibilités connues de transformations automatiques de programme par l'analyse des données lues et modifiées, ce modèle permet d'exprimer, de façon transparente à l'utilisateur, un parallélisme simple et efficace pouvant être adapté aux contraintes logicielles et matérielles que son exploitation engendre.

La puissance du modéle induit peu de contrainte sur les formes de parallélisme exploitables ainsi nous avons rappelé ses applications aux structures classiques des langages de programmation : instructions de choix, boucles, sous-programmes.

Le Data Flow Macroscopique est également apparu bien adapté à la prise en compte des mécanismes plus complexes, support du temps réel ou permettant la récupération d'exceptions dans un programme, implantés dans le langage Ada. Ainsi après avoir examiné les problèmes de sémantique que l'exploitation du parallélisme dans les tâches Ada peut engendrer, nous avons montré comment les mécanismes de rendez-vous (accept, appel, appel conditionnel appel avec attente limitée, select) peuvent être pris en compte par des décompositions macroscopiques adaptées permettant l'expression de nombreux cas de parallélisme. En particulier, nous avons montré comment, en fonction de la complexité et du parallélisme inhérent de chaque tâche, le parallélisme dégagé peut être choisi de façon à obtenir la meilleure efficacité.

Puis les problèmes relatifs aux traitements des exceptions ont été étudiés avec le soucis constant de présenter des solutions simples minimisant les contrôles systèmes. Ces solutions peuvent être plus complexes dans la mesure où les besoins d'expressions du parallélisme justifient ce surcroit de complexité système.

En s'appuyant sur les études d'une machine langage intermédiaire nous avons montré comment une mise en oeuvre du Data Flow Macroscopique pouvait être réalisée sur cette machine sans modification fondamentale du système. Enfin une étude d'éléments d'architecture pour une machine multi microprocesseurs adaptée au système Data Flow Macroscopique a été présentée au chapitre 4.

De nombreux prolongements à ce travail sont à envisager.

Il serait sans doute nécessaire de formaliser le modèle Data Flow Macroscopique afin d'en donner une expression rigoureuse qui permettrait de vérifier les décompositions macroscopiques et impliquerait une certaine cohérence des découpages. Cette formalisation est nécessaire à la maîtrise du modèle et à son évolution.

Un domaine particulièrement intéressant à étudier est l'application du modèle à la parallélisation de langages propres à l'intelligence artificielle. Ainsi, il semble que le Data Flow Macroscopique puisse être bien adapté à l'expression du parallélisme dans Prolog.

Enfin, de nombreuses études systèmes sont à réaliser, en particulier en ce qui concerne la distribution des tâches. Ces études devraient conduire à des évaluations de performances et plus précisément à des évaluations de la taille minimum du traitement effectif de chaque macro-instruction permettant d'obtenir le parallélisme le plus efficace.

B I B L I O G R A P H I E

(Ack 79) ACKERMAN, W. B., and DENNIS, J. B. "VAL-A value oriente algorithmic Language preliminary reference manual," Tech Rep. TR-218, Lab. for Computer Science, Massachusetts Institute of Technology, June 1979

(AG 82) B. ARDEN, R. GINOSAR, MP/C A Multiprocessor/ Computer Architecture, IEEE Trans. on Comp Vol C31 n° 5 p 455 - 473, Mai 1982

(Arv 80) ARVIND, KATHAIL, V. and PINGALI. K. "A processing element for a large multiprocessor dataflow machine", in Proc. Int. Conf. Circuits and Computers (New York, Oct 1980), IEEE New York 1980

(Arv 78) ARVIND, GOSTELOW, K.P. and PLOUFFE W. "An asynchronus programming language and computing machine". Tech. Rep 114a. Dep Information and Computer Science, Univ. Of California Irvine Dec 1978

(Bar 68) G. BARNES et al, The ILLIAC IV Computer, IEEE Trans. on Comp., Vol C17, n° 8, pp 746-757, Aout 1986

(Bar 81) G. BARNES, S. LUNDSTROM, Design and Validation of a Connection Network for Many-processor Multiprocessor Systems, Computer, December 1981

(Bat 74) K. BATCHER, Staram Parallel Processor System Hardware, Proc. AFIPS 1984 Nat. Comp. Conf., Vol 43, pp 405-410, Mai 1974

(Bat 76) K. BATCHER, The flip network in STARAN, Proc. 1976, Int'l Conf. Parallel Processing, p65-71, August 1976

(Bat 80) K. BATCHER, Design of a Massively Parallel Processor, IEEE Trans. on Comp., Vol C29, n°9, pp 836-840, August 1980

(BDH 81) F. BRIGGS, M. DUBOIS, K. HWANG, Throughput Analysis and Configuration Design of a Shared-ressource Multiprocessor System : PUMPS, Proc. of the 8th Ann. Symp. on Comp. Arch., pp 67-79, Los Angeles, Mai 1981

(Ben 65) V.E. BENES, Mathematical theorie of connecting networks and telephone traffic, New York academic Press, 1965

(Ber 66) A.J. BERNSTEIN, Analysis of programs for parallel processing.
IEEE Transaction on electronic computers, vol EC-15, n° 5, October 1966

(Ber 82) R. BERNHARD, Computing at the speed limit IEEE Spectrum, Juin 1982

(Bra 83) A. BRADIER, Mécanismes matériels réalisant à l'exécution l'optimisation des programmes, thèse de 3ème cycle, Lille, Septembre 1983

(Brh 78) BRINCH HANSEN. P., Distributed Processes : a concurrent programming concept, C.A.C.M., Novembre 1978

(Bry 83) J. BRYGIER, Etudes d'éléments d'architecture pour une machine langage Ada, thèse de 3ème cycle, Lille, Septembre 1983

(Bur 82) R. BUEHRER et al, The ETH-Multiprocessor EMPRESS A. Dynamically Configurable MIMD System, IEEE Trans. on Comp., Vol C31, n°11, Novembre 1982

(CK 81) J.S. CONERY, D.F. KIBLER, Parallel Interpretation of Logic Programs. Proceedings of the 1981 Conference on Functionnal Programming and Computer Architecture, 18-22 Oct 1981, p163-170

(CON 63) M. CONWAY, A Multiprocessor System Design Proceedings AFIPS FJCC, 1963

(Com 76) COMTE, D., DURRIEU, A., GELLY, O., PLAS, A., and SYRE,J.C., "TEAU 9/7 : SYSTEME LAU-Summary in English" CERT Tech Rep 1/3059, Centre d'Etudes et de Recherches de Toulouse, Oct. 1976

(Cor 78) V. CORDONNIER, "Architecture pipeline" Support du cours IRIA sur le Traitement Parallèle - Lille, 29-2 Juin 1978

(CRS 84) J.C. COTTET, C. RENVOISE, D. SCIAMMA, Vectorisation automatique et paramétrée de programmes. Proceeding sur le 6ème Colloque International sur la Programmation, Toulouse, Avril 1984

(DAVI 78) DAVIS,A.L., "The architecture and system method of DDM1 : A recursively structured data driven machine", in Proc 5th Annu. Symp. Computer Architecture (Palo Alto, Calif., Apr 3-5), ACM, New York, 1978, pp 210-215

(DDB 81) P. DENNING, T. DENNIS, J. BRUMFIELD, Low Contention Semaphores and Ready Lists, Comm. of the ACM, Vol 24, n° 10, pp 687-699, Octobre 1981

(Dij 75) DIJKSTRA, E.W., Guarded commands, non-determinacy and a calculus for the derivation of programs. Com of the ACM, vol n°18, Aug 1975

(DJ 81a) D. DIAS et J.R. JUMP, Packet switching interconnection networks for modular systems, Computer, p43-53, Décembre 1981

(DJ 81b) D. DIAS et J.R. JUMP, Analysis and Simulation of buffered Delta Networks, IEEE Trans. on Comp., Vol C30, n°4, p273-282, Avril 1981

(Den 74) DENNIS,J.B. "First version of a data flow procedure language" in Programming Symp. Proc. Colloque sur la Programmation (Paris, France, Apr 1974)
B. ROBINET, Ed., Lecture notes in computer science, vol 19, Springer-Verlag, New York, 1974, pp 362-376

(Den 79) DENNIS,J.B., LEUNG, C.K.C., and MISUNAS, D.P. "A highly parallel processor using a data flow machine language" Tech. Rep. CSG Memo 134-1, Lab for Computer Science. Massachusetts Institute of Technology, June 1979

(DR 83) P. DOUSPIS et D.ROCACHER, Etude bibliographique des multiprocesseurs MIMD/MSIMD/Data Flow, Note interne Centre de recherches BULL, Août 1983

(ELT 82) J. ERHEL, A. LICHNEWSKY, F.THOMASSET, Multiprocessor INRIA : Structure et Fonctionnement, Rapports techniques n°14, Juillet 1982

(Erh 82) Jocelyne ERHEL, Parallélisation d'algorithmes numériques, thèse de 3ème cycle, Mars 1982

(Fen 74) T.Y. FENG, Data manupulating functions in parallel processors and their implentation, IEEE Trans. on Comp., vol C23, p309-318, Mars 1974

(Fen 81) T.Y. FENG, A survery of interconnection networks, Computer, p12-27, Décembre 1981

(Fly 72) FLYNN M., Some computer organizations and their effectiveness, IEEE Trans. on Computer, vol C21, pp 948-960, 1972

(Fr 81) M.A. FRANKIN, VLSI performance comparaison of Banyan and Crossbar communication networks, IEEE Trans. on Compu., vol C30, n°4, april 1981

(Ful 78) S. FULLER et al, Multi Microprocessors : An overview and working example, Proc of the IEEE, vol 66, n°2, Février 1978

(GL 73) L.R. GOKE et J. LIPOVSKI, Banyan networks for partionning multiprocessing system, Proc : first Annual comp arch. conf., p 21-28, décembre 1973

(GK 83) D. CJAJSKI, D. KUCK et al, CEDAR A Large Scale Multiprocessor, Computer Architecture News, Mars 1983

(Got 83) A. GOTTLIEB et al, the NYU Ultra-computer - Designing an MIMD Shered Memory Parallel Computer, IEEE Trans. on Comp., vol C32 n°2, Février 1983

(Gou 82) Michel GOUGET, Etude et réalisation d'un réseau de permutation OMEGA-BONES et de sa commande, Thèse de Docteur Ingénieur à l'Université de Nice, Juillet 1982

(GS 82) A. GOTTLIEB, J.SCHWARTZ, Networks and algorithm for Very Large Scale Parallel Computation, computer, Janvier 1982

(JB 82) R.JENEVEIN, J.BROWNE, A control processor for a reconfigurable array computer, Proc of the 9th Ann. Symp. on Comp. Arch., 1982

(JDL 81) R.JENEVEIN, D.DEGROOT, G.LIPOVSKI, A hardware support mechanism for scheduling ressources in a parallel machine environment, Proc of the 8th Ann. Symp. on Comp. Arch., Los Angeles, Mai 1981

(Joh 79) JOHNSON D. et al, "Automatic partitioning of programs in multiprocessor systems", in Proc IEEE COMPCON 80 (Feb 1980), IEEE, New York, pp 175-178

(Jon 82) Anita JONES, Anders ACDO, Comparative Efficiency of Different Implementations of the Ada Rendez-vous ADATIC, DCM 1982, p 212-223

(Kku 79) R.KOBER, C.KUZNIA, SMS 201 - A Powerfull Parallel Processor with 128 Microprocessors, Euromicro Journal, 5, pp 48-52, 1979

(KM 79) KOMPS E. et MUCHNICK S., Three extensions to LAU and its hardware architecture, 1st European Conference an Parallel and Distributed Processing, 14-16 Fev 1979

(KS 82) D.KUCK, R.STOKES, The Burroughs Scientifc Processor (BSP). IEEE Trans. on Comp., vol C31, n°5, pp 363-376, Mai 1982

(KS1 81) J.KUEHN, H.SIEGEL, Simulation studies of FASM in SIMD mode, IEEE Comp. Soc. Workshop on Comp. Arch. for Pattern Analysis, Hot Springo, Novembre 1981

(LB 80) S.LUNDSTROM, G.BARNES, a controllable MIMD architecture, Proc of the 8th Int. Conf. on Parallel Processing, pp 19-27, Columbus, Août 1980

(Lec 79) LECOUFFE M.P., Etude et définition d'un modéle de machine parallèle dirigée par les données, thèse de 3ème cycle, Juillet 1979

(Mac 79) M.MACKAUS et al, Experimental polyprocessor system (EPOS) Architecture, Proc of the 6th Ann. Symp. on Comp. Arch., pp 188-195, Philadelphie, 1979

(MD 81) R.MANNER, B.DELUIGI, The Heidelberg Polyp - A flexible and fault-tolerant poly-processor, Computer Physico Communications, 22, pp 279-284, 1981

(MM 81) B.A. MAKRUCK, T.N. MUDGE, Probabilistic of a crossbar switch, SEL report n°150, department of electrical and computer engineering, University of Michigan, Janvier 1981

(Mot 83) Tohru Moto-oka, Overview to the Fifth Generation Computer System Project, 1983 ACM 0149-7111

(MS 82) R.J. Mc MILLEN, H.J. SIEGEL, Performance and fault tolerance improvements in the inverse data manipulator network, Proc of th e 9th Annu. Symp. on Comp. Arch., p63-72, 1982

(Müh 81) K.MUHLEMAN, Tavards an Implementation of ADA Rendez-vous synchronization - Euromicro 81, p289-301

(Nel 80) V.NELSON, Fault Tolerance in Reconfigurable Multiprocessor Systems, Computer Softward and Applic. Conf., Chicago, Octobre 1980

(Nuk 84) NUKIYAMA T at al, A VLSI Image Pipeline Processor, Fevrier 1984, IEEE ISCS

(Pa 81) J.H. PATEL, Peformance of Processor-memory interconnections for multiprocessors, IEEE Trans. Comp. vol C30 n°10, Oct 1981

(Pea 77) M.C. PEASE, The indirect binary n-cube microprocessor array, IEEE Trans. on Comp. vol C26, n° 5, pp 548-573, Mai 1977

(Pet 81) B. PETITPREZ, Etude et réalisation d'un système multiprocesseur à pilotage par les données, Thèse Docteur Ingénieur, Université de Lille, Janvier 1981

(PR 82) D.S.PARKER et C.S. RAGHAVENDRA, The gamma network : a multiprocessor interconnection network with redundant paths, Proc 9th Annual Symp. on Comp. Arch., Austin, Avril 1982

(RDL 82a) RENVOISE C., DOUSPIS P., LEGOUX M., BRADIER A., BRYGIER J., "Machine Langage ADA - Eléments d'Architecture, rapport de Recherche CII Honeywell Bull, Oct 1982

(RDL 82b) RENVOISE C., DOUSPIS P., LEGOUX M., BRADIER A., BRYGIER J., "Machine Langage ADA - Eléments de Spécification de l'Architecture Matérielle" - Rapport de Recherche CII Honeywell Bull, Déc. 1982

(RDo 81a) RENVOISE C., DOUSPIS P., "Etude d'un schéma d'Adressage Protégé Adapté à l'Exécution de Programmes ADA" - Rapport de Recherche CII Honeywell Bull, Juin 1981

(DRo 81b) RENVOISE C., DOUSPIS P., "Machine Langage ADA - Etude des Instructions Machine Adaptées" - Rapport de Recherche CII Honeywelle Bull, Déc 1981

(Ren 80) RENVOISE C., ROUQUIE G., COTTET J.C., FEAUTRIER P., Détection automatique d'opérations parallèles et vectorielles dans les programmes, rapport de recherche CII-HB, Novembre 1980

(Ren 82) RENVOISE C., Data Flow Macroscopique, Présentation générale, rapport interne, CII-HB, Juillet 1982

(Ren 83) RENVOISE C., AMOS : Architecture Multi-opérateurs Séquentiels, Note interne, Janvier 1983

(Rie 81) C. RIEGER, ZMOB : Doing it in parallel !, IEEE Comp. soc. workshop on comp. arch. for pattern analysis, Hot springs, Fevrier 1981

(Roc 83) D. ROCACHER, Etude Bibliographique des réseaux d'interconnexion, Juillet 1983, Rapport interne centre de recherche BULL

(Rus 78) R. RUSSEL, The CRAY-1 Computer System, Comm. of the ACM, vol 21, pp 63-72, Janvier 1978

(Ru 77) J. RUMBAUGH, A dataflow multiprocessor, IEEE Trans. on Comp., vol C26, n°2, Février 1977

(SBK 77) H. SULLIVAN, T.BASHKOW, D.KLAPPHOLZ, A large scale, Homogeneous, fully distributed parallel machine, II, Proc of the 4th Ann. Symp. on Comp. Arch., 1977

(Sej 80) M.SEJNOWOKI et al, An overview of the Texas Reconfigurable Array Computer, AFIPS Conf. Proc. 49, 1980

(Sha 78) H. SHAPIRO - Theorical limitations on the efficient Use of parallel memories IEEE Trans. on Comp., vol C27, n°5, Mai 1978

(Sie 81) H. SIEGEL et al, PASM : A partitionable SIMD/MIMD system for image processing and pattern recognition, IEEE Trans. on Comp., vol C30, n°12, Décembre 1981

(SLS 82) K.SHIN, Y.L. LEE, J.SASIDHAR, Design of HM2p - A hierarchical Multimicroprocessor for general purpose applications, IEEE Trans. on Comp., vol C31, n°11, Novembre 1982

(SN 80) D.SATISHCHANDRA, V.NELSON, A reconfigurable distributed digital filter, Proc. Distributed Data Acquisition, Computing and Control Symp, Miami, Décembre 1980

(Sni 80) Marc SMIR, Crossbar 1616, a neophyte incursion into VLSI design, Université Einburgh

(Sny 82) L. SNYDER, Introduction to the configurable, highly parallel computer, Computer, Janvier 1982, pp47-50

(Swa 78) R.SWAN, The switching structure and adressing architecture of an extensible multiprocessor : CM*, PhD Thesis, Carnegie Mellon University, Pittsburg, 1978

(SYRE 77) SYRE, J.C., COMTE D. and HIFDI N.N "Pipeling parallelism and asynchronism in the LAU system" in Proc. 1977 Int Conf Parallel Processing (AUg 1977) pp 87-92

(Tes 68) TESLER L.G. et ENEA H.J., A language design for concurrent processe" - Proceedings SJCC 1968

(Tim 84) C.TIMSIT, ISIS : Machine pour le calcul scientifique de haute performance - A paraître Bulletin de liaison de l'INRIA

(TL 79) A.R.TRIPATHI, G.J. LIPOVSKI, Packet switching in Banyan, Proc of the 6th Ann. Symp. on Comp. Arch., Philadelphie 79, pp 160-167

(Tou 79) B.TOURSEL, Contribution à l'étude de l'architecture des ordinateurs : proposition pour un nouveau mode de fonctionnement. Thèse de docteur es sciences mathématique présentée à Lille le 26 Septembre 1979

(Tr 78) P.TRELEAVEN et al, The design of highly concurrent computingg system, University of Newcastle, Tech. Rep., Juillet 1978

(USG 83) United States Department of Defense, Reference Manual for the Ada Programming Language Proposed Standard Document, January 1983

(UTL 81) L.UHR, M.THOMPOON, J.LACKEY, A 2- leyered SIMD/MIMD parallel pyramidal "ARRAY/NET", IEEE Comp. Soc. Workshop on Comp. Arch. for Pattern Analysis, Hot Springs, Novembre 1981

(Vas 82) VAST user's guide Pacifique Sierra Research Cop, June 1982

(WATS 79) WATSON I., and GURD J. "A prototype data flow computer with token labeling" in Proc Nat. Computer Conf. (New York, N.Y. June 4-7) vol 48, AFIPS Press, Arlington, Va, 1979, pp 623-628

(WB 72) W.WULF, C.BELL, C.m.m.p. - A multiminiprocessor, Proc AFIPS FJCC, 1972

(WF 80a) CLWu et T.Y. FENG, on a distributed processor communication architecture, Proc Compcar fall 1980 p 599-605

(WF 80b) CLWU et T.Y. FENG, on a class of multistage interconnection networks, IEEE Trans. on Computer vol C29, n°8, pp 694-702, Août 1980

(Wil 82) K.WILSON, A program of computing support for scientific research, Conference donnée à l'ENS, Décembre 1982

(Wij 69) WIJNGAARDEN, Report on the Algorithmic Language ALGOL 68, Springer-Verlay, Numeriche Mathematik, 14

(Wir 71) WIRTH N., The programming language Pascal, Acta Informatica, 1971

(Wle 82) R.NEATHERLY, J.LEATHRUM, Efficient semaphore management using read/modify/write memory cycles, ACM Operating System Review, Vol 16, n°1, Janvier 1982

(Wli 81) J.WELSH, A.LISTER, A comparative study of task communication in Ada, Soft. Pract. and Exp., vol 11, pp 257-290, 1981

RESUME

Les expériences des systèmes dirigés par les données ont montré qu'il est difficile d'exploiter le parallélisme au niveau de l'instruction. Un parallélisme plus global étendu à un groupe de plusieurs instructions, appelé "Data Flow Macroscopique", permet non seulement la prise en compte de structures classiques des langages de programmation (IF, CASE, LOOP,...), mais est aussi applicable à des structures plus complexes à mettre en oeuvre telles que celles rencontrées dans le langage ADA (tâche, rendez-vous, exception).

Basée sur l'examen de nombreuses machines parallèles, de réseaux d'interconnexion et d'une machine langage, une architecture est proposée pour l'application de ce modèle au langage Ada.

MOTS CLES : parallélisme - dirigée par les données - multi-processeurs - réseaux d'interconnexion - machine langage - langage ADA - rendez-vous - exception.