N° d'ordre : 767

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir le titre de

DOCTEUR DE 3ème CYCLE

(TRAITEMENT DE L'INFORMATION)

par

Marie Paule LECOUFFE

ETUDE ET DEFINITION D'UN MODELE DE MACHINE PARALLELE DYNAMIQUE DIRIGE PAR LES DONNEES

Soutenue le 5 juillet 1979, devant la Commission d'Examen

MEMBRES DU JURY :		
	Président :	M. V. CORDONNIER
	Rapporteur :	M. V. CORDONNIER
	Membres :	M. C. CARREZ
		M. J. LENFANT
		M. J.C. SYRE

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE

01.02.1976

DOYENS HONORAIRES de l'Ancienne Faculté des Sciences

MM. R. DEFRETIN, H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES des Anciennes Facultés de Droit et Sciences Economiques, des Sciences et des Lettres

M. ARNOULT, Mme BEAUJEU, MM. BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, CORSIN, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, P. GERMAIN, HEIM DE BALSAC, HOCQUETTE, KAMPE DE FERIET, KOUGANOFF, LAMOTTE, LASSERRE, LELONG, Mme LELONG, MM. LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, NORMANT, PEREZ, ROIG, ROSEAU, ROUBINE, ROUELLE, SAVART, WATERLOT, WIEMAN, ZAMANSKI.

PRESIDENTS HONORAIRES DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. R. DEFRETIN, M. PARREAU.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

M. J. LOMBARD.

PROFESSEURS TITULAIRES

M.	BACCHUS Pierre	Astronomie
M.	BEAUFILS Jean-Pierre	Chimie Physique
M.	BECART Maurice	Physique Atomique et Moléculaire
M.	BILLARD Jean	Physique du Solide
M.	BIAYS Pierre	Géographie
M.	BONNEMAN Pierre	Chimie Appliquée
M.	BONNOT Ernest	Biologie Végétale
M.	BONTE Antoine	Géologie Appliquée
M.	BOUGHON Pierre	Algèbre
M.	BOURIQUET Robert	Biologie Végétale
M.	CELET Paul	Géologie Générale
M.	CONSTANT Eugène	Electronique
M.	DECUYPER Marcel	Géométrie
M.	DELATTRE Charles	Géologie Générale
M.	DELHAYE Michel	Chimie Physique
M.	DERCOURT Michel	Géologie Générale
M.	DURCHON Maurice	Biologie Expérimentale
M.	FAURE Robert	Mécanique
M.	FOURET René	Physique du Solide
M.	GABILLARD Robert	Electronique
M.	GLACET Charles	Chimie Organique
M.	GONTIER Gérard	Mécanique
M.	GRUSON Laurent	Algèbre
M.	GUILLAUME Jean	Microbiologie
M.	HEUBEL Joseph	Chimie Minérale
M.	LABLACHE-COMBIER Alain	Chimie Organique
M.	LANSRAUX Guy	Physique Atomique et Moléculaire
M.	LAVEINE Jean-Pierre	Paléontologie
M.	LEBRUN André	Electronique
M.	LEHMANN Daniel	Géométrie

.../...

Mme	LENOBLE Jacqueline	Physique Atomique et Moléculaire
M.	LINDER Robert	Biologie et Physiologie Végétales
M.	LOMBARD Jacques	Sociologie
M.	LOUCHEUX Claude	Chimie Physique
M.	LUCQUIN Michel	Chimie Physique
M.	MAILLET Pierre	Sciences Economiques
M.	MONTARIOL Frédéric	Chimie Appliquée
M.	MONTREUIL Jean	Biochimie
M.	PARREAU Michel	Analyse
M.	POUZET Pierre	Analyse Numérique
M.	PROUVOST Jean	Minéralogie
M.	SALMER Georges	Electronique
M.	SCHILTZ René	Physique Atomique et Moléculaire
Mme	SCHWARTZ Marie-Hélène	Géométrie
M.	SEGUIER Guy	Electrotechnique
M.	TILLIEU Jacques	Physique Théorique
M.	TRIDOT Gabriel	Chimie Appliquée
M.	VIDAL Pierre	Automatique
M.	VIVIER Emile	Biologie Cellulaire
M.	WERTHEIMER Raymond	Physique Atomique et Moléculaire
M.	ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS SANS CHAIRE

M.	BELLET Jean	Physique Atomique et Moléculaire
M.	BODARD Marcel	Biologie Végétale
M.	BOILLET Pierre	Physique Atomique et Moléculaire
M.	BOILLY Bénoni	Biologie Animale
M.	BRIDOUX Michel	Chimie Physique
M.	CAPURON Alfred	Biologie Animale
M.	CORTOIS Jean	Physique Nucléaire et Corpusculaire
M.	DEBOURSE Jean-Pierre	Gestion des entreprises
M.	DEPREZ Gilbert	Physique Théorique
M.	DEVRAINNE Pierre	Chimie Minérale
M.	GOUDMAND Pierre	Chimie Physique
M.	GUILBAULT Pierre	Physiologie Animale
M.	LACOSTE Louis	Biologie Végétale
Mme	LEHMANN Josiane	Analyse
M.	LENTACKER Firmin	Géographie
M.	LOUAGE Francis	Electronique
Mle	MARQUET Simone	Probabilités
M.	MIGEON Michel	Chimie Physique
M.	MONTEL Marc	Physique du Solide
M.	PANET Marius	Electrotechnique
M.	RACZY Ladislas	Electronique
M.	ROUSSEAU Jean-Paul	Physiologie Animale
M.	SLIWA Henri	Chimie Organique

MAITRES DE CONFERENCES (et chargés d'Enseignement)

M.	ADAM Michel	Sciences Economiques
M.	ANTOINE Philippe	Analyse
M.	BART André	Biologie Animale
M.	BEGUIN Paul	Mécanique
M.	BKOUCHE Rudolphe	Algèbre
M.	BONNELLE Jean-Pierre	Chimie
M.	BONNEMAIN Jean-Louis	Biologie Végétale
M.	BOSCQ Denis	Probabilités
M.	BREZINSKI Claude	Analyse Numérique
M.	BRUYELLE Pierre	Géographie

.../...

M. CARREZ Christian	Informatique
M. CORDONNIER Vincent	Informatique
M. COQUERY Jean-Marie	Psycho-Physiologie
Mle DACHARRY Monique	Géographie
M. DEBENEST Jean	Sciences Economiques
M. DEBRABANT Pierre	Géologie Appliquée
M. DE PARIS Jean-Claude	Mathématiques
M. DHAINAUT André	Biologie Animale
M. DELAUNAY Jean-Claude	Sciences Economiques
M. DERIEUX Jean-Claude	Microbiologie
M. DOUKHAN Jean-Claude	Physique du Solide
M. DUBOIS Henri	Physique
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
Me EVRARD Micheline	Chimie Appliquée
M. FONTAINE Jacques-Marie	Electronique
M. FOURNET Bernard	Biochimie
M. FORELICH Daniel	Chimie Physique
M. GAMBLIN André	Géographie
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GUILLAUME Henri	Sciences Economiques
M. HECTOR Joseph	Géométrie
M. JACOB Gérard	Informatique
M. JOURNEL Gérard	Physique Atomique et Moléculaire
Mle KOSMAN Yvette	Géométrie
M. KREMBEL Jean	Biochimie
M. LAURENT François	Automatique
Mle LEGRAND Denise	Algèbre
Mle LEGRAND Solange	Algèbre
M. LEROY Jean-Marie	Chimie Appliquée
M. LEROY Yves	Electronique
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique Théorique
M. LOUCHET Pierre	Sciences de l'Education
M. MACKE Bruno	Physique
M. MAHIEU Jean-Marie	Physique Atomique et Moléculaire
Me N'GUYEN VAN CHI Régine	Géographie
M. MAIZIERES Christian	Automatique
M. MALAUSSENA Jean-Louis	Sciences Economiques
M. MESSELYN Jean	Physique Atomique et Moléculaire
M. MONTUELLE Bernard	Biologique Appliquée
M. NICOLE Jacques	Chimie Appliquée
M. PAQUET Jacques	Géologie Générale
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie Physique
M. PERROT Pierre	Chimie Appliquée
M. PERTUZON Emile	Physiologie Animale
M. PONSOLLE Louis	Chimie Physique
M. POVY Lucien	Automatique
M. RICHARD Alain	Biologie
M. ROGALSKI Marc	Analyse
M. ROY Jean-Claude	Psycho-Physiologie
M. SIMON Michel	Sociologie
M. SOMME Jean	Géographie
Mle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. STEEN Jean-Pierre	Informatique

.../...

M.	THERY Pierre	Electronique
M.	TOULOTTE Jean-Marc	Automatique
M.	TREANTON Jean-René	Sociologie
M.	VANDORPE Bernard	Chimie Minérale
M.	VILLETTE Michel	Mécanique
M.	WALLART Francis	Chimie
M.	WERNIER Georges	Informatique
M.	WATERLOT Michel	Géologie Générale
Mme	ZINN-JUSTIN Nicole	Algèbre

Je tiens à remercier

Monsieur CORDONNIER, Professeur à l'Université de Lille I, qui m'a fait l'honneur de présider le Jury de cette thèse, et qui, après m'avoir orientée vers cette recherche, n'a cessé de me prodiguer conseils et encouragements,

Monsieur CARREZ, Professeur à l'Université de Lille I, pour l'attention avec laquelle il a examiné mes travaux, ainsi que pour ses nombreux conseils et ses critiques constructives,

Monsieur LENFANT, Maître de Conférence à l'Université de Rennes, d'avoir bien voulu examiner ce rapport,

Monsieur SYRE, Ingénieur au CERT, pour l'intérêt qu'il a porté à ce travail et pour l'aide qu'il m'a apportée tout au long de cette recherche,

Je remercie également

Mon mari et tous ceux qui par leur compréhension ou de fructueuses discussions ont contribué à l'avancement de ce travail,

Enfin, je tiens à remercier ceux qui ont assuré la réalisation matérielle de cette thèse dans des délais très rapides :

Mademoiselle FIEVET qui a dactylographié ce texte avec beaucoup de gentillesse et de compétence,

Monsieur et Madame DEBOCK qui avec beaucoup de soin et de diligence ont assuré le tirage de ce document.

A PIERRE,

A MES PARENTS.

TABLE DES MATIERES

CHAPITRE III - LE MODELE DYNAMIQUE

0. Introduction

1. Limites et inconvénients du modèle statique

2. Introduction de primitives nouvelles
   PROGRAMME POUR MAUD

3. Paramètres des nouvelles primitives

1. PRIMITIVE EXECBLOC

2. PRIMITIVE ATTENDRE

4. Graphe des programmes executables

1. NOTATIONS

2. GRAPHES

5. Implications du caractère dynamique au niveau du modèle

1. LE MODELE DYNAMIQUE

2. NOUVEAUX ENSEMBLES D'OBJETS
   1. Ensemble D de demandes d'exécution
   2. BIBLIOTHEQUE

6. Implications du caractère dynamique au niveau des processeurs de traitement

1. PRIMITIVE DEBUT-DE-BLOC

2. PRIMITIVE EXECBLOC
   A. Cas général
   B. Cas des opérandes marqués

3. PRIMITIVE ATTENDRE

7. Fonction des opérateurs dans le modèle dynamique

1. OPERATEUR DE TRAITEMENT T_i

2. OPERATEUR CONSTRUCTEUR

3. OPERATEUR MAJ

8. Etapes de la vie d'un BLOC

9. Gestion des noms de communication
   FOURNISSEUR DE NOMS DE COMMUNICATION

CHAPITRE IV - PROPOSITION D'ARCHITECTURE

1. Caractéristiques d'une architecture adaptée au modèle

2. Communication entre processeurs par une mémoire commune

1. SOLUTIONS CLASSIQUES

2. UTILISATION D'UNE MEMOIRE CIRCULANTE

3. Réalisation des opérateurs

4. Lá mémoire de MAUD

1. ORGANISATIONS POSSIBLES
   1. Quatre parties
   2. Objets les uns derrière les autres
   3. *Cadre*

2. PROCEDURES D'ECHANGE ENTRE LES PROCESSEURS ET L'ANNEAU
   1. Processeurs de traitement
   2. CONSTRUCTEUR
   3. MAJ

5. Solution 1 : Bloc en attente dans l'anneau

6. Solution 2 : domaines de sortie dans l'anneau

7. Solution 3 : cadres domaines de sortie "typés"

8. Echanges avec l'extérieur

9. Interruptions

10. Traitement des erreurs

11. Régulation de la charge de l'anneau

1. ROLE DU GERANT

2. CHOIX DES OBJETS A ECARTER DE L'ANNEAU

INTRODUCTION

Le besoin d'accélérer la vitesse de traitement des machines conduit à la nécessité du traitement parallèle de l'information. La percée des microprocesseurs facilite ce type de traitement. En effet, les microprocesseurs permettent de réaliser facilement et à un faible coût des groupements de processeurs, et par là de réaliser des architectures de type MIMD (multiple instruction stream multiple data stream) qui ont le meilleur potentiel d'exploitation du parallélisme.

Bien que l'on sache réaliser matériellement de tels groupements de microprocesseurs, on sait encore peu les utiliser car de nombreux problèmes se posent ; entre autres, la répartition du travail entre les différents processeurs, la communication des données entre eux, la synchronisation des opérations. Ces difficultés ont plusieurs explications :

- les architectures à base de microprocesseurs sont calquées sur les architectures conventionnelles dans lesquelles le flot de contrôle centralisé est inadéquat pour représenter le parallélisme ;

- les langages de programmation utilisés reflètent les caractères séquentiels des machines pour lesquels ils ont été définis, bien que certains moyens d'expression du parallélisme leur aient été ajoutés : primitives de contrôle ([Con 66] et parbegin, parend d'Algol 68) et primitives de synchronisation ([Dij 68], [Bri 73],[RV 77]).

Une alternative à ces problèmes est le traitement dirigé 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 machines sont ainsi capables de détecter à l'exécution les opérations qui peuvent s'exécuter en parallèle. De nombreuses études ont été faites et sont en cours à ce sujet : [Den 74], [Syr 76], [AGP 78], [Syr 78].

Cependant le parallélisme est exploité en général au niveau des instructions d'un programme. Dans ce rapport, nous proposons d'exploiter le parallélisme à un niveau plus global que celui de l'instruction. Pour cela, nous introduisons une notion de BLOC, qui est différente de celle utilisée dans les langages de programmation classiques : un BLOC se compose d'un ensemble d'instructions et de l'ensemble des objets utilisés par ces instructions, un programme étant constitué d'un ensemble de BLOCS. L'application de la règle d'Assignation Unique au niveau des BLOCS permet d'exprimer naturellement le parallélisme d'exécution existant entre les BLOCS et assure le déterminisme.

Dans le Chapitre II, nous décrivons un modèle de machine parallèle, dirigée par les données, utilisant une structuration des programmes en BLOCS. Ce modèle comporte un certains nombre d'opérateurs de traitement et un opérateur spécialisé. Tous ces opérateurs ne communiquent pas directement entre eux mais par l'intermédiaire d'ensembles communs d'informations.

Le chapitre III décrit une extension de ce modèle permettant de donner une composante dynamique à l'exécution d'un programme, grâce à l'utilisation de nouvelles primitives au niveau des opérateurs de traitement. Celles-ci autorisent des graphes d'exécution de programmes plus variés que dans le modèle précédent, mais nécessitent l'adjonction d'un nouvel opérateur.

Une proposition d'architecture pour ce dernier modèle est faite dans le chapitre IV, à partir d'une mémoire circulante utilisée comme mémoire commune. Diverses solutions sont envisagées quant à la répartition des informations dans la mémoire commune.

CHAPITRE I

L'EXPLOITATION DU PARALLELISME

DANS LES ORDINATEURS

1. L'EXPLOITATION DU PARALLELISME DANS LES MACHINES CLASSIQUES

Le parallélisme implicitement ou explicitement présent dans un algorithme est encore peu pris en compte par les machines actuelles :

* Soit parce que le langage dans lequel est décrit cet algorithme ne permet pas de l'exprimer suffisamment,

* Soit parce que la machine sur laquelle est exécuté cet algorithme est incapable de le supporter, parce qu'elle ne dispose pas de moyen de détection du parallélisme ou parce que son organisation ne permet pas de le concrétiser à l'exécution, sauf peut-être à un niveau relativement spécifique (Entrées/Sorties par exemple).

Pourtant un certain nombre de tentatives ont été faites afin de mieux prendre en compte ce parallélisme, soit au niveau matériel, soit au niveau logiciel.

1.1. AU NIVEAU MATÉRIEL

C'est une recherche dans l'accroissement de la vitesse de traitement qui a mis en évidence les possibilités de parallélisme à ce niveau. Trois types de machines ont ainsi vu le jour :

1.1.1. LES PIPELINES

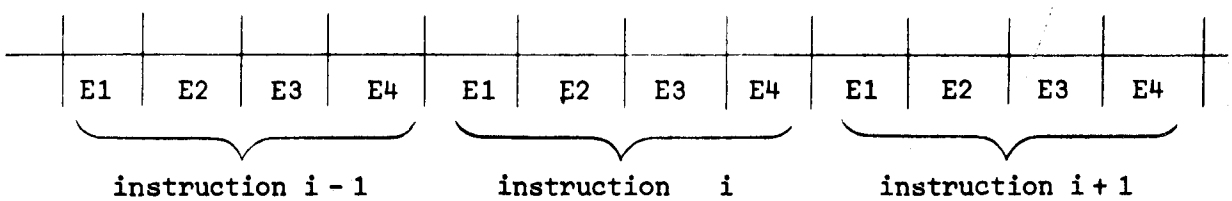
Le concept de pipeline représente une forme précise du parallélisme qui consiste à proposer des moyens de réaliser une exécution simultanée de plusieurs tâches à partir d'une description série [Cor 78].

Les machines pipelines atteignent leurs performances par le chevauchement des différentes étapes contribuant à l'achèvement d'un travail. Elles se composent de n unités autonomes, chacune spécialisée dans le traitement de l'une des étapes du travail à effectuer.

Cette technique est très intéressante pour une tâche se répétant souvent, comme par exemple le processus d'exécution d'une instruction. Ce processus peut se décomposer en quatre étapes :

E1 : Recherche de l'instruction
E2 : Décodage
E3 : Recherche des opérandes
E4 : Exécution proprement dite

Si ce processus est confié à une seule unité le diagramme des temps est le suivant :



Si on dispose de quatre unités autonomes capables d'assurer chacune des étapes E1, E2, E3, E4, le diagramme des temps est le suivant :

temps / unités	t_1	t_2	t_3	t_4	t_5	t_6	t_7			
UE1	I_j	I_{j+1}	I_{j+2}	I_{j+3}						
UE2		I_j	I_{j+1}	I_{j+2}	I_{j+3}					
UE3			I_j	I_{j+1}	I_{j+2}	I_{j+3}				
UE4				I_j	I_{j+1}	I_{j+2}	I_{j+3}			

où I_j, I_{j+1}, I_{j+2}... représentent une séquence d'instructions à exécuter.

La prise en charge des différentes étapes par des unités autonomes permet un recouvrement temporel qui est une forme de parallélisme ; au temps t_4 :

- UE4 Exécute l'instruction I_j
- UE3 Recherche les opérandes de I_{j+1}
- UE2 Décode l'instruction I_{j+2}
- UE1 Recherche l'instruction I_{j+3}

Cette technique est applicable à un travail qui peut être fait en n étapes, on multiplie alors le débit par n (la durée de chaque étape restant la même). Elle devient efficace lorsque ce travail doit se répéter souvent.

Une autre utilisation des pipelines concerne l'application d'une opération unique à un flot de données. C'est le principe utilisé dans Cray-I : les opérateurs vectoriels appliquent une même fonction à des ensembles d'opérandes (vecteurs).

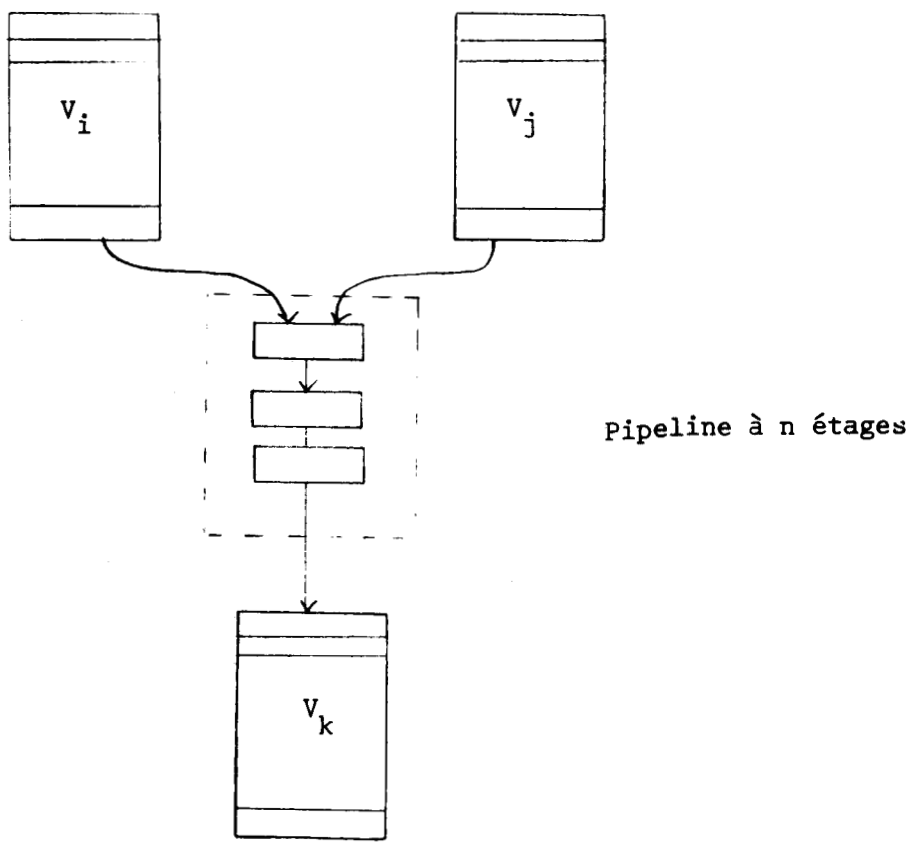


Figure 1. : Instruction vectorielle dans Cray-I.

1.1.2. LES MACHINES TABLEAUX

Ces machines utilisent une seule unité de contrôle pour commander un certain nombre d'unités synchrones qui accomplissent la même fonction simultanément sur un certain nombre de données. Ces unités sont appelées des processeurs élémentaires et disposent d'une mémoire propre.

Les machines tableaux sont des machines de type SIMD (Single Instruction Multiple Data) dans la classification de Flynn. Leur intérêt réside dans l'exécution parallèle d'une même opération sur tous les éléments d'un vecteur ou d'une matrice, chaque opération étant effectuée simultanément par un des processeurs élémentaires.

ILLIAC IV [Bou 72] est l'exemple type de cette classe de machines. C'est une machine composée de 256 processeurs élémentaires groupés en 4 sous-tableaux de 64 chaque sous-tableau possédant une seule unité de contrôle. Mais seul un sous-tableau de 64 éléments a été réalisé.
Chaque processeur élémentaire a sa propre mémoire locale et dispose d'un index pour l'adressage dans cette mémoire. Le système ne dispose pas de mémoire centrale : elle est représentée par les 64 mémoires locales. L'organisation des données dans les mémoires locales pose toutefois quelques problèmes, surtout si les matrices ou les vecteurs utilisés n'ont pas exactement la taille idéale de 64 : le travail doit alors être préparé et cela n'est intéressant que si le pourcentage de traitement parallèle dans le traitement total est très grand. Il apparaît en outre un problème de communication entre processeurs qui restreint souvent les possibilités de simultanéit

1.1.3. LES MULTIPROCESSEURS

Ces machines comprennent un certain nombre d'unités centrales (qui sont alors appelées processeurs) qui fonctionnent simultanément et se partagent un même ensemble de mémoires et de périphériques, chaque processeur étant capable d'effectuer n'importe quel travail.

L'avantage de cette organisation réside surtout dans une augmentation de la fiabilité puisqu'il devient possible de fonctionner en mode dégradé.

De plus, on accroît les performances, bien que le doublement d'une unité centrale n'entraîne pas tout-à-fait le doublement de la puissance, compte-tenu des conflits supplémentaires qui en découlent pour l'accès aux ressources communes, et principalement à la mémoire qui peut devenir un goulot d'étranglement. Car le principal problème de ce type d'organisation est un problème d'interconnexions entre processeurs et mémoire.

1.1.4. LES MULTIORDINATEURS

Les multiordinateurs sont composés d'un certain nombre d'ordinateurs interconnectés par un outil de communication, ou reliés entre eux par des connexions directes. Chaque processeur possède une mémoire locale. Les processeurs peuvent être tous identiques, ou certains peuvent être banalisés, mais tous peuvent effectuer simultanément un travail particulier.

Comme pour les multiprocesseurs, on accroît les performances et la fiabilité. Cependant, les processeurs ayant chacun leur mémoire locale, ils sont capables de fonctionner de façon autonome pendant un certain temps à partir du moment où ils ont tous les éléments nécessaires pour effectuer un travail. Il n'existe plus de conflits d'accès à la mémoire (sauf peut-être dans les cas particuliers de systèmes hybrides où les communications entre processeurs se font par l'intermédiaire d'une mémoire commune), mais seulement des problèmes d'interconnexion entre processeurs.

Ce type d'organisation est d'ailleurs favorisé par le développement actuel des microprocesseurs qui abaisse le coût d'un système informatique. Il est possible même d'envisager des multimicroprocesseurs comportant un grand nombre de processeurs (supérieur à 100) même si leur taux d'utilisation n'est pas très élevé.

Les multiordinateurs sont certainement les machines qui ont le meilleur potentiel d'exploitation du parallélisme. Mais le problème le plus important est celui de l'utilisation de ces machines, problème qui se pose aussi pour les multiprocesseurs d'ailleurs : comment distribuer le travail entre les différents processeurs ? Comment les coordonner ? Cela ne facilite pas la tâche des systèmes d'exploitation qui doivent avoir une connaissance du parallélisme qui existe dans le traitement à effectuer.

De ce côté, un certain nombre de moyens logiciels existent et nous en parlons brièvement au paragraphe I.1.2.

Dans tous les cas, il apparaît nécessaire de situer une approche parallèle par rapport à tous les aspects du traitement, c'est-à-dire à partir de l'algorithme en passant par son expression programmée. Ceci implique, beaucoup plus que dans les machines classiques, une étroite interdépendance entre le matériel et le logiciel.

1.2. AU NIVEAU LOGICIEL

Des mécanismes permettant d'expliciter le parallélisme ont été introduits dans les langages classiques, comme en Algol 68, PL/I ou Pascal. Deux classes de primitives ont été ajoutées :

- des primitives de contrôle, telles que FORK et JOIN [Con 63], qui permettent l'exécution simultanée de plusieurs ensembles d'instructions.

- des primitives de synchronisation qui sont utilisées pour coordonner des processus séquentiels ; ce sont, entre autres, les moniteurs [Bri 73], les sémaphores [Dij 68], les modules de contrôle [RV 77].

Toutefois, l'utilisation de ces primitives ne permet pas toujours d'exprimer tout le parallélisme contenu dans le programme. D'autre part, la manipulation de ces primitives est, au moins en partie, à la charge du programmeur : il doit exprimer explicitement le parallélisme contenu dans son algorithme, encore faut-il qu'il le connaisse. Cela demande un certain effort de codage d'informations supplémentaires et réduit la lisibilité du programme.

Afin d'éviter ces problèmes, une transformation automatique de programmes séquentiels en programmes parallèles a été étudiée par [RW 73]. Elle était basée sur une structuration des programmes en blocs, un bloc étant constitué d'un ensemble d'instructions d'affectation dépendant d'un même ensemble de conditions. A un programme est associée une variable d'activation, et à chaque bloc est associée une valeur exclusive de cette variable.

L'exécution d'un bloc est déclenchée lorsque la variable d'activation prend la valeur associée à ce bloc.

Cependant, une meilleure solution à ces problèmes est que seule la façon de représenter le programme suffise à exprimer la séquentialité et le parallélisme, comme nous le verrons au paragraphe II.2.

1.3. CONCLUSION

Il existe pour chaque type de machine décrit ci-dessus des limites au degré de parallélisme atteint dans le traitement d'une tâche. En effet, et particulièrement pour les machines pipelines et les machines tableaux, elles sont limitées à un flot unique d'instructions. De fréquentes mises à jour de la mémoire sont nécessaires, ce qui freine le traitement ; surtout pour les machines pipeline où les conflits d'accès à une même adresse mémoire doivent être contrôlés et gérés.

Par contre, les multiprocesseurs ont la faculté de traiter des flots multiples d'instructions et de données et ont donc le maximum de possibilités pour exploiter le parallélisme. Cependant, trop souvent encore, ils possèdent au niveau matériel certaines limites qui les obligent à utiliser une exécution séquentielle (registres centralisés, compteur ordinal). De plus, même si certains outils logiciels sont fournis à l'utilisateur ils sont programmés dans des langages possédant une structure séquentielle inhérente et il y a un désaccord entre la représentation utilisée dans la machine pour exploiter le parallélisme et la représentation fournie par le langage.

2. TRAITEMENT DIRIGE PAR LES DONNEES

Une alternative à ce problème d'exploitation du parallélisme est l'approche "traitement dirigé par les données", expression que nous utilisons pour traduire "data flow".

Dans une machine dirigée par les données, l'exécution d'un programme n'est plus dirigée par le flot des instructions, mais par le flot des données.

Dans la plupart des cas, la conception des machines dirigées par les données est faite à partir des langages de programmation.

Un programme dirigé par les données est un ensemble d'opérations partiellement ordonné où l'ordre partiel est déterminé à l'exécution, uniquement par la nécessité d'obtenir des valeurs, c'est-à-dire :

* une opération ne peut s'exécuter que lorsque tous sés opérandes sont disponibles,

* de plus, une opération est purement fonctionnelle et ne produit pas d'effets de bord comme résultat de son exécution.

Par conséquent, toutes les instructions dont les opérandes sont calculés peuvent s'exécuter en parallèle.

2.1. TRAVAUX DU MIT

Les premières études sur le traitement dirigé par les données ont été faites au MIT dans l'équipe du professeur Dennis et utilisent une représentation graphique pour la représentation des programmes.

Dans le langage dirigé par les données défini par Dennis [Den 74], un programme est un graphe orienté. Les noeuds du graphe sont de deux types :

* noeuds de chainage (link),

* noeuds acteurs (actor).

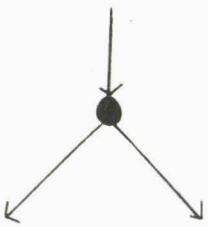
Ces noeuds sont reliés par des arcs qui sont des chemins le long desquels circulent des marques (token). Il existe deux sortes de marques :

* des marques de données associées à des valeurs ,

* des marques de contrôle associées à une valeur booléenne.

Les arcs spécifient les dépendances entre données. Il peut exister plusieurs arcs d'entrée, mais toujours un seul arc de sortie.

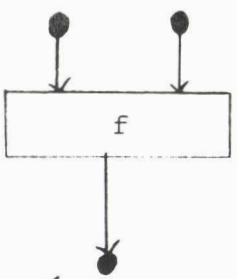
Les principaux types de noeuds du langage de base de Dennis sont décrits figure 2.



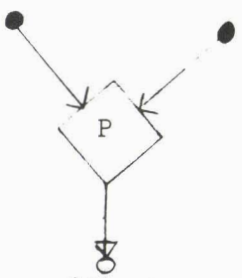
Chainage de données

Chainage de contrôle

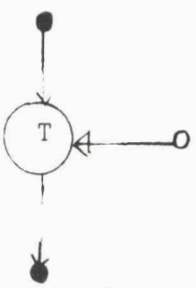
* noeuds de chainage



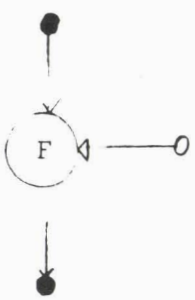
opérateur



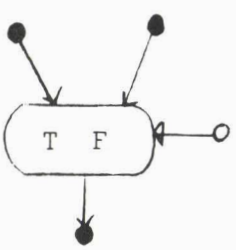
prédicat



porte T



porte F



merge

* noeuds acteurs

Figure 2. : Principaux noeuds du langage de base du MIT.

Un noeud acteur est activable quand des marques sont présentes sur tous ses arcs d'entrée et qu'il n'y a pas de marque sur l'arc de sortie. Il peut alors être activé à n'importe quel moment :

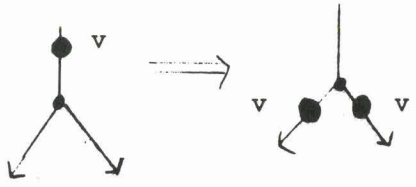
* les marques sont enlevées des arcs d'entrée,

* le calcul est effectué à partir des valeurs associées aux marques d'entrée,

* la valeur calculée est associée à une valeur résultat et placée sur l'arc de sortie.

Un résultat peut être envoyé à plus d'une destination par l'intermédiaire d'un noeud de chainage qui prend une marque sur son arc d'entrée et place des marques identiques sur ses arcs de sortie.

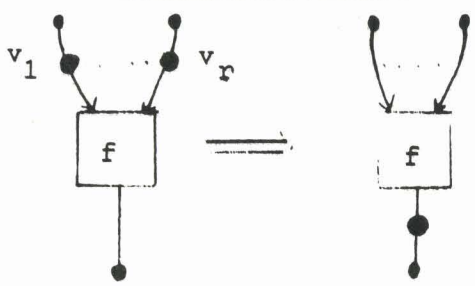
L'exécution d'un programme dirigé par les données est une suite de "flash", chacun montrant le graphe muni de ses marques et des valeurs associées. On passe d'un flash à l'autre par traversée des noeuds du graphe suivant un certain nombre de règles dont les principales sont décrites dans la figure 3.

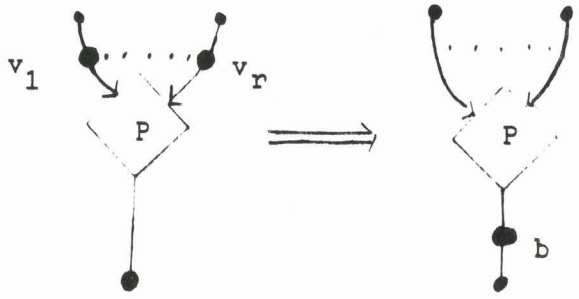


Chainage de donnée

Chainage de contrôle

* noeuds de chainage





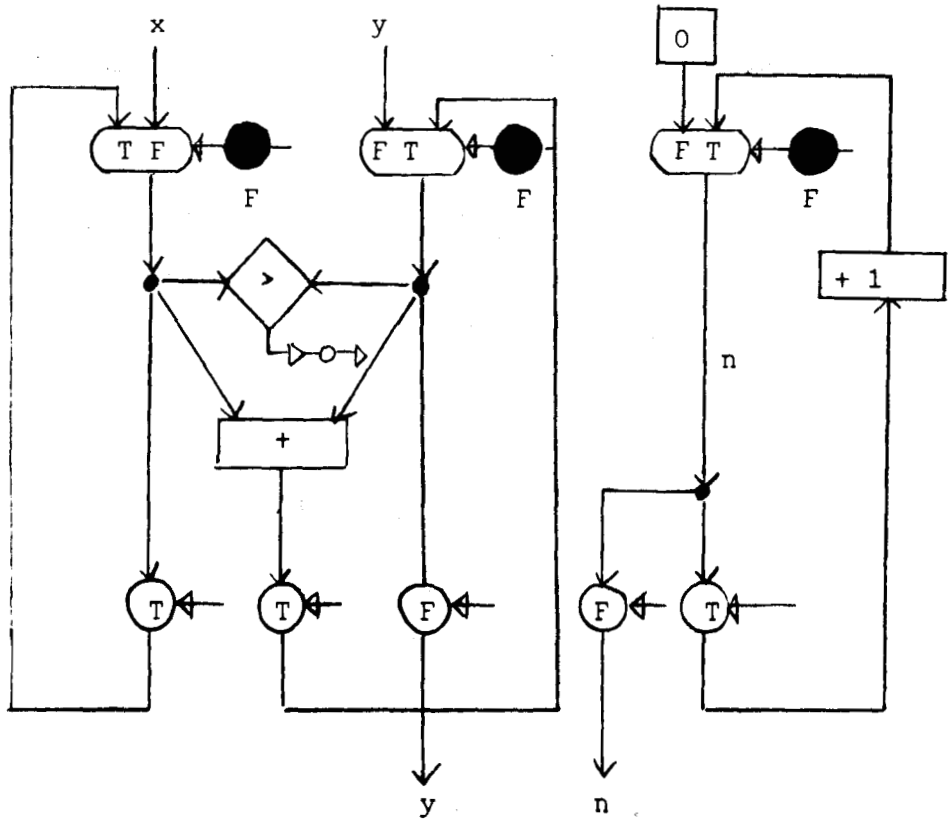
* Opérateurs et prédicats

Figure 3. : Règles d'activation des noeuds du graphe.

Exemple de programme

```
Input y, x
n : = 0
while y < x do
y : = y + x
n : = n + 1
end
output y, n
```

Schéma dirigé par les données correspondant



La structure de base d'une machine capable d'interpréter les schémas dirigés par les données [DM 74] comporte une mémoire qui est un ensemble de cellules (figure 4). Chaque cellule contient une instruction et les opérandes de cette instruction.

L'arbitre interconnecte les cellules de la mémoire avec les éléments de l'unité de traitement en fournissant un chemin entre une cellule et une unité de traitement élémentaire lorsque l'opération contenue dans la cellule est prête à être exécutée. L'unité de traitement exécute l'instruction et fournit au distributeur un résultat pour chaque instruction destination. Le distributeur fournit les résultats aux instructions désignées.

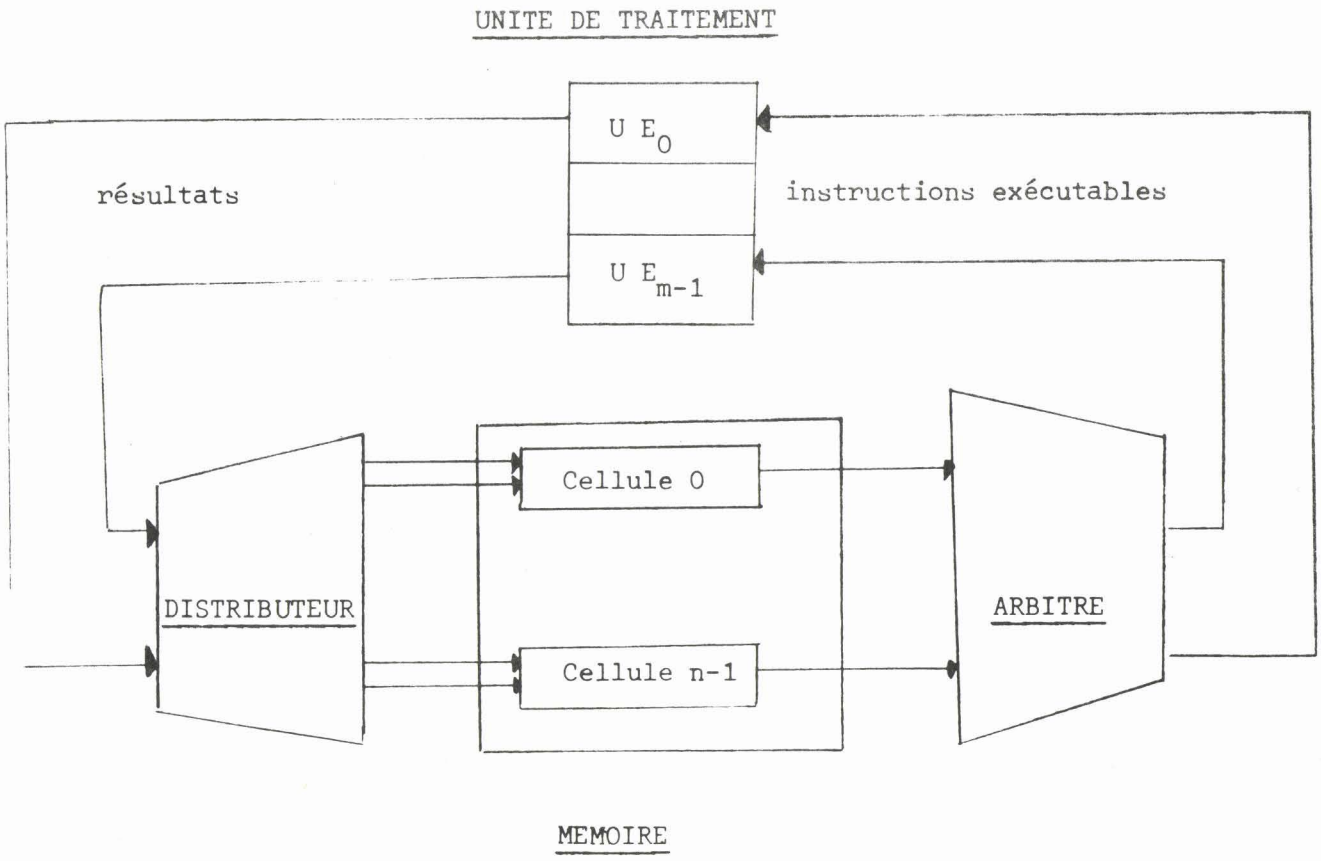


Figure 4. : Structure de base de la machine dirigée par les données du MIT.

2.2. Travaux de RUMBAUGH

Le modèle de RUMBAUGH utilise le fait qu'un programme est souvent structuré en procédures, chaque procédure pouvant être appelée plusieurs fois, simultanément ou non.

Chaque occurence d'un appel de procédure est appelée "activation de procédure" et contient un pointeur de retour au programme appelant pour indiquer la destination du résultat. Bien entendu, l'exécution proprement dite de la procédure s'effectue en parallèle avec l'exécution du programme appelant.

La machine de RUMBAUGH (Figure 5) [Rum 77] est composée de plusieurs unités asynchrones :

* plusieurs processeurs d'activation qui contiennent et exécutent une activation d'une procédure,

* un gérant qui coordonne les processeurs d'activation et leur assigne des activations,

* une mémoire de programme qui contient les procédures qui peuvent être appelées,

* une mémoire de rangement qui contient les activations de procédures temporairement inactives,

* un réseau d'échange qui effectue les transferts des activations de procédure entre les mémoires de programme et de rangement et les processeurs d'activation,

* une mémoire de structure qui contient les structures de données de taille trop importante pour résider dans les processeurs d'activation,

* des contrôleurs de structure qui contrôlent l'accès à la mémoire de structure, et agissent sur les structures à la demande des processeurs.

Toutes ces unités travaillent concurremment et indépendamment les unes des autres. Aucune synchronisation n'est nécessaire entre les différents processeurs d'activation.

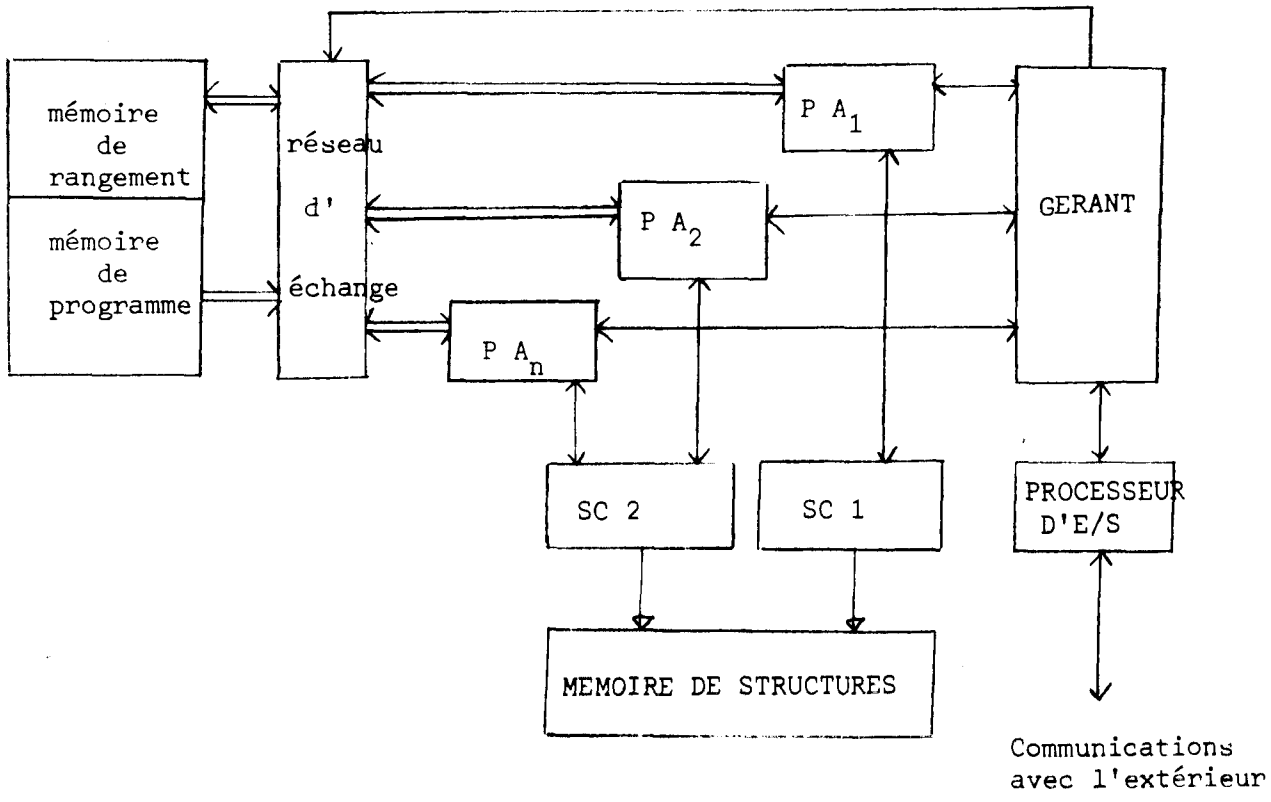
C'est le gérant qui crée, conserve et détruit les activations de procédure, et qui les assigne aux processeurs pour l'exécution. A chaque activation de procédure est assigné un identificateur unique, le pointeur d'activation. Le gérant gère donc trois tables :

* une table des processeurs, qui identifie les activations de procédure en cours dans les processeurs,

* une table d'activations, qui identifie l'appelant et l'endroit d'appel de chaque activation de procédure,

* une liste d'appels, qui contient les appels de procédures attendant d'être traitées.

Le contrôle centralisé des processeurs au niveau du gérant ne nous semble pas permettre une utilisation optimale des processeurs. Des processeurs capables de s'allouer eux mêmes une tâche autoriseraient un plus grand asynchronisme. Cependant, le gérant permet de connaître à chaque instant l'activité de la machine par simple consultation des tables ; dans l'autre hypothèse, il faudrait consulter chaque processeur avant d'avoir toute l'information.



PA = processeur d'activation ; SC = contrôleur de structure.

Figure 5. : Machine dirigée par les données de RUMBAUGH.

2.3. TRAVAUX DU CERT

Une machine dirigée par les données complètement originale est en cours de réalisation à Toulouse. Elle est basée sur le concept d'Assignation Unique [TE 68] qui est le suivant :

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

L'utilisation de l'Assignation Unique implique une expression naturelle du parallélisme contenu dans un programme et permet une exécution dirigée par les données. Cela permet même d'exprimer le parallélisme maximum contenu dans le programme lors de son exécution [Urs 73]. Ce qui montre l'intérêt de ce concept pour la réalisation de machines hautement parallèles.

L'équipe du CERT a d'abord étudié de façon approfondie les conséquences du concept d'Assignation Unique sur les principes de l'architecture des machines. Puis elle a étudié un système complet, logiciel et matériel, entièrement basé sur ce concept :

* un langage évolué d'Assignation Unique, non basé sur une représentation graphique, facilement utilisable parce que très proche des langages de programmation habituels,

* une structure multiprocesseur dans laquelle l'implémentation d'un procésseur élémentaire à été très précisément étudiée, puis simulée et dont la réalisation est pratiquement terminée.

La machine définie utilise la notion d'Unités de Programme. Une Unité de Programme est constituée :

* d'un ensemble d'instructions I,

* d'un ensemble de données d'entrée E,

* d'un ensemble de données de sortie S.

Une Unité de Programme est exécutée suivant un contrôle dirigé par les données, c'est-à-dire lorsque toutes les données d'entrée sont calculées. Elle est terminée lorsque toutes les données de sortie sont produites.

La structure générale du Processeur élémentaire d'Assignation Unique (PAU) est celle de la figure 6.

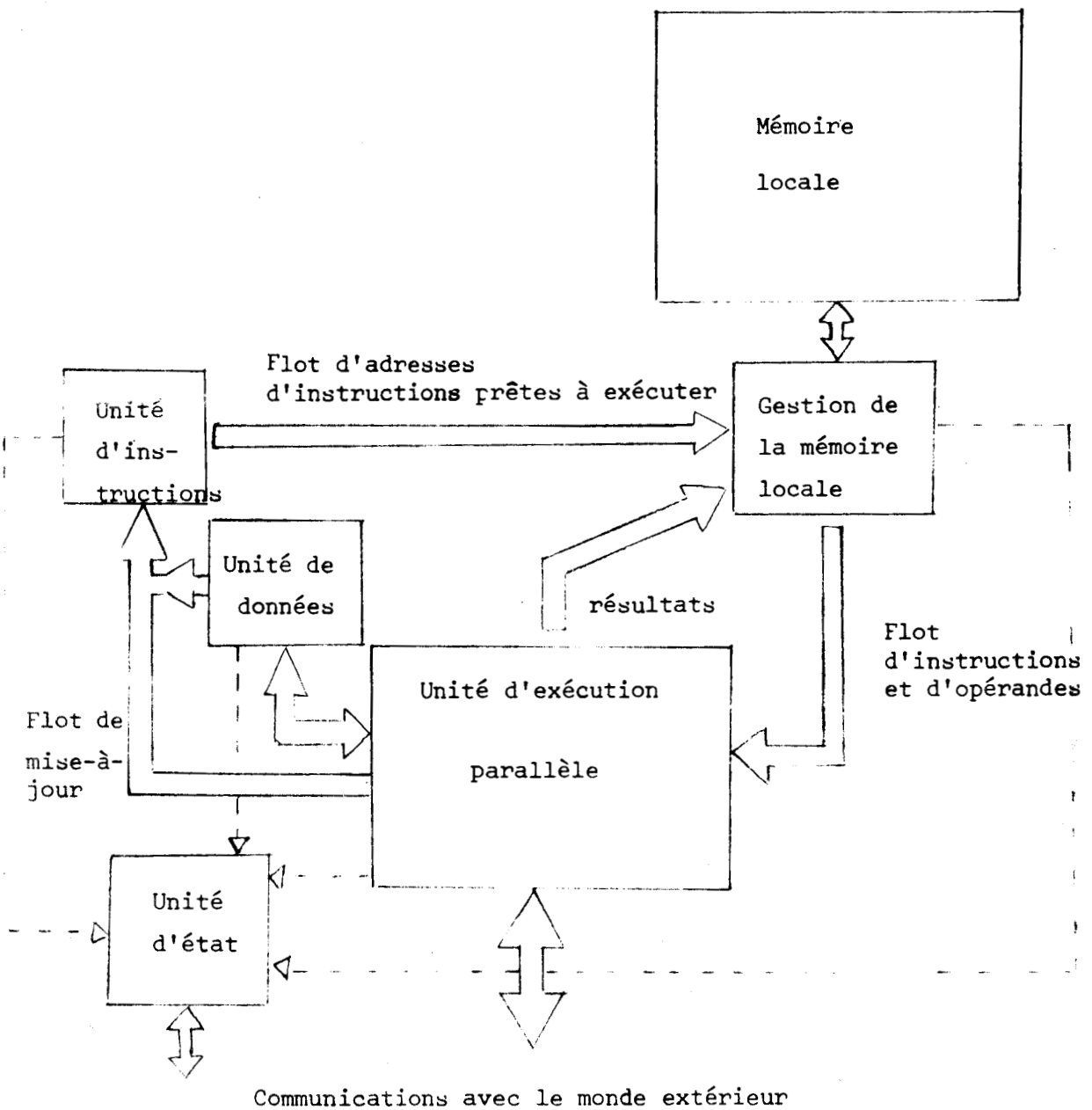


Figure 6. : Structure du PAU.

Le PAU est composé des éléments suivants :

1) la mémoire locale qui contient tout ou partie du programme, ainsi que les données nécessaires à l'exécution de cet ensemble d'instructions.

2) une unité d'exécution parallèle, qui est une unité pseudopipeline capable de traiter simultanément plusieurs instructions.

3) une unité de contrôle qui gère les mécanismes d'exécution dirigée par les données. Cette unité comprend :

i) une mémoire de contrôle d'instructions dont chaque mot est associé à un mot de la mémoire locale. Les mots ont une largeur de 3 bits : 2 bits, C_{i1} et C_{i2}, indiquent l'état des opérandes-données de l'instruction ; le troisième bit, C_{i0}, est un indicateur dépendant d'un certain nombre de conditions ; il est nécessité par l'imbrication possible des Unités de Programme.

A cette mémoire sont associés un opérateur de mise à jour des bits de contrôle et un mécanisme de recherche d'instructions prêtes à exécuter, c'est-à-dire pour lesquelles les 3 bits C_{i0}, C_{i1}, C_{i2} sont égaux à 1.

ii) une mémoire de contrôle d'opérandes dont chaque mot de 1 bit est associé à un mot de la mémoire locale. Ce bit, C_d, permet de connaître l'état des opérandes, calculé ou non. Associés à cette mémoire, un opérateur permet de faire les mises à jour des bits C_d, et un mécanisme de recherche des opérandes calculés active l'opérateur de mise à jour de la mémoire d'instructions en cas de recherche positive.

L'exécution d'une instruction se déroule de la façon suivante :

1) détection de l'exécutabilité de l'instruction par le contrôleur de la mémoire de contrôle d'instructions qui place son adresse dans une file d'adresses d'instructions prêtes destinées au gérant de la mémoire locale.

2) lecture de l'instruction par le gérant de la mémoire locale qui la place dans une file d'instructions prêtes à destination de l'Unité d'Exécution.

3) exécution de l'instruction par l'Unité d'Exécution. Un décodeur aiguille l'instruction vers l'opérateur concerné. Cet opérateur doit se charger de la lecture des opérandes d'entrée puis de l'écriture du résultat en mémoire locale en plus de l'exécution proprement dite de l'instruction. De même, c'est lui qui propage la connaissance du calcul du résultat en envoyant des ordres de mise à jour à l'unité d'instructions, ainsi qu'un ordre de mise à 1 du bit C_d associé à ce résultat vers l'unité de données.

Il est possible que le choix qui a été fait de faire supporter toutes les mises à jour impliquées par l'exécution d'une instruction à l'opérateur qui a exécuté cette instruction soit un choix coûteux parce qu'il immobilise l'opérateur. Ce choix n'est pas génant si cet opérateur existe à de multiples exemplaires. La raison invoquée pour ce choix est d'ailleurs justifiée par le fait qu'un opérateur unique chargé de toutes les mises à jour pourrait être un goulot d'étranglement. Cela est vrai parce que chaque instruction provoque des mises à jour, mais serait moins justifié si les mises à jour n'étaient faites qu'après l'exécution complète d'une Unité de Programme par exemple.

Cependant le taux de parallélisme atteint peut être très important : dans un exemple comme le calcul du champ laplacien, lorsque le nombre d'opérateurs fixé par la simulation est de 36, il y a jusqu'à 31 opérateurs actifs à certains moments de l'exécution. La comparaison avec le même algorithme exécuté sur un Iris 80 montre que le traitement est de 6 à 10 fois plus rapide sur le PAU. Bien entendu, ces résultats sont très relatifs car les deux machines, PAU et Iris 80, sont très différentes ; de plus, les résultats obtenus pour le PAU le sont par simulation. Cependant, la comparaison se fait toujours à l'avantage du PAU et il est raisonnable de penser qu'il en sera de même sur la machine réelle.

2.4. TRAVAUX DE L'UNIVERSITÉ D'IRVINE

Les recherches faites à l'Université d'Irvine se sont inspirées aussi du langage de base de Dennis et l'ont développé en associant un nom unique à chaque marque du graphe dirigé par les données. Un langage de haut niveau (Irvine Data flow language : ID) à assignation unique et basé sur la notion d'expression a été défini : un programme en ID est une liste d'expressions dont les quatre plus importantes sont les blocs, les expressions conditionnelles, les boucles et les applications de procédures. Les expressions conditionnelles et les boucles utilisent les concepts habituels. Les blocs permettent de diviser un programme en sous-expressions, tandis que les applications de procédures sont utilisées pour nommer et appeler des sous graphes dirigés par les données particuliers [AGP 78].

Le langage ID étant supposé pouvoir être utilisé pour l'écriture de systèmes d'exploitation, des moyens de synchronisation et de gestion des ressources ont été inclus, en particulier la notion de moniteur [AGP 77].

Les variables dans le langage ID sont des variables simples, dont la valeur est représentée par une seule marque dans le graphe dirigé par les données, ou des variables "stream", dont la valeur est représentée par une séquence ordonnée de marques (qui peut être non bornée), chaque marque portant une valeur simple. Ce type de variable peut être appelé un "train". L'intérêt des trains réside dans le fait qu'ils permettent d'entrer dans un opérateur ou de sortir d'un opérateur un nombre potentiellement infini de marques en n'utilisant qu'un seul arc. De plus, l'entrée et la sortie d'un train à travers un opérateur est complétement asynchrone, c'est-à-dire que toutes les marques d'entrée n'ont pas besoin d'être arrivées avant que des marques de sortie soient produites. Ce type de variable permet d'introduire un nouveau niveau d'asynchronisme qui est très intéressant pour l'exploitation du parallélisme.

Pour ce nouveau langage, un interpréteur "de débrouillement" (Unraveling Interpretor) a été développé : il permet l'exécution simultanée d'appels distincts de la même procédure et l'expansion automatique des boucles.

Une architecture basée sur cet interpréteur est en cours d'étude. Elle comprend une matrice de processeurs élémentaires tous identiques ; les processeurs élémentaires sont groupés en colonnes ; chacun d'eux est relié à un système de communication dans lequel circulent les marques associées aux valeurs, et à une unité mémoire par l'intermédiaire de contrôleurs de mémoire (figure 7) [AG 77].

Le système de communication est formé par un ensemble de bus bidirectionnels. Tous les processeurs se trouvant sur une même colonne se partagent le même bus (figure 8). Les marques circulent dans le système de communication à la recherche d'un processeur libre ou qui les attend. Afin de réduire le temps de circulation des marques, la machine peut se partitionner en domaines physiques comprenant chacun un certain nombre de processeurs élémentaires, de contrôleurs de mémoire et la partie associée du système de communication. L'espace de circulation des marques est ainsi réduit à un domaine (en gros, cela correspond à l'espace d'exécution d'une procédure), ce qui diminue le taux de communication dans le système, donc les conflits et permet d'accroître la vitesse de traitement.

Ce type d'architecture est très séduisant, cependant beaucoup de problèmes restent à résoudre.

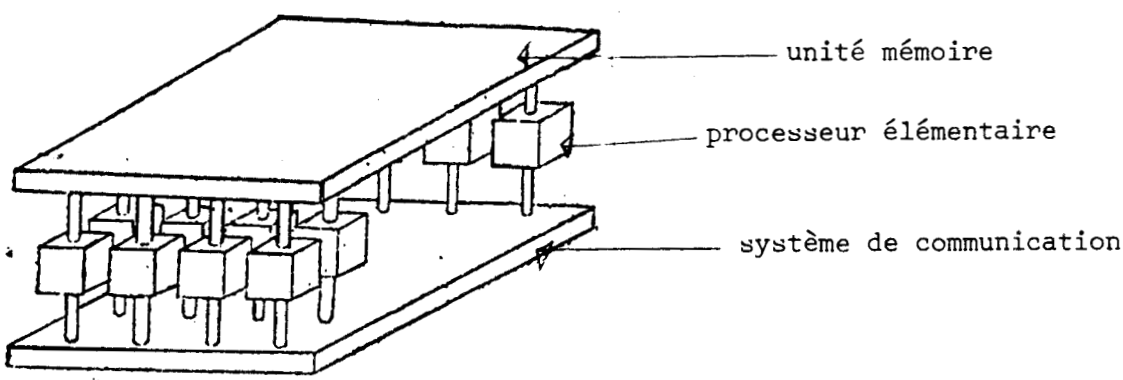


Figure 7. : Architecture proposée par l'Université d'Irvine.

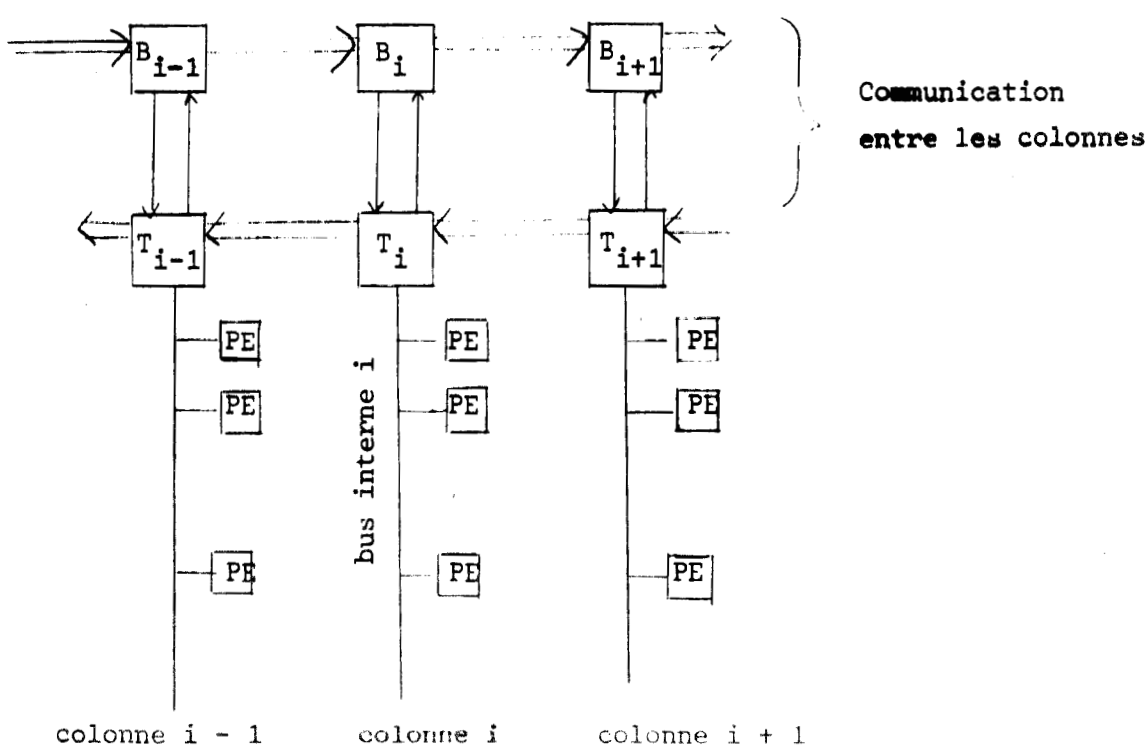


Figure 8. : Le système de communication.

2.5. TRAVAUX DE L'UNIVERSITÉ DE NEWCASTLE

Le but du projet de l'Université de Newcastle est d'étudier une machine de type MIMD (Multiple Instruction, Multiple Data) universelle en utilisant les meilleures caractéristiques des deux organisations : flot de contrôle et flot de données.

Pour cela, un modèle à flot de contrôle généralisé (Generalized Control Flow) a été défini. Ce modèle permet de supporter aussi bien une représentation d'un programme dirigée par les données lorsqu'un haut degré de parallélisme doit être exploité qu'une représentation type Von Neumann lorsqu'on utilise des langages conventionnels.

Le langage de programmation qui est en cours de développement est un langage de haut niveau basé sur une représentation à l'aide de graphes orientés et structurés (Structured Directed Graph representation), c'est-à-dire des graphes orientés dans lesquels on peut distinguer des sous-graphes.

Du modèle GCF a été déduite une architecture basée sur le contrôle dirigé par les données et qui utilise la notion de tâche, une tâche étant une instruction simple ou une procédure. La machine comprend (figure 9):

* une unité de gestion des tâches qui contient les adresses des tâches attendant d'être traitées,

* une unité de traitement qui peut comprendre un grand nombre de processeurs élémentaires,

* une mémoire qui contient le programme et les données.

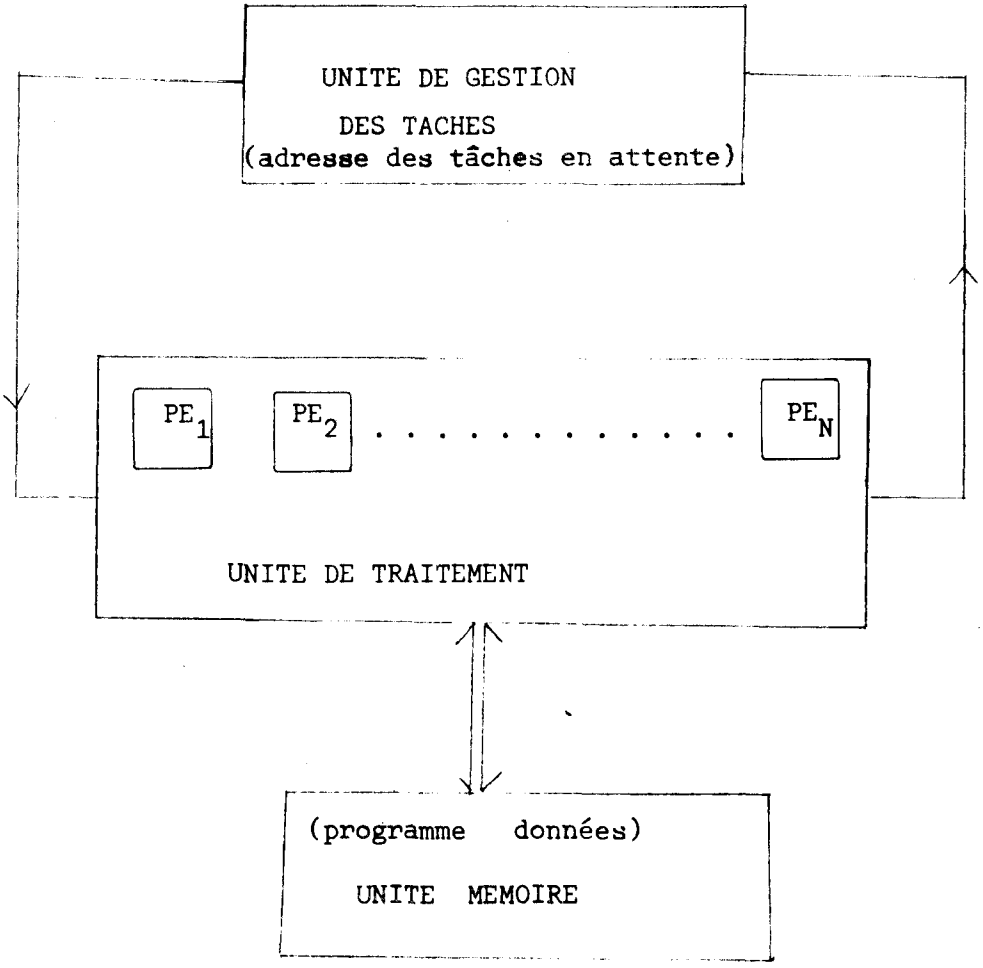


Figure 9. : Architecture GCF fonctionnelle.

Lorsqu'un processeur élémentaire est libre, il extrait l'adresse d'une instruction à exécuter de l'Unité de Tâches et recherche cette instruction dans la Mémoire. Chaque instruction contient :

* un code opération,

* les adresses des opérandes d'entrée,

* les adresses des opérandes résultats,

* l'adresse des instructions suivantes.

Après avoir recherché les opérandes d'entrée, effectué l'opération demandée et rangé le résultat, le processeur élémentaire place l'adresse de l'instruction suivante dans l'Unité de Gestion de Tâches.

Dans l'implémentation de cette machine, l'Unité Mémoire a été décomposée en 2 sous-unités, l'une contenant le programme et l'autre les données. Chaque sous-unité contient un certain nombre de processeurs mémoire, c'est-à-dire un bloc de mémoire et un contrôleur d'accès à ce bloc.

L'Unité de Gestion de Tâches est représentée par un tampon circulaire dans lequel chaque position peut contenir un "paquet". Il existe quatre sortes de "paquets" :

* le paquet NIA : qui contient une adresse d'instruction suivante,

* le paquet INC : qui contient une instruction sans ses opérandes d'entrée,

* le paquet COM : qui contient une instruction avec ses opérandes d'entrée,

* le paquet RES : qui contient le résultat et son adresse, ainsi que l'adresse de l'instruction suivante.

Les paquets circulent dans le tampon et sont pris, suivant leur type, par les processeurs concernés, qui, en échange, fournissent un autre paquet suivant le tableau ci-dessous.

	paquet extrait	paquet rendu
Processeur de traitement	COM COM	RES NIA
Processeur programme	NIA	INC
Processeur données	INC RES	COM NIA

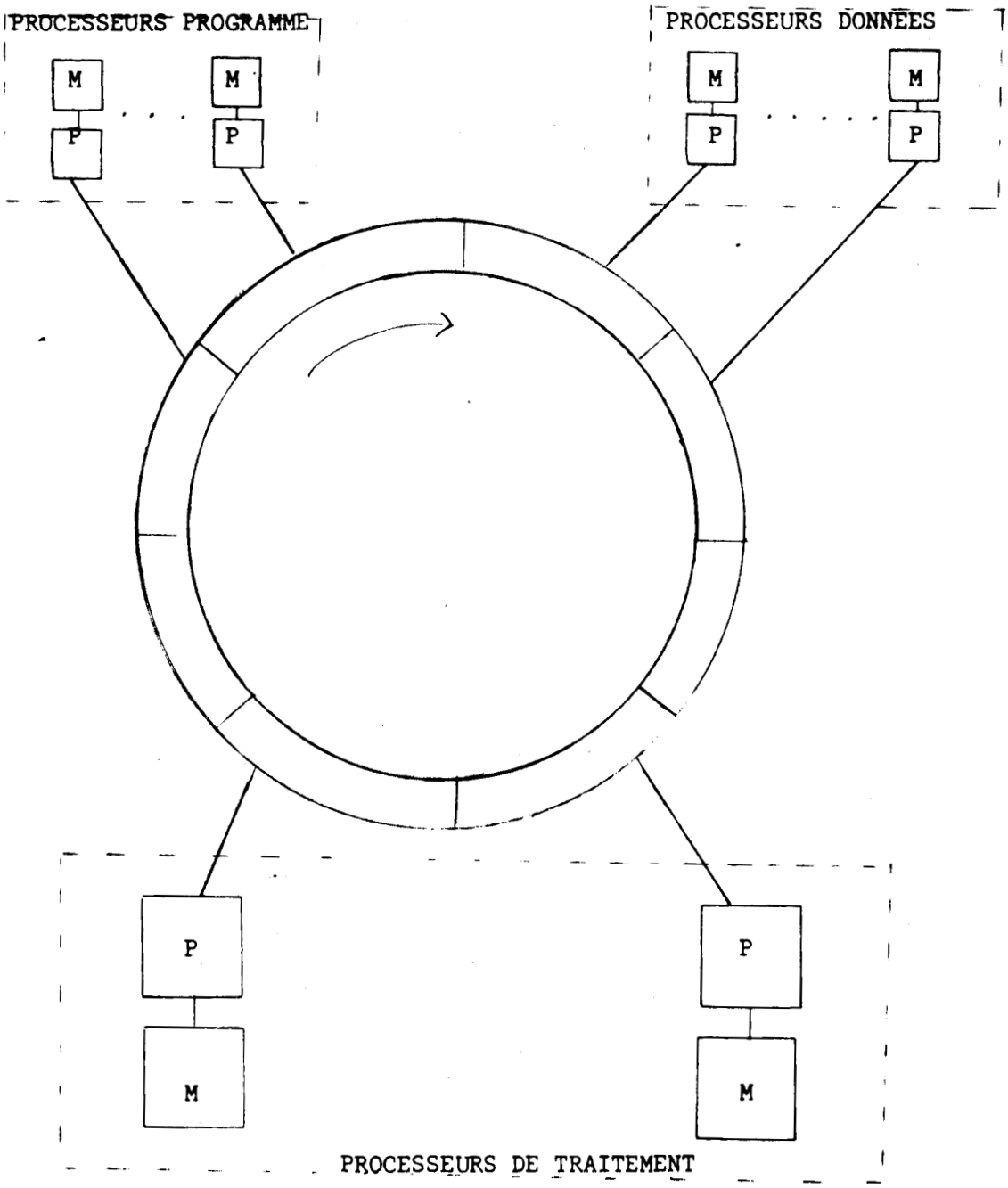


Figure 10. : Architecture GCF implémentée.

Il existe beaucoup d'analogies entre cette implémentation et celle que nous proposerons pour notre modèle. Pourtant ces deux études sont parties de deux idées différentes et ont été menées séparément. Le fait qu'elles aient abouti à une solution dont les principes sont identiques semble indiquer que l'utilisation d'un anneau en tant qu'outil de communication est très bien adaptée à un haut degré d'asynchronisme dans la réalisation de multiprocesseurs.

3. CONCLUSION

On notera à travers les projets assez divers analysés ici quelques grands traits communs autour desquels s'articulent toutes les approches du traitement dirigé par les données.

En premier lieu, l'intérêt d'une représentation par graphe servant soit de modèle soit même de support d'expression d'un traitement.

En second lieu, la nécessité d'une gestion de mémoire originale par rapport aux organisations classiques. Bien que conduisant à des solutions différentes, toutes les études concernant le traitement dirigé par les données conduisent à donner la plus grande importance à des relations internes et à s'affranchir de l'adressage. Beaucoup trouveraient un grand avantage à employer des supports associatifs.

Enfin, la notion d'assignation unique, bien que non directement liée au traitement dirigé par les données, apparait comme un facteur de normalisation et de simplification du contrôle. Plus qu'une règle de rédaction, elle fait apparaître l'avantage que l'on trouve à donner un nom spécifique à toute nouvelle valeur, un nom ne représentant pas un emplacement mémoire. Dans ce sens, il s'agit d'un principe très naturel et très fiable. L'espace des noms devient alors aussi simple et facile à gérer que l'espace des adresses d'une machine usuelle.

Par ailleurs, les architectures dirigées par les données apportent une grande contribution à l'exploitation du parallélisme. Ceci parce qu'elles sont capables de détecter à l'exécution ce qui peut s'exécuter en parallèle dans un traitement. Il est possible d'espérer qu'elles finiront par exploiter tout le parallélisme existant dans un traitement.

Cependant, le parallélisme est exploité en général au niveau des instructions d'un programme. Cela implique un grand volume de communications dans le système, les instructions, et même quelquefois leurs opérandes, étant communiqués séparément.

Or, les circuits LSI devenant de moins en moins chers, il semble naturel de ne plus utiliser des opérateurs élémentaires auxquels il faut fournir une instruction à la fois, mais des opérateurs évolués capables d'exécuter un ensemble d'instructions ou un module de façon complètement autonome sans avoir recours à aucune autre unité du système.

Si les opérateurs ne communiquent avec le système que pour rechercher un module à exécuter et fournir des résultats lorsque l'exécution de ce module est terminée, le volume de communications diminue fortement. En même temps que la réduction du volume de communication, l'emploi des circuits LSI, en particulier des microprocesseurs, conduit à une réalisation particulièrement simple.

Dans ce cadre, un programme devient un ensemble de modules plutôt qu'un ensemble d'instructions. Le découpage en modules est préalable au traitement. Le but étant d'exécuter les modules en parallèle, il faut maximiser leur indépendance et donc minimiser leur couplage. Or le couplage minimal entre modules est obtenu lorsque toutes les informations reçues et toutes les informations retournées par un module sont passées comme paramètres. Si la règle d'Assignation Unique est appliquée aux paramètres de communication entre modules, le parallélisme existant entre les modules s'exprimera naturellement. Et en conséquence, l'utilisation d'un contrôle dirigé par les données s'imposera pour l'exécution d'un tel ensemble de modules dans une machine parallèle.

Ce sont ces remarques qui ont conduit à la définition d'un modèle, basé sur le découpage d'un programme en modules que nous appelons BLOCS, et utilisant le concept d'Assignation Unique.

CHAPITRE II

LE MODELE STATIQUE

0. INTRODUCTION

Dans les chapîtres qui suivent, nous nous attachons à décrire MAUD, (Machine d'Assignation Unique Dynamique). C'est une machine parallèle basée sur une exécution de type dirigée par les données et s'appuyant sur une structuration des programmes en entités appelées BLOCS. La notion de BLOC utilisée dans MAUD n'est pas celle des langages de programmation classiques : elle en diffère à la fois par la structure d'un BLOC et par la façon dont il est exécuté. La règle d'Assignation Unique est appliquée au niveau des BLOCS. Le modèle comporte un certain nombre de processeurs de traitement et un ou plusieurs processeurs spécialisés entre lesquels il n'existe aucun moyen direct de communication.

Pour être exécutable dans MAUD un programme doit être décomposé en BLOCS. Un BLOC est constitué d'un ensemble d'instructions et de l'ensemble des objets utilisés par ces instructions. Il peut être considéré comme une instruction généralisée ayant plusieurs opérandes-données et plusieurs opérandes-résultats. Les opérandes-données sont les opérandes d'entrée du BLOC ; ils sont calculés par d'autres BLOCS. L'exécution d'un BLOC fournit des résultats, ou opérandes de sortie du BLOC qui peuvent être utilisés pour l'exécution d'autres BLOCS.

Le modèle que décrit ce chapître II est relativement simple et s'apparente aux modèles dirigés par les données habituels. Les notions qui sont utilisées pour les mécanismes de gestion des BLOCS et de transmission entre BLOCS pourront paraître complexes par rapport à la simplicité du modèle. L'introduction d'un certain nombre d'éléments dans les BLOCS ne sont pas réellement nécessaires pour décrire ces mécanismes, mais ils ont été introduits parce que le modèle possède des limites : en effet, l'écriture d'un programme sous forme de BLOCS comme nous les définissons ci-après n'est pas toujours possible ; de plus, une bonne exploitation des possibilités de parallélisme du système est rendue difficile parce que le découpage en BLOCS doit être fait préalablement à l'exécution du programme.

Ce découpage est donc fort statique et l'exécution d'un programme l'est aussi. En conséquence, ce modèle sera appelé modèle statique.

Pour supprimer ces problèmes, un autre modèle a été défini qui permet de donner un comportement dynamique à l'exécution d'un programme, mais qui nécessite l'utilisation d'informations particulières dans les BLOCS. Ces informations ne sont connues et utilisées qu'au niveau de modèle, et complètement transparentes au programmeur. Nous les avons utilisées pour décrire le modèle statique. Cela complique légèrement la description du modèle mais permet de mieux distinguer par la suite les mécanismes mis en oeuvre pour donner le caractère dynamique des mécanismes de gestion des BLOCS et de transmission entre BLOCS, qui sont fondamentaux dans MAUD.

1. ASSIGNATION UNIQUE PAR BLOCS

Un programme pour MAUD se compose d'un certain nombre d'entités appelées BLOCS. Un BLOC est constitué par un ensemble d'instructions et l'ensemble des objets utilisés par ces instructions. Les instructions d'un BLOC ne peuvent communiquer avec les instructions d'un autre BLOC qu'à travers un nombre limité d'objets :

* des *objets d'entrée* qui conditionnent l'exécution du BLOC ;

* des *objets de sortie* qui sont calculés par les instructions du BLOC et utilisés comme *objets d'entrée* par d'autres BLOCS.

La règle d'Assignation Unique [TE 68] est appliquée à l'échelle des BLOCS pour tous les objets transmis entre les BLOCS, c'est-à-dire qu'un *objet de sortie* peut recevoir au plus une valeur pendant toute l'exécution du programme.

Le respect de l'Assignation Unique à ce niveau permet :

1) d'exprimer naturellement les dépendances qui existent entre les différents BLOCS d'un programme, et par conséquent d'en déduire un parallélisme potentiel d'exécution.

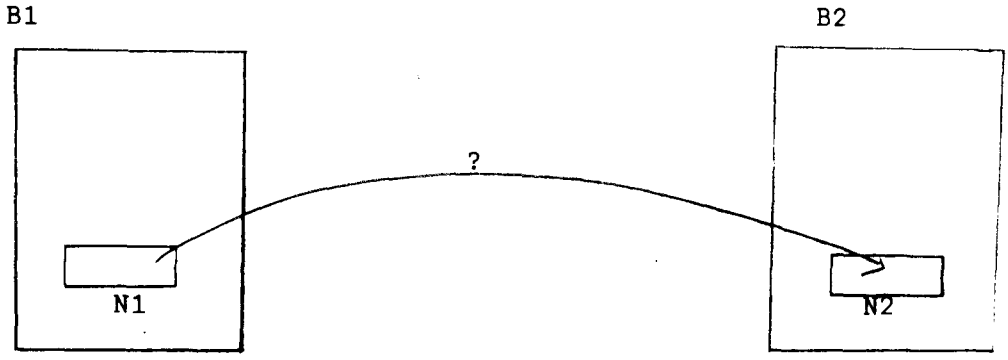
2) d'utiliser un mécanisme de type "contrôle dirigé par les données" pour l'exécution d'un programme : un BLOC ne devient prêt à exécuter que lorsque tous ses *objets d'entrée* ont reçu une valeur.

L'Assignation Unique au niveau des BLOCS est indispensable : elle permet d'assurer un comportement déterministe à l'exécution d'un programme puisque, lorsqu'un *objet de sortie* a été calculé, on est certain qu'il ne peut plus être modifié. Aucun autre contrôle que le contrôle dirigé par les données n'est nécessaire.

En conclusion, un programme exécutable est écrit en Assignation Unique par BLOCS.

2. COMMUNICATION ENTRE BLOCS

Les communications entre BLOCS se font par l'intermédiaire des *objets d'entrée* et *objets de sortie*. Le contrôle de l'exécution du programme découpé en BLOCS consiste donc à transmettre à un BLOC B1 l'objet calculé par un BLOC B2.



Dans le BLOC B1, l'objet est désigné par un nom N1 ; dans B2, il est désigné par un nom N2. Le BLOC B1 ne connait pas les noms propres au BLOC B2, et réciproquement. La transmission n'est donc pas possible directement. Elle peut se faire en attribuant à chaque *objet de sortie* un nom particulier qui ne servira qu'à faire la communication d'un objet entre deux BLOCS. Ces noms particuliers sont appelés *noms de communication* puisqu'ils sont utilisés pour permettre les communications entre BLOCS. Ces noms ne sont connus qu'à l'échelle des BLOCS, inconnus à l'intérieur des BLOCS. Les noms utilisés à l'intérieur des BLOCS, et inutilisables à l'extérieur, sont appelés des *noms internes*.

Les *noms de communication* pourraient être donnés par le programmeur ou un préprocesseur. Cependant, si les BLOCS sont écrits indépendamment les uns les autres, un générateur de *noms de communication* est nécessaire.

Il faut remarquer que les *noms de communication* ne désignent pas des emplacements mémoire explicites, mais servent simplement à désigner des valeurs.

Les *noms de communication* sont gérés par l'intermédiaire d'ensembles que nous appelons des *domaines*.

Définition

* un *domaine* est un ensemble de couples (N_i, V_i) ou (N_i, *non-évalué*) où :
  - N_i désigne un nom,
  - V_i désigne une valeur appartenant à l'ensemble des valeurs utilisables dans MAUD
  - et *non-évalué* signifie qu'aucune valeur n'a été associée au nom N_i.

La valeur *non-évalué* n'a de signification qu'au niveau de MAUD et ne peut jamais être utilisée à l'intérieur d'un BLOC car elle ne peut être fournie au BLOC. Elle n'a pas de signification pour le programmeur. Il sera cependant nécessaire de disposer pour les *noms internes* d'une valeur particulière *non-affecté* :

non-affecté a une signification au niveau du programme
non-évalué a une signification au niveau de la gestion des BLOCS.

Définitions

* un *domaine d'entrée* est un *domaine* où les noms N_i sont les *noms de communication* associés aux *objets d'entrée* du BLOC.

* un *domaine de sortie* est un *domaine* où les noms N_i sont les *noms de communication* associés aux *objets de sortie* du BLOC.

* un *domaine d'entrée* est dit *complet* lorsque chaque *nom de communication* qui y figure est associé à une valeur différente de *non-évalué*.

* un *domaine d'entrée* vide est *complet*.

* un *domaine propre* est un ensemble de couples (NI_j, V_j) où
  . NI_j désigne un *nom interne*
  . V_j désigne la valeur associée à ce nom, ou *non-affecté*.

Un *domaine* peut être vide, c'est-à-dire ne comporter aucun couple. Un BLOC ayant un *domaine d'entrée* vide est un BLOC dont l'exécution ne dépend pas de l'exécution d'autres BLOCS. Ce sera le cas de tous les BLOCS permettant de commencer l'exécution d'un programme. Un BLOC ayant un *domaine de sortie* vide est un BLOC ne fournissant aucun résultat. Cela n'a pas de signification dans le modèle statique, sauf pour des BLOCS particuliers permettant des échanges avec l'extérieur du système.

Un *nom de communication* ne peut figurer que dans un *domaine d'entrée* ou dans un *domaine de sortie*. De plus, il ne peut figurer que dans un seul *domaine de sortie* à cause du respect de la règle d'Assignation Unique au niveau des BLOCS, l'assignation de valeurs aux *noms de communication* se faisant lors de la production du *domaine de sortie* à la fin de l'exécution d'un BLOC.

Les *noms de communication* ne servent qu'à transmettre des valeurs entre BLOCS (mais jamais pendant l'exécution d'un BLOC). Dans le modèle statique, il serait possible d'utiliser ces noms à l'intérieur des BLOCS (à condition de ne pas les réutiliser pour les *objets de sortie*) s'ils sont attribués par le programmeur ou un préprocesseur. Dans le modèle dynamique, cela ne sera plus possible, le même BLOC pouvant être utilisé plusieurs fois, et donc avec un *domaine d'entrée* contenant des *noms de communication* différents à chaque utilisation du BLOC.

3. STRUCTURE D'UN BLOC

Un BLOC est constitué de 3 parties :

1) une partie "communication" réservée à MAUD et qui contient :
   . le *domaine d'entrée*
   . le *domaine de sortie*

2) une partie interne décrite par le programmeur contenant :
   . un ensemble d'instructions
   . les objets utilisés par ces instructions. A chaque objet est associé un *nom interne*. Ils sont regroupés dans un *domaine propre*.

3) une partie servant d'interface entre la partie interne et la partie communication et permettant de faire les transmissions de valeurs entre la partie communication et la partie interne. Elle sera constituée par une *liste de correspondance*. Cette partie sera décrite en même temps que le traitement d'un BLOC par un opérateur.

4. PRINCIPES DE FONCTIONNEMENT

Un BLOC apparaît comme un ensemble indivisible d'instructions qui est exécuté lorsque son *domaine d'entrée* est *complet* et qui produit un *domaine de sortie*.

Pendant l'exécution d'un programme, il y a trois sortes de BLOCS :

1) des BLOCS qui sont en cours d'exécution ;

2) des BLOCS dont le *domaine d'entrée* est *complet*, mais qui n'ont pas encore été exécutés ; ces BLOCS sont des *BLOCS exécutables* et forment l'ensemble X ; dans un modèle théorique, cet ensemble ne devrait pas exister : un BLOC peut être exécuté dès qu'il est *exécutable*. Pratiquement cela n'est jamais vrai, le nombre d'opérateurs utilisés n'étant jamais illimité. C'est pourquoi l'ensemble X figure dans le modèle.

3) des BLOCS dont le *domaine d'entrée* n'est pas *complet*. Ces BLOCS sont des *BLOCS en attente* et forment l'ensemble A.

Un autre ensemble, l'ensemble S, reçoit les *domaines de sortie* produits par l'exécution complète d'un BLOC. Un *domaine de sortie* n'étant pas, le plus souvent, transmis tout entier à un autre BLOC, les couples qui y figurent sont utilisés un par un. S désigne donc l'ensemble des *objets de sortie*.

L'exécution des BLOCS est réalisée par des opérateurs de traitement évolués, de type processeur, dont la fonction est l'exécution d'un BLOC et le rangement du *domaine de sortie* dans l'ensemble S lorsque l'exécution du BLOC est complètement terminée. A ce stade, un BLOC est bien considéré comme une instruction généralisée en ce sens qu'on ne considère pas les différentes opérations effectuées par l'opérateur pour aboutir à l'exécution complète du BLOC.

Un opérateur de mise-à-jour, l'opérateur MAJ, est chargé de faire les communications d'objets entre les BLOCS : il distribue les *objets de sortie* de l'ensemble S aux *BLOCS en attente* de ces objets dans l'ensemble A, et transfère les *BLOCS en attente* dont le *domaine d'entrée* est *complet* de l'ensemble A vers l'ensemble X.

Le parallélisme résulte de l'existence de plusieurs opérateurs de traitement capables de fonctionner simultanément. Fonctionnellement, l'opérateur de mise à jour est unique. Mais il n'y a aucune difficulté à multiplier les sites de mise-à-jour si l'équilibre des tâches l'exige. L'utilisation de l'assignation unique trouve dans cette multiplication une nouvelle justification.

Remarque

Au lieu de transmettre la valeur d'un *objet de sortie* vers un *domaine d'entrée*, nous aurions pu, comme dans la machine LAU, ne transmettre que la connaissance de la valeur de cet objet. Nous avons préféré une solution du type de celle adoptée par Rumbaugh pour sa machine dirigée par les données et qui consiste à donner à l'opérateur d'exécution un "paquet" contenant l'instruction et ses opérandes.

Dans MAUD, dès qu'un opérateur a pris en charge un BLOC, il peut commencer sont exécution, sans avoir à faire référence à un ensemble quelconque, ce qui semble apporter une optimisation dans l'utilisation des opérateurs.

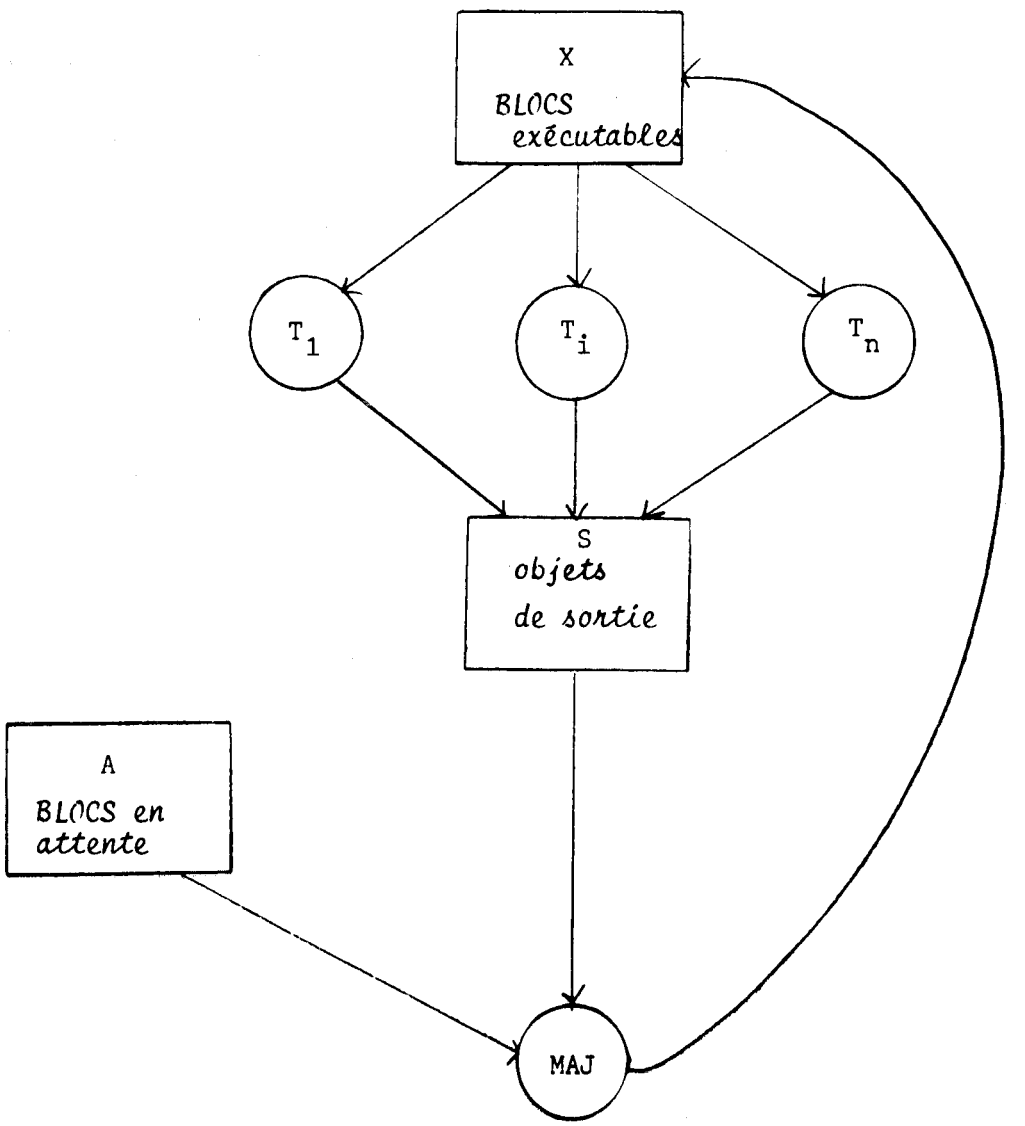


Figure 1. : Le modèle statique.

Conventionnellement, sur ce schéma et sur tous les autres schémas, un rond représente un opérateur, et un rectangle représenteun ensemble.

4.1. FONCTION DES OPÉRATEURS

4.1.1. OPERATEUR DE TRAITEMENT T_i

La fonction de l'opérateur T_i est l'exécution d'un BLOC. Ce traitement nécessite l'utilisation implicite de deux primitives : la primitive début-de-bloc de prise en compte d'un BLOC, et la primitive fin-de-bloc de fin d'exécution d'un BLOC. Ces primitives ont pour principal rôle de faire des transmissions de valeurs de la partie interne vers la partie communication et vice-versa.

Pour cela, il faut, dans chaque BLOC, un intermédiaire permettant le passage de l'objet désigné par un *nom de communication* à l'extérieur du BLOC à l'objet désigné par un *nom interne* à l'intérieur du BLOC. L'utilisation d'une *liste de correspondance* va nous le permettre.

le même *nom de communication* NC doit alors figurer dans le *domaine de sortie* de B1 et dans le *domaine d'entrée* de B2.

Définition

* une *liste de correspondance* est un ensemble de couples (NC_j, NI_j) où :

- NC_j désigne un *nom de communication*
- NI_j désigne un *nom interne*.

Chaque *liste de correspondance* est composée de deux sous-ensembles :

- LCE, *liste de correspondance d'entrée*
- LCS, *liste de correspondance de sortie*

De plus, il faut remarquer que :

1) card (*liste de correspondance*) =
card (*domaine d'entrée*) + card (*domaine de sortie*).

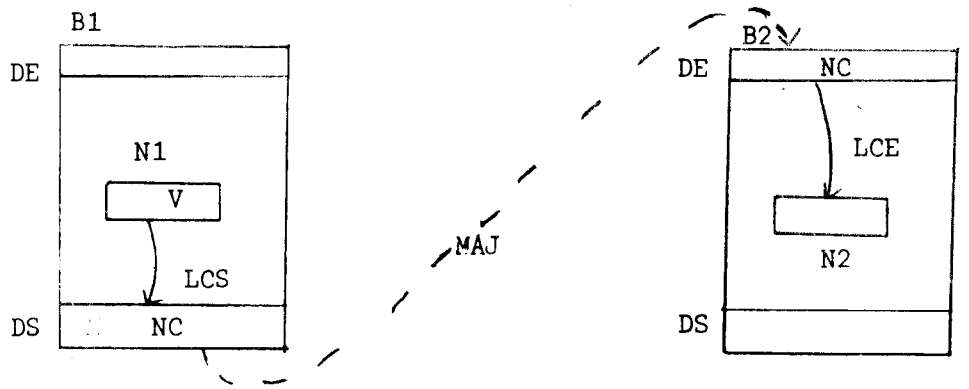
2) Il n'existe qu'une et une seule *liste de correspondance* par BLOC.

Définitions

* une *liste de correspondance d'entrée*, ou LCE, est un ensemble de couples (NC_j, NI_j) permettant de mettre en correspondance un *nom interne* NI_j avec un *objet d'entrée* désigné par le *nom de communication* NC_j.

* une *liste de correspondance de sortie*, ou LCS, est un ensemble de couples (NC_j, NI_j) permettant de mettre en correspondance un *nom de communication* NC_j avec un objet de *nom interne* NI_j et calculé par le BLOC.

La *liste de correspondance* d'un BLOC permet donc de faire la transmission d'une valeur désignée par un *nom interne* vers un *nom de communication* et vice-versa.



C'est l'opérateur MAJ précédemment décrit qui fait la transmission du couple (NC,V) du *domaine de sortie* de B1 vers le *domaine d'entrée* de B2.

Remarque

La transmission des valeurs au début et à la fin de l'exécution d'un BLOC pourrait se faire par position en réduisant la *liste de correspondance* à une liste de *noms internes*. L'utilité des couples (NC, NI) apparaitra clairement dans la partie dynamique du modèle : c'est un moyen de contrôle et de protection pour les mauvaises utilisations des noms en général.

Relations entre les différents constituants d'un BLOC.

Entre les différents constituants d'un BLOC, deux relations sont toujours vérifiées :

Soient DE le *domaine d'entrée* d'un BLOC
DS le *domaine de sortie*
LCE la *liste de correspondance d'entrée*
LCS la *liste de correspondance de sortie*
DP le *domaine propre*

R1 * pour chaque couple $(NC_j, V_j) \in DE$, il existe un couple (NI_k, V_k) et un seul $\in DP$ et un couple (NC_j, NI_k) et un seul $\in LCE$.

R2 * pour chaque couple $(NC_j$, *non-évalué*$) \in DS$, il existe un couple (NI_k, V_k) et un seul $\in DP$ et un couple (NC_j, NI_k) et un seul $\in LCS$.

Normalement, lorsque la liste de *correspondance de sortie* est utilisée, la valeur V_k doit être différente de *non-affecté*. Il se pourrait cependant qu'elle soit égale à *non-affecté* : cela aurait sans doute une signification par rapport au programme en cours d'exécution et peut être n'être pas considéré comme une erreur. Cependant il doit rester possible de détecter une telle situation.

Les définitions et relations précédentes permettent de détailler le fonctionnement des primitives début-de-bloc et fin-de-bloc.

1. primitive début-de-bloc

Cette primitive effectue la transmission des *objets d'entrée* se trouvant dans la partie communication du BLOC vers la partie interne du BLOC.

Compte-tenu de la relation R1 précédente existant entre le *domaine d'entrée* DE, la *liste de correspondance d'entrée* LCE et le *domaine propre* DP, le traitement réalisé par la primitive début-de-bloc est le suivant :

$$DP = DP - \{(NI_k, V_k) \mid (NC_j, NI_k) \in LCE\} + \{(NI_k, V_j) \mid (NC_j, V_j) \in DE\}$$

la valeur V_k devrait ici être *non-affecté* ; mais dans le modèle dynamique, elle pourra être quelconque.

2. primitive fin-de-bloc

Cette primitive effectue la transmission des *objets de sortie* de la partie interne du BLOC vers la partie communication et range le *domaine de sortie* ainsi modifié dans l'ensemble S.

Compte-tenu de la relation R2 précédente existant entre le *domaine de sortie* DS, la *liste de correspondance de sortie* LCS et le *domaine propre* DP, le traitement réalisé par la primitive fin-de-bloc est le suivant :

$DS = DS - \{(NC_j,$ *non-évalué*$) \mid (NC_j, NI_k) \in LCS\}$
$\quad + \{(NC_j, V_k) \mid (NC_j, NI_k) \in LCS$ et $(NI_k, V_k) \in DP\}$

$S = S + DS$

3. traitement réalisé par l'opérateur T_i

L'opérateur T_i extrait un *BLOC exécutable* de l'ensemble X. Puis il exécute la primitive début-de-bloc. Ensuite il exécute la partie interne du BLOC. Lorsqu'il a complètement terminé cette exécution, il exécute la primitive fin-de-bloc. Puis il recherche un autre *BLOC exécutable* dans l'ensemble X. S'il n'y en a pas, il continue sa recherche jusqu'à ce qu'il en trouve un ; sinon, s'il y en a au moins un, il l'extrait de l'ensemble X.

4.1.2. L'OPERATEUR DE MISE-A-JOUR (MAJ)

L'opérateur MAJ est chargé de 2 tâches :

1) mettre à jour les *domaines d'entrée* des *BLOCS en attente* se trouvant dans l'ensemble A, à l'aide des couples (NC_j, V_j) se trouvant dans l'ensemble S, c'est-à-dire :

Si n est le nombre de BLOCS se trouvant dans l'ensemble A au moment de la mise-à-jour, et DE_i, i = 1,..., n les *domaines d'entrée* de ces BLOCS B_i,

Si $(NC_j, V_j) \in S$ et $(NC_j,$ *non-évalué*$) \in DE_i$
alors $S = S - (NC_j, V_j)$
$\quad DE_i = DE_i - (NC_j,$ *non-évalué*$) + (NC_j, V_j)$
et ceci pour chaque *objet de sortie* (NC_j, V_j) se trouvant dans S.

2) la deuxième tâche de l'opérateur MAJ est de supprimer de l'ensemble A les *BLOCS en attente* dont le *domaine d'entrée* est *complet* et d'ajouter ces BLOCS devenus *exécutables* à l'ensemble X, c'est-à-dire :

Si DE_i est *complet* alors $A = A - B_i$
$X = X + B_i$

Contrainte d'utilisation des *objets de sortie*

Afin de faciliter le travail de l'opérateur MAJ, et surtout, afin d'éviter les problèmes de taille de l'ensemble S, qui, dans un modèle, est potentiellement infinie, mais qui, dans une réalisation pratique, sera toujours limitée, nous avons introduit une contrainte quant à l'utilisation des objets apparaissant dans les *domaines de sortie* : ils ne peuvent être utilisés qu'une fois. C'est-à-dire :

$\forall (NC_j, V_j) \in S$, il existe un et un seul *BLOC en attente* B_i de *domaine d'entrée* DE_i tel que $(NC_j$, *non-évalué*$) \in DE_i$.

En effet, si l'utilisation unique n'était pas imposée, le seul moyen de savoir si un *objet de sortie* peut être enlevé de l'ensemble S est de connaître son nombre de références. Ce qu'il est possible de connaître par un pré-examen dans un traitement statique, mais qui ne le sera plus dans un traitement dynamique : dans ce cas, à tout moment, ce nombre de références est susceptible de changer. Il faudrait alors disposer d'organisations particulières de mémoire où la gestion d'un compteur de références serait prise en charge par la mémoire elle-même [Leg 78].

Remarque

L'opérateur MAJ n'opère que sur les *domaines d'entrée* et *domaines de sortie*. Fonctionnellement, il est donc possible de séparer l'ensemble A en deux sous-ensembles dont les éléments sont en correspondance biunivoque. En effet, il suffit de décomposer chaque BLOC en deux parties :

- le *domaine d'entrée* du BLOC
- le BLOC sans son *domaine d'entrée*

Le schéma fonctionnel devient celui de la figure 2 où :

- DENC est l'ensemble des *domaines d'entrée* des *BLOCS en attente*, par conséquent non *complets* ;

- BLOCSSANSDE est l'ensemble des *BLOCS en attente* privés de leur *domaine d'entrée* ;

- DEC est l'ensemble des *domaines d'entrée complets* obtenus après action de l'opérateur MAJDE ;

- MAJDE est l'opérateur de mise-à-jour MAJ décrit ci-dessus mais dont la fonction est limitée à la première tâche.

- REUNION est un opérateur extrêmement simple qui, à partir des *domaines d'entrée complets* et des éléments de l'ensemble BLOCSANSDE reconstituent un *BLOC exécutable* qu'il ajoute à l'ensemble X. Il réalise en réalité la deuxième tâche de l'opérateur MAJ précédemment décrit.

Par la suite, nous ne considérerons que le modèle de la figure 1.

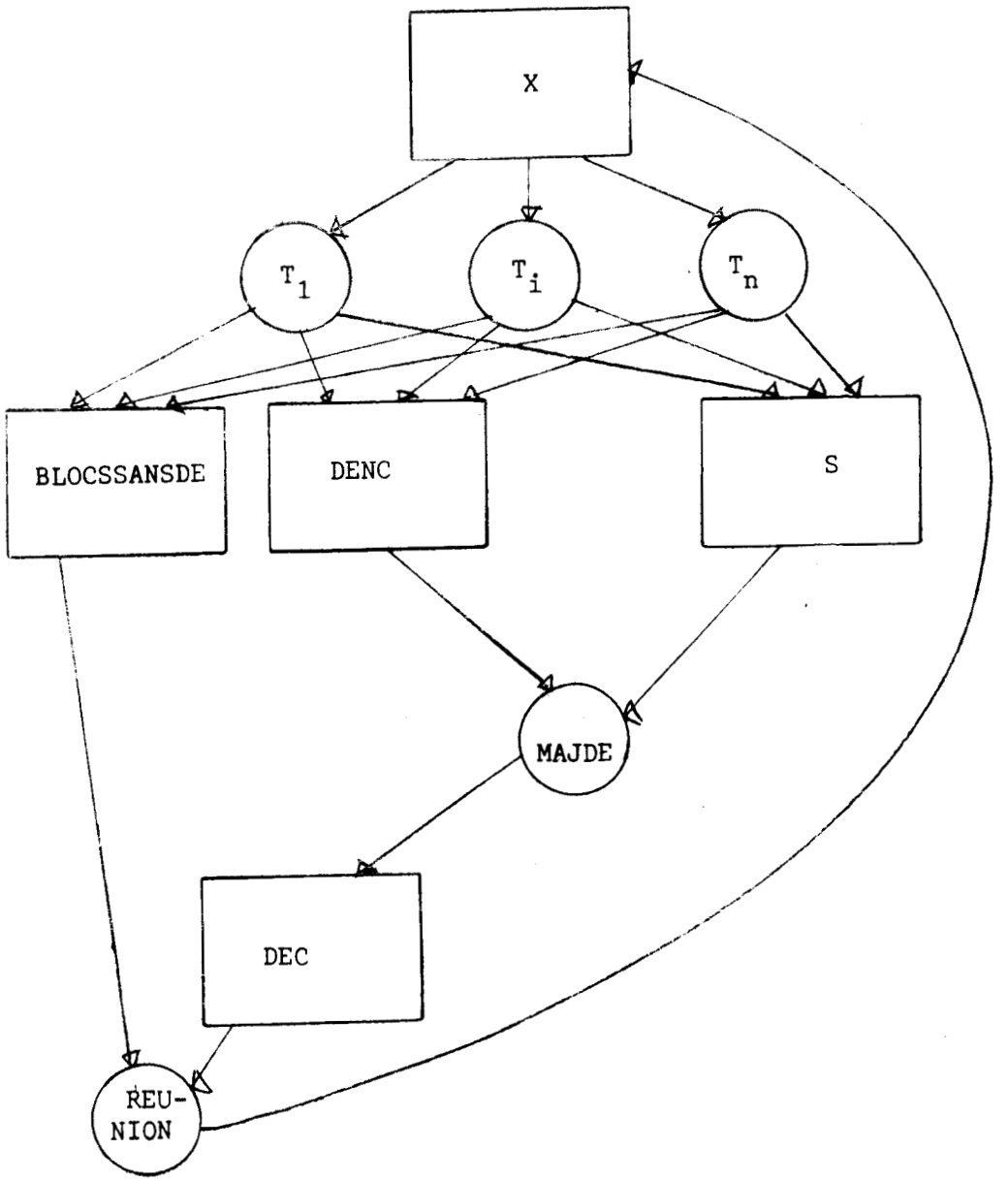


Figure 2. : cas où les *domaines d'entree* sont séparés des BLOCS.

4.2. FONCTIONNEMENT GÉNÉRAL DU SYSTEME

Au début d'un traitement, les BLOCS constituant ce traitement se trouvent :

- soit dans l'ensemble X, si leur *domaine d'entrée* est *complet*, s'ils sont donc *exécutables*,

- Soit dans l'ensemble A, si leur *domaine d'entrée* n'est pas *complet*, s'ils sont *en attente*.

Chaque opérateur de traitement extrait un *BLOC exécutable* de l'ensemble X. Lorsque l'exécution de ce BLOC est complètement terminée, il ajoute à l'ensemble S le *domaine de sortie* ainsi évalué. Puis il recherche un nouveau *BLOC exécutable* dans X.

L'opérateur MAJ extrait un *objet de sortie* (NC_i, V_i) de l'ensemble S, et met à jour dans l'ensemble A le *BLOC en attente* de ce couple. Simultanément, lorsqu'il trouve un *BLOC en attente* dont le *domaine d'entrée* est *complet*, il l'extrait de l'ensemble A et l'ajoute à l'ensemble X.

Un BLOC ne pouvant être exécuté que lorsque son *domaine d'entrée* est *complet* l'exécution d'un traitement est bien de type "dirigé par les données".

Le traitement est en principe terminé lorsque tous les *BLOCS en attente* sont devenus *exécutables* et ont été exécutés par les opérateurs de traitement. Dans ce cas, les ensemble A et X sont vides et l'ensemble S contient les résultats du traitement. Cependant, il se peut qu'à un moment donné l'ensemble X soit vide sans que l'ensemble A le soit. A ce moment, deux cas peuvent se présenter :

- les opérateurs de traitement, ou un certain nombre d'entre eux, sont actifs. Dans ce cas, il sont probablement en train d'exécuter des BLOCS dont le *domaine de sortie* permettra, par l'action de l'opérateur MAJ, de transférer les BLOCS qui sont dans l'ensemble A vers l'ensemble X ; le traitement n'est pas terminé et doit vraisemblablement se poursuivre.

- les opérateurs de traitement sont tous inactifs. Dans ce cas, il existe certainement une erreur de programmation. En effet, lorsqu'un BLOC est terminé, tous les objets de son *domaine de sortie* sont délivrés, puis sont utilisés par l'opérateur MAJ pour compléter les *domaines d'entrée* des *BLOCS en attente*. Si le *domaine d'entrée* d'un *BLOC en attente* n'est pas *complet* et qu'il n'y a aucun BLOC en cours d'exécution, cela ne peut être qu'une erreur de programmation.

Un certain nombre de remarques peuvent être faîtes en ce qui concerne le fonctionnement et l'utilisation du modèle :

1) au début d'un traitement, il n'est pas indispensable que la répartition en *BLOCS en attente* et *BLOCS exécutables* soit déjà faite : tous les BLOCS constituant un traitement peuvent se trouver dans l'ensemble A, et c'est l'opérateur MAJ qui se chargera d'en extraire les *BLOCS exécutables* et de les ranger dans l'ensemble X.

Cela aurait d'ailleurs un autre intérêt : puisqu'au début d'un traitement, il faudra bien introduire les BLOCS dans le système, il suffira de constituer un seul ensemble, l'ensemble A.

2) dans le modèle tel qu'il a été défini jusqu'à présent, le découpage en BLOCS d'un programme quelconque peut être fait par l'utilisateur. Les éléments ajoutés à la partie interne (instructions + objets utilisés) d'un BLOC, tels que *domaines* et *listes de correspondance*, ne se justifient vraiment que par ce qui sera décrit dans le chapitre suivant. Il suffit que l'utilisateur respecte bien l'Assignation Unique pour les *objets de sortie* des BLOCS. Ceci n'est possible bien entendu que parce que le traitement est complètement statique, et que par conséquent tout est parfaitement connu dès le début du traitement.

3) aucune contrainte n'est donnée quant à la taille des BLOCS. Pour un traitement sur une machine réelle, il est évident qu'il faudra se donner une limite, ce qui ne nous permettra plus d'écrire n'importe quel programme sous la forme acceptable par MAUD. Le modèle que nous venons de décrire a donc des limites pratiques, bien que, pour une bonne modularité, il soit souhaitable de ne pas avoir de BLOCS de taille trop importante.

4) il n'existe dans le modèle aucun problème d'encombrement dus aux objets puisqu'un élément d'un ensemble utilisé par un opérateur disparaît de cet ensemble.

CHAPITRE III

LE MODELE DYNAMIQUE

0. INTRODUCTION

Jusqu'ici le modèle ne présente pas de caractéristiques très différentes des modèles classiques dirigés par les données. Il est fonctionnellement identique à une machine dirigée par les données traitant des instructions en Assignation Unique, les BLOCS tenant lieu d'instructions. Son originalité vient de la notion de *domaines* et de *BLOCS*.

Or, le caractère statique du modèle ne permet pas une bonne adaptation de tous les programmes aux possibilités d'exécution parallèle. Tout est figé dès la phase d'écriture du programme en Assignation Unique par BLOCS. En effet :

* le graphe d'exécution du programme peut être connu avant l'exécution en recherchant les relations de dépendances entre BLOCS par examen de leurs paramètres d'entrée et de sortie ;

* le découpage en BLOCS représente une contrainte. Il ne peut être réalisé que pour un certain nombre de programmes, compte-tenu du fait que dans une réalisation matérielle la taille d'un BLOC sera toujours limitée.

Afin de limiter ce caractère statique, un second modèle a été étudié qui introduit un comportement dynamique au niveau de l'exécution d'un programme.

Dans cet autre modèle, appelé modèle dynamique, les BLOCS effectivement utilisés ne sont connus qu'au moment de l'exécution ; le graphe d'exécution des programmes n'est plus connu à l'avance.

Pour illustrer de façon plus précise ce qui a entraîné l'évolution du modèle précédemment décrit, nous allons donner quelques exemples de programmes pour le modèle statique qui vont permettre de mettre en évidence ses limites et ses inconvénients.

1. LIMITES ET INCONVENIENTS DU MODELE STATIQUE

A cette étape, le langage d'écriture d'un programme pour MAUD n'a pas besoin d'être complètement défini, pas plus que le langage d'écriture de la partie interne des BLOCS qui dépendra des choix définitifs faits pour la réalisation matérielle. Cependant, nous allons donner quelques éléments permettant d'écrire des programmes pour MAUD.

Dans la suite, un BLOC sera écrit en utilisant les notations suivantes :

```
BLOC nom-de-BLOC ;
entrée  liste-des-objets-d'entrée ;
sortie  liste-des-objets-de-sortie ;
local   liste-des-objets-locaux ;

                partie interne du BLOC

FIN BLOC
```

Le langage utilisé pour décrire la partie interne est de type algorithmique.

Un programme pour MAUD est constitué d'un ensemble de BLOCS écrits suivant le modèle précédent.

Pour commencer l'exécution d'un programme, il se peut que des valeurs d'entrée soient nécessaires. Bien que les échanges avec l'extérieur ne posent pas de problème particulier pour MAUD, ils ne seront évoqués qu'au moment où une architecture sera envisagée. En ce qui concerne les exemples de ce chapitre, les valeurs d'entrée d'un programme seront supposées être placées dans l'ensemble S des *objets de sortie* sous la forme normale des *objets de sortie* : (*nom de communication*, valeur).

Exemple 1

Soit à calculer CH(X) et SH(X)

Tout le calcul peut s'effectuer à l'intérieur d'un même BLOC BO :

Exemple 1.1.

```
BLOC BO ; entrée X ; sortie CH, SH ;
   local A, B ;
   si X = 1 alors CH : = 1 ;
                  SH : = 0

            sinon A : = exp (X) ;
                  B : = exp (-X) ;
                 CH : = (A + B)/2 ;
                 SH : = (A - B)/2

            fsi

FIN BLOC
```

Evidemment, cette rédaction ne permet pas d'exploiter les possibilités de parallélisme de MAUD. Il est préférable de l'écrire de la façon suivante :

Exemple 1.2.

```
BLOC B1; entrée X1 ; sortie RP ;
            RP : = si X1 = 0 alors 1 sinon exp (X1) fsi

FIN BLOC
```

```
BLOC B2 ; entrée X2 ; sortie RM;
            RM : = si X2 = 0 alors 1 sinon exp (-X2) fsi
FIN BLOC
```

```
BLOC B3 ; entrée RM, RP ; sortie CH, SH ;
          CH : = (RM + RP)/2 ;
          SH : = (RM - RP)/2

FIN BLOC
```

```
BLOC BO ; entrée X ; sortie  X1, X2 ;
          X1 : = X ;
          X2 : = X

FIN BLOC
```

Le BLOC BO est nécessaire à cause de la contrainte d'utilisation unique des données (§ II 4.1.2.). Il sert uniquement à dupliquer la valeur de X pour avoir deux *noms de communication* distincts.

Lorsque la valeur de X est placée dans l'ensemble S, le BLOC BO devient *exécutable* (grâce à l'intervention de l'opérateur MAJ). La production des valeurs de X1 et X2 autorise l'exécution des BLOCS B1 et B2 ; leur exécution pourra se faire en parallèle. Lorsque leurs paramètres de sortie seront élaborés, l'exécution du BLOC B3 pourra commencer.

Cette solution est meilleure que la première. Cependant elle nécessite la présence des quatre BLOCS dans MAUD, quelle que soit la valeur donnée à X. Or dans le cas où la valeur de X est égale à 1, il est très coûteux d'utiliser quatre opérateurs pour ce calcul. Cela est bien du au caractère statique qui ne permet pas la prise en compte des valeurs des paramètres pour une meilleure utilisation.

Exemple 2

Soit à effectuer le traitement suivant, où F est une fonction totale, longue à calculer.

Exemple 2.1.

```
BLOC BO ; entrée ; sortie  ....;
          local  .....;
          pour i jusqu'à 4 faire
           t [i] : = F(i)

          fin faire;
          autres-opérations-utilisant-t ;
FIN BLOC
```

La fonction F ne dépendant pas des valeurs du tableau t.

Afin de mieux exploiter les possibilités de parallélisme, il est possible d'écrire :

Exemple 2.2.

```
BLOC B1 ; entrée ; sortie t1 ;
        t1 : = F(1)
FIN BLOC
.
.
.
BLOC B4 ; entrée ; sortie t4 ;
        t4 : = F(4)
FIN BLOC

BLOC BO ; entrée t1, t2, t3, t4 ; sortie ... ;
        t[1] : = t1 ; t[2] : = t2 ;
        t[3] : = t3 ; t[4] : = t4 ;
        autres-opérations-utilisant-t;
FIN BLOC
```

Les BLOCS B1, B2, B3, B4 peuvent s'exécuter en parallèle. Mais cette forme présente l'inconvénient d'obliger à écrire quatre fois la fonction F.

Malheureusement, cette façon d'écrire le programme n'est plus possible si le nombre d'éléments de tableau à calculer n'est connu qu'au moment de l'exécution du programme : il faut réaliser tout le calcul à l'intérieur d'un même BLOC. Par exemple :

```
BLOC B ; entrée n ; sortie ..... ;
         pour i jusqu'à n faire
         t [i]: = F(i)
         fin faire ;
         autres-opérations-utilisant-t ;

FIN BLOC
```

Cela représente une très mauvaise utilisation du système, surtout si l'exécution du BLOC B conditionne la suite de l'exécution du programme : cela signifie qu'un certain nombre d'opérateurs resteront longtemps inactifs alors que le calcul de tous les éléments de t peut de façon evidente se faire en parallèle.

Le modèle décrit jusqu'ici ne permet donc pas une bonne adaptation de tous les programmes aux possibilités d'exécution parallèle. Afin de réaliser cette adaptation, il faut pouvoir faire varier le nombre de BLOCS exécutés en fonction des données de ce programme, donc dynamiquement lors de son exécution.

Certains programmes ne pourraient d'ailleurs jamais être traités par MAUD. En effet, dans une réalisation matérielle les opérateurs de traitement seront des processeurs, dont la mémoire propre sera limitée. Pour pouvoir exécuter un BLOC, il faut que ce BLOC tienne dans la mémoire propre d'un processeur, Si un programme comprend une boucle itérative dont le corps de boucle a une taille supérieure à la taille de la mémoire propre d'un processeur de traitement de BLOC, il ne peut être pris en charge sur MAUD.

2. INTRODUCTION DE PRIMITIVES NOUVELLES

Le caractère dynamique est introduit dans le modèle par l'utilisation d'une nouvelle primitive au niveau de l'exécution des BLOCS : la primitive execbloc.

La primitive execbloc permet d'exécuter un BLOC dont un modèle se trouve dans une BIBLIOTHEQUE de BLOCS. Dans une première approche, elle est analogue à un appel de procédure : elle appelle un nouveau BLOC B1 ; le BLOC BO en cours d'exécution est momentanément abandonné pour exécuter le BLOC B1. Puis on reprend l'exécution du BLOC BO.

Le programme de l'exemple 1 peut se réécrire :

Exemple 1.3.

```
BLOC BO ; entrée X ; sortie CH, SH ;
local A, B ;
          si X = 1 alors
                        CH : = 1 ;
                        SH : = 0
                   sinon
                        execbloc (B1 ; X ; A) ;
                        execbloc (B1 ; - X ; B) ;
                        CH : = (A + B)/2 ;
                        SH : = (A - B)/2
                   fsi
FIN BLOC
```

Le BLOC B1 se trouvant en BIBLIOTHEQUE fournit le calcul de EXP. Ce BLOC ne sera appelé que si la valeur de X placée dans l'ensemble S est différente de 1.

Le programme de l'exemple 2 peut se réécrire :

Exemple 2.3.

```
BLOC B0 ; entrée ... ; sortie ... ;
local ..... ;
      .
      .
      .
      pour i jusqu'à 4 faire
            execbloc (B1 ; i ; t[i])
            fin faire ;

      autres-opérations-utilisant-t ;

FIN BLOC
```

Le BLOC B1 se trouvant en BIBLIOTHEQUE fournit le calcul de la fonction-longue-à-calculer.

La primitive execbloc ainsi définie permet donc :

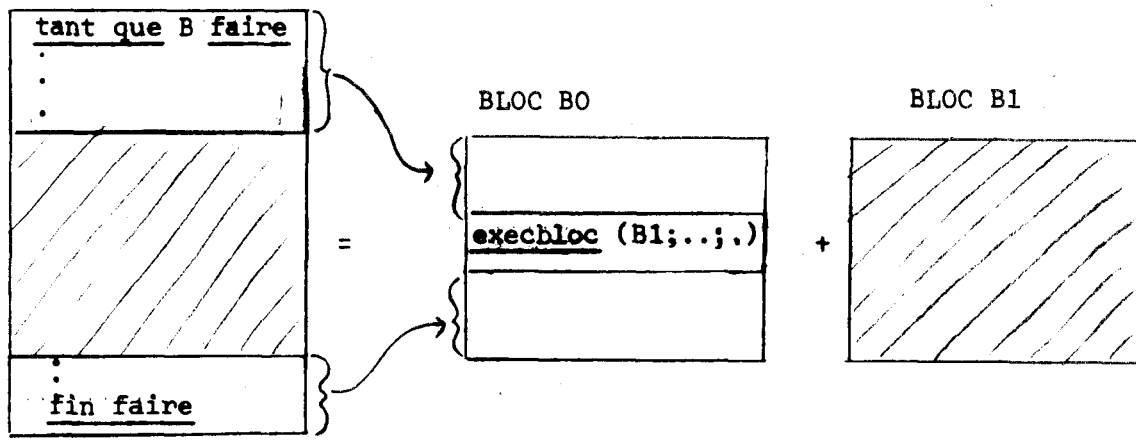
1) d'éviter l'écriture de plusieurs BLOCS identiques aux *objets d'entrée* et *objets de sortie* près (exemples 2.2. et 2.3.).

2) d'éviter l'écriture de BLOCS ne servant qu'à dupliquer une valeur (comme le BLOC B0 dans l'exemple 1.2.).

3) de n'introduire dans l'ensemble des *BLOCS exécutables* que les BLOCS réellement nécessaires, suivant les données du programme (exemple 1.3.).

4) d'exécuter des programmes non exécutables dans le modèle statique tels que ceux comportant une boucle itérative de taille très importante. Dans ce cas précis, le problème se résoud en transformant en BLOC de BIBLIOTHEQUE un (ou plusieurs) ensembles d'instructions de la boucle (convenablement choisis en s'aidant des structures de contrôle qui y figurent), et en remplaçant ces instructions par une primitive execbloc jusqu'à ce que la boucle itérative tienne complètement dans l'espace fixé pour un BLOC.

Exemple 3



Il faut remarquer que la transformation ci-dessus n'affecte pas la séquentialité d'exécution de la boucle itérative.

Dans tous les cas, les possibilités de parallélisme ne sont pas mieux exploitées. En effet, si la primitive execbloc n'est qu'analogue à un appel de procédure, dans les exemples 1.3. et 2.3., l'exécution du BLOC BO est abandonnée momentanément après l'exécution du premier execbloc pour exécuter le BLOC B1, puis reprise pour l'exécution du deuxième execbloc et abandonnée de nouveau pour l'exécution du BLOC B1 avec de nouveaux opérandes, et ainsi de suite. Une séquentialité totale est conservée au niveau du BLOC BO.

De plus, dans l'exemple 2.3., tous les éléments de t sont calculés séquentiellement, alors qu'ils pourraient l'être en parallèle.

Par conséquent, la primitive execbloc ne doit pas être un appel de procédure si l'on veut utiliser les possibilités de parallélisme. Elle doit être analogue à un FORK. C'est-à-dire qu'après l'exécution du premier execbloc, l'exécution du BLOC BO doit pouvoir se poursuivre concurremment avec l'exécution du BLOC B1. Et de même pour tous les BLOCS demandés par une primitive execbloc dans le BLOC BO : ils doivent pouvoir tous s'exécuter en parallèle et concurremment avec l'exécution du BLOC BO (ou une partie du BLOC BO).

Ainsi dans l'exemple 1.3., le BLOC BO et deux copies du BLOC B1 peuvent s'exécuter en parallèle ; dans l'exemple 2.3., le BLOC BO et quatre copies du BLOC B1.

Dès lors, cela implique l'existence d'une autre primitive permettant d'attendre la fin de l'exécution d'un BLOC demandée par une primitive execbloc. En effet, dans l'exemple 1.3., on ne peut exécuter "CH : = (A + B)/2" sans que les valeurs de A et B aient été calculées puis transmises au BLOC BO. Or il n'existe pas de communication directe entre opérateurs, les valeurs calculées pour A et B ne peuvent être transmises au BLOC BO pendant son exécution (d'ailleurs on ne connait pas l'opérateur qui l'exécute). Il faut donc suspendre l'exécution du BLOC BO pour que les valeurs de A et B puissent lui être communiquées et le ranger dans l'ensemble des *BLOCS en attente* puisque l'opérateur MAJ ne travaille que sur des BLOCS appartenant à cet ensemble.

La primitive permettant de suspendre un BLOC est la primitive attendre. Elle permet d'attendre que les BLOCS demandés lors de l'exécution des primitives execbloc aient élaborés leurs *objets de sortie*, c'est-à-dire que leur exécution soit terminée.

Cette primitive sert en réalité à dissocier l'appel d'un nouveau BLOC de l'attente de la fin de son exécution qui était faite automatiquement et immédiatement après l'appel dans la première approche de la primitive execbloc. Et de plus, une seule primitive attendre peut être associée à plusieurs primitives execbloc (exemple 1.4.), c'est ce qui permet d'augmenter le nombre de BLOCS exécutés en parallèle. Mais cela correspond aussi à une certaine exploitation du parallélisme existant dans le BLOC et suppose une connaissance des contraintes de précédence des données. Il existe un certain nombre d'algorithmes permettant de les découvrir, en particulier [Ber 66], [RG 70].

Le but que nous nous étions fixés étant d'utiliser au mieux les possibilités de parallélisme du modèle, les problèmes posés par l'écriture des instructions à l'intérieur d'un BLOC n'ont pas été approfondis. Les programmes sont supposés correctement écrits (bien que les primitives execbloc et attendre permettent de détecter certaines erreurs dans les conditions de précédence). Les problèmes que nous mettrons cependant en évidence pourraient en grande partie être résolus par l'utilisation d'une programmation structurée rigoureuse.

Sous ces conditions, le programme de l'exemple 1 peut s'écrire :

Exemple 1.4.

```
BLOC BO ; entrée X ; sortie CH, SH ;
local A, B ;
          si X = 1 alors
                     CH : = 1 ;
                     SH : = 0
                   sinon
                     execbloc (B1 ; X ; A) ;
                     execbloc (B1 ; - X ; B) ;
                     attendre (A, B) ;
                     CH : = (A + B)/2 ;
                     SH : = (A - B)/2
                   fsi

FIN BLOC
```

Le BLOC B1 ne sera extrait de la BIBLIOTHEQUE que si la valeur fournie pour X est différente de 1. Dans ce cas, la primitive attendre permettra de suspendre l'exécution du BLOC BO en attendant que A et B soient évalués. Les deux copies du BLOC B1 pourront s'exécuter en parallèle.

De même, le programme de l'exemple 2 s'écrit :

Exemple 2.4.

```
BLOC BO ; entrée ... ; sortie .... ;
local..... ;
    pour i jusqu'à 4 faire
        execbloc (B1 ; i ; t[i]) ;
    fin faire ;
    attendre (t[1], t[2], t[3], t[4]) ;
    autres-opérations-utilisant-t ;

FIN BLOC
```

Les quatre copies du BLOC B1 peuvent s'exécuter en parallèle ; et l'exécution du BLOC BO est reprise dès que les 4 paramètres de sortie ont été calculés et transmis au BLOC BO.

En définitive, les deux primitives execbloc et attendre représentent les outils fondamentaux de la gestion dynamique du traitement dans MAUD:

1) elles permettent d'exploiter efficacement les possibilités de traitement parallèle en donnant la possibilité de lancer l'exécution de plusieurs BLOCS dynamiquement en fonction des données du programme ;

2) elles permettent l'exécution de programmes non exécutables dans le modèle statique, tels celui de l'exemple 3.

Pour utiliser ces primitives dans nos exemples, et bien qu'elles soient des primitives de base des opérateurs de traitement, une représentation en langage évolué leur a été donnée. Ces primitives peuvent être utilisées par le programmeur qui écrit directement des programmes pour MAUD sous forme d'un ensemble de BLOCS, avec les conditions que cela suppose et dont nous avons parlé précédemment. Ce type de programmation présente quelques similitudes avec l'utilisation des opérations FORK et JOIN de Conway [Con 66]. Il est évident que l'utilisation de ces primitives par un compilateur qui traduit des programmes rédigés en langage évolué en programmes pour MAUD sera plus délicate. Mais l'observation des règles strictes de la programmation structurée dans le langage évolué contribuera à diminuer les problèmes et à faciliter la traduction.

Programmes pour MAUD

Compte-tenu des possibilités offertes par l'utilisation des primitives **execbloc** et **attendre**, un programme pour MAUD se compose de :

1) un ensemble de BLOCS de BIBLIOTHEQUE, qui peuvent être appelés lors de l'exécution du programme. Ces BLOCS sont des BLOCS écrits par le programmeur pour la résolution d'un problème particulier et des BLOCS qui peuvent

être qualifiés de "standard" c'est-à-dire à la disposition de tous les programmeurs, pour des opérations d'échange avec l'extérieur ou des traitements d'erreurs par exemple.

2) un ensemble de *BLOCS exécutables* et de *BLOCS en attente* se trouvant dans l'ensemble X des *BLOCS exécutables* et l'ensemble A des *BLOCS en attente* au début de l'exécution du programme. Cet ensemble est appelé *programme initial*. L'un au moins des BLOCS doit être *exécutable*. Aucun de ces BLOCS ne pourra être l'objet d'une primitive execbloc puisqu'ils ne sont pas en BIBLIOTHEQUE. Par contre, tous les BLOCS ne faisant pas partie du *programme initial* devront être appelés par une primitive execbloc pour être exécuté.

Pour faciliter le lancement d'un programme, il est souhaitable de réduire le *programme initial* à un seul *BLOC exécutable*.

3. PARAMETRES DES NOUVELLES PRIMITIVES

3.1. PRIMITIVE EXECBLOC

Format de la primitive execbloc

les paramètres nécessaires à la primitive sont :

- un nom de BLOC
- une liste de paramètres d'entrée qui seront utilisés lors de l'exécution du BLOC
- une liste de paramètres de sortie qui seront les résultats de l'exécution du BLOC

Il existe plusieurs possibilités quant à la forme des paramètres d'entrée et de sortie :

1. Un paramètre d'entrée peut être une expression qui, calculée, deviendra un *objet d'entrée* du BLOC demandé, (comme dans les exemples précédents). Son utilisation est analogue à celle des paramètes appelés par valeur dans une procédure.

Si cette possibilité était la seule utilisable pour les paramètres d'entrée, elle ne conduirait pas à une bonne exploitation du parallélisme à l'exécution. En effet, soit l'exemple suivant :

Exemple 4.1.

```
BLOC B1 ; entrée ; sortie X ;
local I, J ;
   .
   .
   .
   execbloc (B2 ;...; I) ;
   attendre (I) ;
   execbloc (B3 ; I ; J) ;
   .
   .
   .
   attendre (J) ;
   .
   .
   .
FIN BLOC
```

Il est dommage de devoir suspendre l'exécution de B1 en attendant la valeur attribuée à I par le BLOC B2 pour, dès que l'exécution du BLOC B1 est reprise, fournir cette valeur à un BLOC B3. Il semble beaucoup plus intéressant de pouvoir transmettre directement la valeur de I au BLOC B3 sans passer par l'intermédiaire de B1. Pour cela, il faut autoriser dans l'utilisation de la primitive execbloc l'existence de certains paramètres non calculés.

2. Un paramètre d'entrée peut être un nom. Ce nom est un *nom interne* puisque figurant dans la partie interne d'un BLOC.

Afin d'éviter toute ambigüité quant à la signification d'un *nom interne* placé en paramètre d'entrée (doit-il être considéré comme un nom ou comme une expression ?), nous marquerons ce nom en le faisant précéder d'un caractère '*', et il sera alors appelé "opérande marqué". Notre but n'est pas ici de définir une syntaxe d'écriture des BLOCS, mais de rendre plus lisibles les exemples qui sont donnés.

L'exemple 4.1. peut alors se réécrire :

Exemple 4.2.

```
BLOC B1 ; entrée ; sortie X ;
local I, J ;
  .
  .
  .
    execbloc (B2 ; ... ; I) ;
    execbloc (B3 ; * I ; J) ;
  .
  .
  .
FIN BLOC
```

Dans cet exemple, il faut remarquer que l'exécution du BLOC B1 n'est plus suspendue et qu'elle se poursuit concurremment à l'exécution des BLOCS B2, puis B3. Ce gain de temps sera surtout très important au niveau d'une implémentation matérielle, car en réalité, l'exécution du BLOC B1 suspendue et en attente des résultats de l'exécution du BLOC B2 ne pourra reprendre, non pas dès que l'exécution du BLOC B2 sera terminée, mais seulement dès qu'un opérateur sera libre.

Quelques contrôles devront toutefois être faits au moment de l'exécution de la primitive execbloc. En effet, pour obéir à la règle d'utilisation unique (§ II.4.1.2.) l'objet calculé par le BLOC B2 ne peut être utilisé à la fois par le BLOC B1 et le BLOC B3, c'est-à-dire que dans l'exemple 4.2. ci-dessus, un "attendre (I)" placé derrière le deuxième execbloc serait incorrect. Il faudra donc, lorsqu'un opérande marqué est utilisé en paramètre d'entrée :

- Vérifier que ce nom est déjà apparu en paramètre de sortie d'une précédente primitive execbloc.

- Contrôler que c'est sa première utilisation.

Remarque

La possibilité d'utiliser un nom comme paramètre d'entrée ne signifie pas que le *nom interne* du BLOC sera transmis à l'extérieur du BLOC. Comme dans le modèle statique, à chaque *nom interne* qui doit recevoir une valeur d'un autre BLOC ou émettre une valeur vers un autre BLOC est associé un *nom de communication*. C'est ce *nom de communication* qui sera transmis.

3. Un paramètre de sortie peut être un nom. C'est le cas qui a été utilisé dans tous les exemples jusqu'ici. L'utilisation de ce paramètre est analogue à celle des paramètres résultats d'une procédure. C'est un *nom interne* et comme pour les paramètres d'entrée qui sont des noms, c'est le *nom de communication* qui lui est associé qui est transmis à l'extérieur du BLOC.

4. Un paramètre de sortie peut être un des paramètres de sortie du BLOC dans lequel se trouve la primitive execbloc.

La signification de cette possibilité apparaitra plus nettement lorsque le traitement effectué au moment de l'exécution de la primitive execbloc sera détaillé. Ce cas est symétrique de celui décrit pour les paramètres d'entrée qui sont des noms. Il correspond à faire une transmission de noms de paramètres de sortie à travers les BLOCS.

Exemple 5

```
BLOC B1 ; entrée ; sortie X, Y ;
   .
   .
   .
   execbloc (B2 ; ... ; * Y) ;
   .
   .
   .
FIN BLOC
```

A la fin de l'exécution du BLOC B1, un seul *objet de sortie* est produit (qui prendra la valeur de X) puisque Y a déjà été utilisé (c'est le BLOC B2 qui calculera la valeur de Y).

Toutefois, il se pose un problème d'écriture initiale des paramètres du BLOC puisqu'un seul des deux paramètres indiqués est finalement fourni par B1.

3.2. PRIMITIVE ATTENDRE

Format de la primitive attendre

Le paramètre nécessaire à la primitive attendre est la liste de noms des objets attendus qui permettront de reprendre l'exécution du BLOC. Cette liste ne comprend pas nécessairement tous les noms qui sont apparus en paramètres de sortie dans les primitives execbloc précédant l'opération attendre. C'est-à-dire qu'il est possible d'écrire :

Exemple 6

```
BLOC B0 ; entrée ... ; sortie ... ;
    .
    .
    .
    execbloc (B1 ; ... ; x, y) ;
    .
    .
    .
    attendre (y) ;
    .
    .
    .
    execbloc (B2 ; ... ; z) ;
    .
    .
    attendre (x, z) ;
    .
    .
    .
FIN BLOC
```

Des contrôles sont nécessaires au moment de l'exécution de la primitive attendre pour vérifier que le nom d'un objet déjà reçu ne figure pas dans la liste des paramètres de cette primitive.

D'autre part, dans certains cas, il est impossible de connaître le nom et le nombre des objets attendus dès l'écriture du programme. Par exemple, lorsqu'un même traitement doit être effectué pour le calcul de tous les éléments d'un ensemble. La seule solution envisageable jusqu'ici est d'écrire:

Exemple 7.1.

```
BLOC BO ; entrée  ..... ; sortie ... ;
    .
    .
    .
    pour i jusqu'à n faire
        execbloc (B1 ; i ; t[i]) ;
        attendre (t[i]) ;
    fin faire ;
    .
    .
    .
FIN BLOC
```

Mais cela signifie qu'une séquentialité d'exécution est imposée là où il existe un parallélisme évident. Pour le but qui est fixé, l'exécution de BLOCS en parallèle, il est essentiel de pouvoir exécuter toutes les primitives execbloc avant de suspendre l'exécution du BLOC BO, afin de ne pas perdre le parallélisme existant. L'exemple 7.1. doit pouvoir se réécrire sous la forme suivante :

Exemple 7.2.

```
BLOC BO ; entrée ... ; sortie ... ;
   .
   .
   .
   pour i jusqu'à n faire
     execbloc (B1 ; i ; t[i]) ;
     fin faire ;
   attendre (tous-les-t[i]) ;
   .
   .
   .
FIN BLOC
```

Il faudra donc introduire au niveau du langage des BLOCS d'autres possibilités pour les paramètres de la primitive attendre. Par exemple :

- Un paramètre spécial, TOUS, signifiant que les noms à utiliser sont tous ceux qui sont apparus en paramètres de sortie des précédentes primitives execbloc, sauf ceux qui ont déjà été utilisés, dans une précédente primitive attendre ou comme paramètre d'entrée d'une primitive execbloc, des contrôles étant faits à ce niveau lors de l'exécution des deux primitives.

- La possibilité de désigner un ensemble de noms :

. soit au moyen d'une expression évaluable au moment de l'exécution, comme dans certains langages évolués, par exemple attendre (t[1 ; n]) pour l'exemple 6.

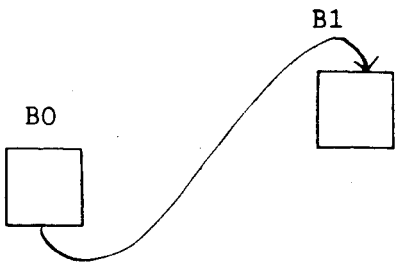
. soit en réalisant des primitives de fabrication d'un ensemble de noms.

4. GRAPHE DES PROGRAMMES EXECUTABLES

4.1. NOTATIONS

Par graphes de programmes, nous entendons graphes d'exécution des programmes. Une représentation schématique est utilisée avec les significations suivantes :

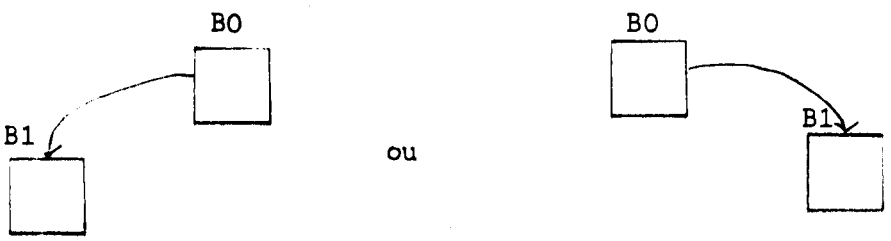
1)



Le *domaine de sortie* du BLOC B0 (ou une partie de ce *domaine de sortie*) est transmis au *domaine d'entrée* du BLOC B1. C'est le seul lien possible entre deux BLOCS dans le modèle statique.

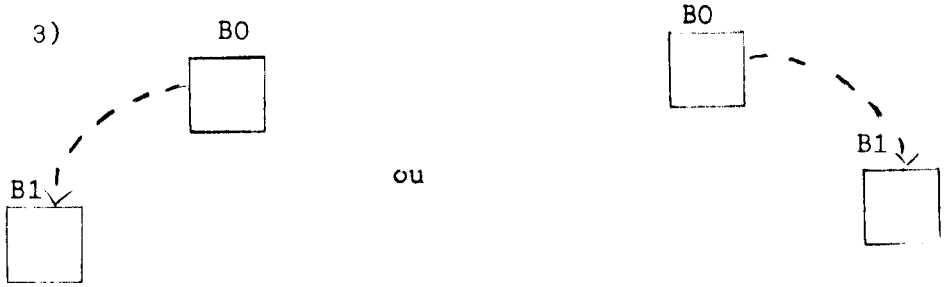
Les liens suivants peuvent exister grâce à l'utilisation des primitives execbloc et attendre

2)



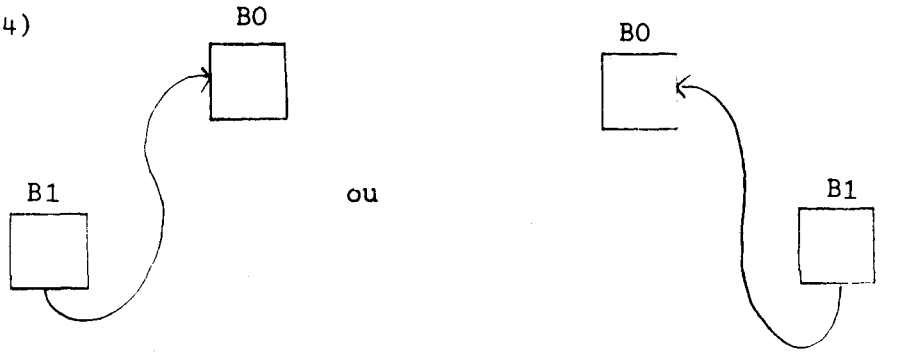
Le BLOC B0 demande l'exécution d'un BLOC B1 par l'intermédiaire d'une primitive execbloc. Tous les paramètres d'entrée possèdent une valeur.

3)



Le BLOC BO demande l'exécution d'un BLOC B1 en utilisant une primitive execbloc. Au moins un des paramètres d'entrée est un opérande marqué (c'est-à-dire dont la valeur sera fournie par un BLOC autre que le BLOC BO).

4)



- Le *domaine de sortie* du BLOC B1 (ou une partie de ce *domaine de sortie*) est transmis au BLOC BO qui a du exécuter une primitive attendre pour l'obtenir. L'exécution du BLOC BO est reprise lorsque l'exécution du BLOC B1 est complètement terminée.

4.2. GRAPHES

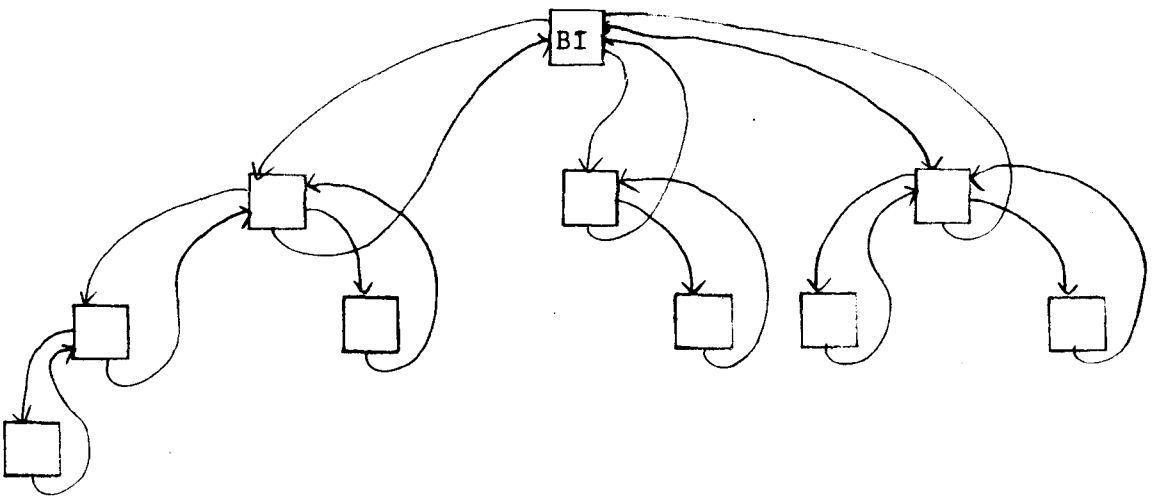
Si les paramètres d'entrée des primitives execbloc ne pouvaient être que des expressions alors les graphes des programmes exécutés par MAUD seraient essentiellement arborescents.

Exemple 8

```
BLOC BO ; entrée ... ; sortie ... ;
    .
    .
    .
    execbloc (B1 ; 4 ; X)
    .
    .
    attendre (X)
    .
    .
    .
FIN BLOC
```

En effet, dans ce cas, la valeur de X calculée par le BLOC B1 ne peut être utilisée que par le BLOC BO, puisque BO est le seul à connaître le *nom de communication* associé au *nom interne* X.

Ainsi, pour chaque BLOC BI constituant le *programme initial*, il peut exister une arborescence telle que la suivante:

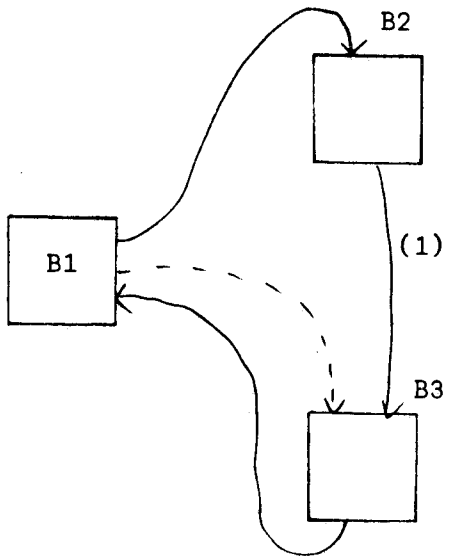


Si il est imposé qu'un *programme initial* soit constitué d'un seul BLOC, le graphe des programmes exécutables est un arbre. Dans ce cas, le BLOC situé à la racine de l'arbre est appelé *BLOC initial* : il doit être *exécutable* et comporter un certain nombre d'opérations execbloc permettant de lancer le traitement. Le *BLOC initial* peut aussi à lui tout seul constituer un programme.

Les programmes ayant un graphe d'exécution en arbre ne permettent pas une très bonne exploitation du parallélisme. L'introduction des opérandes marqués dans les paramètres d'entrée des primitives execbloc autorisent un autre type de graphes.

Exemple 1. : avec opérandes marqués en paramètres d'entrée

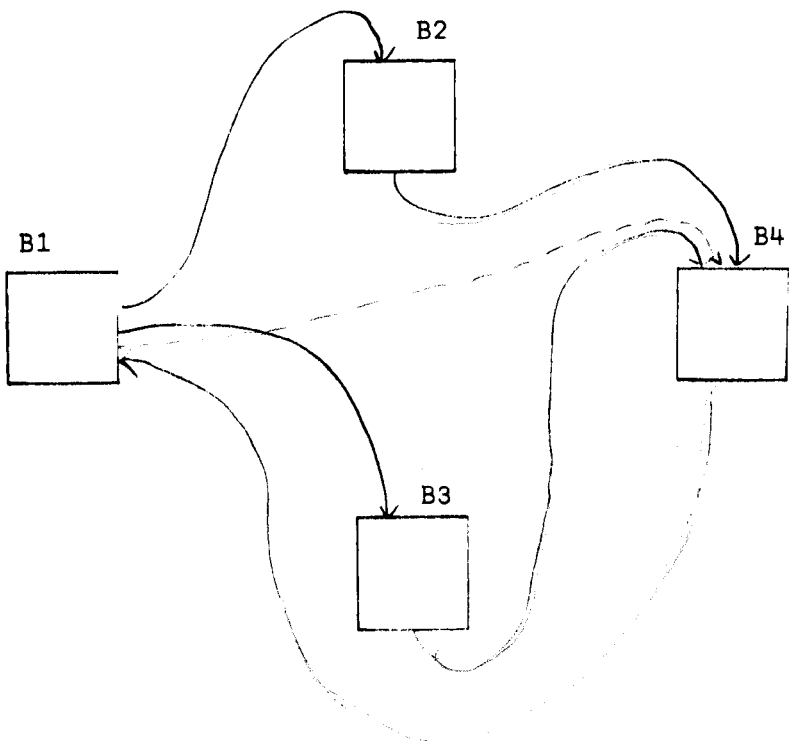
```
BLOC B1 ;...
  .
  .
  .
  execbloc (B2 ;... ; I) ;
  .
  .
  .
  execbloc (B3 ; * I ; J) ;
  .
  .
  .
  attendre (J) ;
FIN BLOC
```



Il faut remarquer que l'arc noté (1) dans le graphe ci-dessus ne pourrait exister qu'entre deux BLOCS du *programme initial* si seules des expressions étaient autorisées en paramètres d'entrée des primitives execbloc.

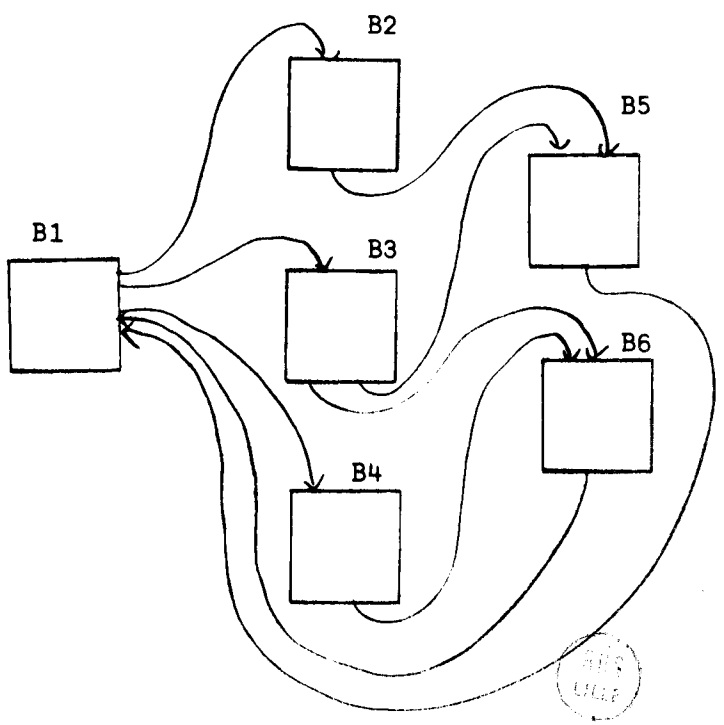
Exemple 2. : avec opérandes marqués en paramètres d'entrée

```
BLOC B1 ; ...
   .
   .
   .
   execbloc (B2 ;...; I) ;
   .
   .
   .
   execbloc (B3 ;...; J) ;
   .
   .
   .
   execbloc (B4 ; * I , * J ; K)
   .
   .
   .
   attendre (K) ;
   .
   .
   .
FIN BLOC
```



Exemple 3. : avec opérandes marqués en paramètres d'entrée

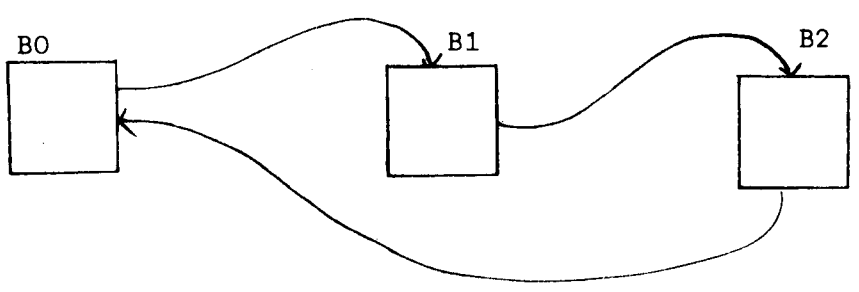
```
BLOC B1 ;...
    .
    .
    .
    execbloc (B2 ;...; I) ;
    .
    .
    .
    execbloc (B3 ;...; J, L) ;
    .
    .
    .
    execbloc (B4 ;...; K) ;
    .
    .
    .
    execbloc (B5 ; * I, * J ; M) ;
    .
    .
    .
    execbloc (B6 ; * K, * L ; N) ;
    .
    .
    .
    attendre (M, N) ;
    .
    .
    .
FIN BLOC
```



De même, le marquage des paramètres de sortie dans la primitive execbloc fournit un autre type de liaison entre BLOCS et donc de nouveaux types de graphes :

Exemple 1. : avec opérandes marqués en paramètres de sortie

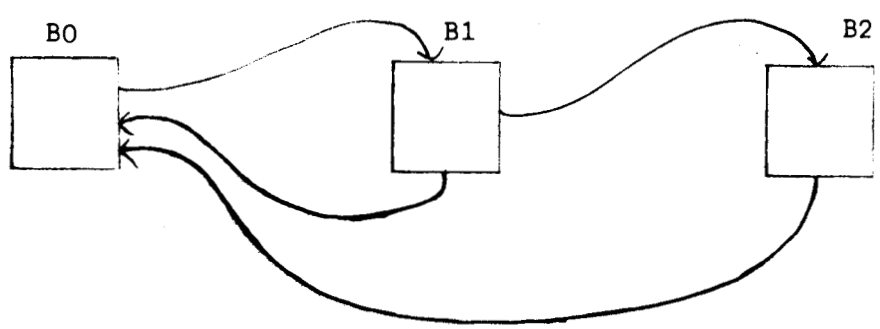
```
BLOC BO ; ...
   .
   .
   .
   execbloc (B1 ; ... ; X, Y) ;
   .
   .
   .
FIN BLOC

BLOC B1 ; entrée ... ; sortie A, B ;
   .
   .
   .
   execbloc (B2 ; ... ; * A, * B) ;
   .
   .
   .
 FIN BLOC
```



Exemple 2. : avec opérandes marqués en paramètres de sortie

```
BLOC BO ; ...
    .
    .
    .
    execbloc (B1 ; ... ; X, Y) ;
    .
    .
    .
FIN BLOC
```

```
BLOC B1 ; entrée ... ; sortie A, B ;
    .
    .
    .
    execbloc (B2 ; ... ; * A) ;
    .
    .
    .
FIN BLOC
```



5. IMPLICATIONS DU CARACTERE DYNAMIQUE AU NIVEAU DU MODELE

5.1. LE MODÈLE DYNAMIQUE

Les primitives execbloc et attendre, qui permettent une gestion dynamique des BLOCS, sont deux nouvelles fonctions qui sont exécutées par les opérateurs de traitement. Leur utilisation entraîne un certain nombre de modifications dans le modèle du chapitre précédent.

Pour MAUD, l'utilisation de la primitive execbloe apparait comme la possibilité d'ajouter un nouveau BLOC à l'ensemble des *BLOCS en attente* : elle est considérée comme une *demande d'exécution* d'un BLOC. Celle-ci est déposée par l'opérateur de traitement dans un ensemble de *demandes d'exécution* de BLOCS. A l'aide de cette *demande d'exécution* et de la BIBLIOTHEQUE de BLOCS, il faut former un nouveau *BLOC* : ce travail est confié à un opérateur de construction de BLOCS, l'opérateur CONSTRUCTEUR.

L'opérateur CONSTRUCTEUR ajoute le BLOC qu'il construit à l'ensemble des *BLOCS en attente*. Le BLOC construit est quelquefois un *BLOC exécutable*, c'est-à-dire que son *domaine d'entrée* est *complet* (cas où les paramètres d'entrée de la primitive execbloc qui a créé la demande d'exécution sont tous des expressions ; sinon il en existe au moins un qui est un opérande marqué). L'opérateur CONSTRUCTEUR pourrait prendre en compte l'état du BLOC, *exécutable* ou *en attente*, et placer le BLOC directement dans l'ensemble adéquat. Cependant, il ne le fait pas, ceci dans le but de garder au système une organisation en pipeline. Et de toute façon, un *BLOC exécutable* se trouvant dans l'ensemble des *BLOCS en attente* en sera rapidement extrait par l'opérateur MAJ puisqu'il ne nécessite aucune modification.

Quant à l'utilisation de la primitive attendre, puisqu'aucune communication directe entre opérateurs n'est possible, elle ne correspond pas à une mise en sommeil de l'opérateur jusqu'à l'arrivée des objets attendus, mais à la libération du processeur de traitement et à la création d'un nouveau *BLOC en attente*. En effet, le BLOC dans lequel la primitive attendre a été exécutée correspond à la notion de *BLOC en attente* du modèle statique, car son exécution ne pourra se poursuivre que lorsque les *objets de sortie* des BLOCS dont il a demandé l'exécution par les primitives execbloc, lui auront été transmis.

Par conséquent, la primitive attendre provoque le rangement de ce BLOC dans l'ensemble A des *BLOCS en attente* (après quelques modifications sur le *domaine d'entrée* qui seront détaillées par la suite). Puis l'opérateur de traitement ainsi libéré recherche un autre *BLOC exécutable* dans l'ensemble X des *BLOCS exécutables*.

Le nouveau modèle ainsi défini est présenté figure 1.

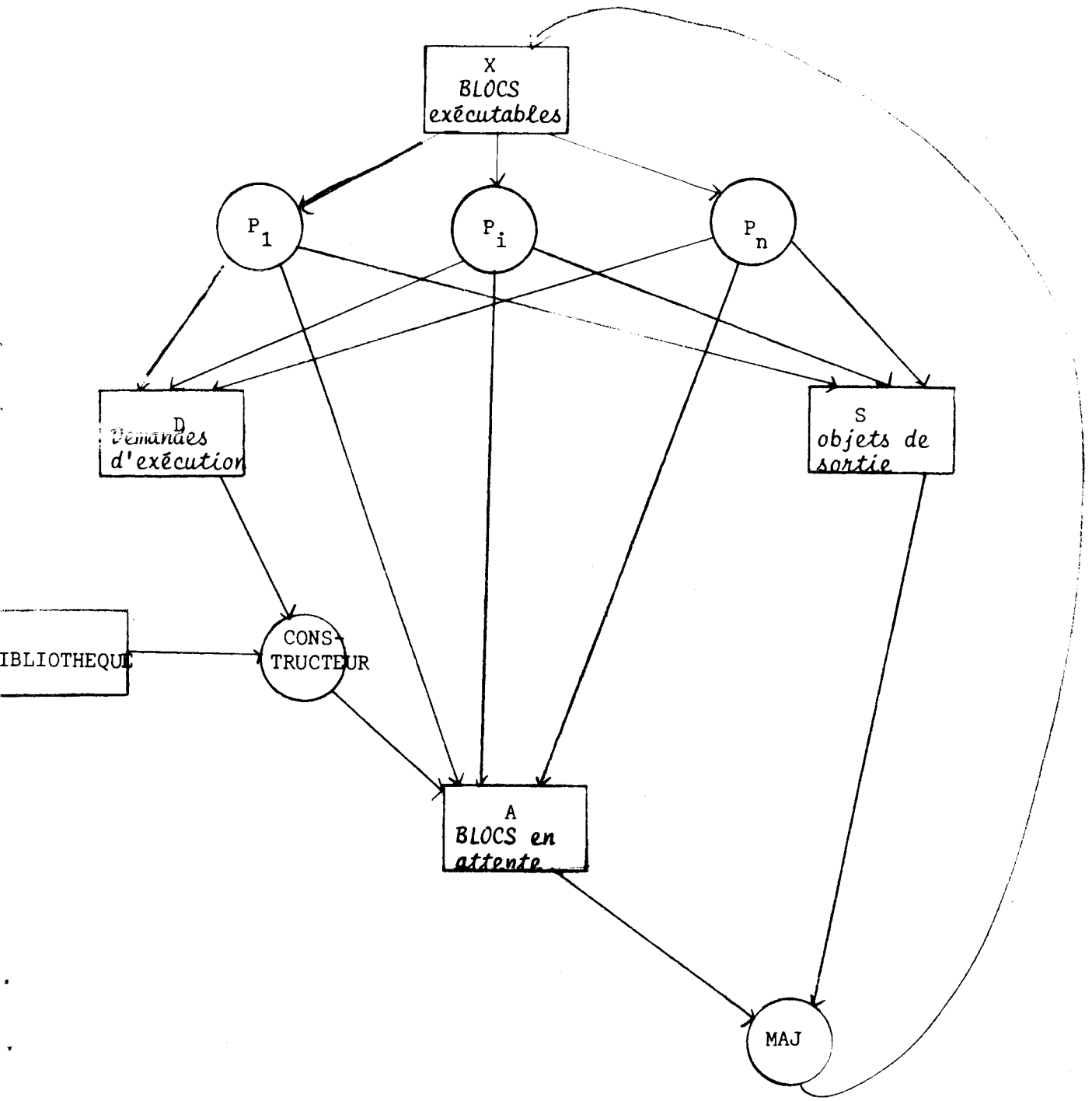


Figure 1. : Le modèle dynamique.

5.2. NOUVEAUX ENSEMBLES D'OBJETS

Deux nouveaux ensembles d'objets sont nécessaires dans le modèle dynamique.

5.2.1. ENSEMBLE D DE DEMANDES D'EXECUTION

Les *demandes d'exécution* servent à construire de nouveaux BLOCS pendant l'exécution d'un programme. Chaque *demande d'exécution* comporte :

* un nom de BLOC, permettant de désigner un modèle de BLOC en BIBLIOTHEQUE ;

* des informations permettant de fabriquer avec le modèle de BLOC trouvé en BIBLIOTHEQUE, un vrai BLOC pour MAUD avec *domaines* et *liste de correspondance* ; plus précisément :

. un *domaine d'entrée*, c'est-à-dire une liste de couples (NC_i, V_i) et/ou (NC_i, *non-évalué*)

. un *domaine de sortie*, c'est-à-dire une liste de couples (NC_i, *non-évalué*).

5.2.2. BIBLIOTHEQUE

Les BLOCS de BIBLIOTHEQUE sont des modèles de BLOCS qui pourront être introduits dans le système pendant l'exécution d'un programme si leur exécution est réclamée. Ces modèles doivent comporter, outre la partie interne du BLOC, des informations permettant de construire un vrai BLOC pour MAUD en utilisant les *demandes d'exécution*.

Les BLOCS de BIBLIOTHEQUE comprennent :

* un nom permettant de les identifier

* la partie interne d'un BLOC : instructions + objets locaux

* deux listes de *noms internes* qui correspondent aux *objets d'entrée* et aux *objets de sortie* du BLOC. Ces listes **serviront** au moment de **la** construction du BLOC pour former la *liste de correspondance.*

La forme des BLOCS de BIBLIOTHEQUE est analogue à celle d'un sous programme. Ils sont normalement écrits par le programmeur. Mais il peut exister des BLOCS standard de traitements d'erreurs ou permettant les échanges avec l'extérieur. Les BLOCS de BIBLIOTHEQUE ne comportent ni *domaine d'entrée* ni *domaine de sortie* qui sont des notions propres à MAUD et donc complètement transparentes à l'utilisateur. De plus, ces *domaines* font intervenir des *noms de communication* qui ne sont connus qu'au moment de **l'**exécution du programme. Il est donc impossible de les construire à priori.

6. IMPLICATIONS DU CARACTERE DYNAMIQUE AU NIVEAU DES OPERATEURS DE TRAITEMENT

Les primitives execbloc et attendre doivent faire partie des primitives de base des opérateurs de traitement au même niveau que les primitives début de bloc et fin de bloc. Elles agissent également sur les parties interface et communication des BLOCS.

Afin de permettre des détections d'erreurs dans l'utilisation des primitives execbloc et attendre, le traitement effectué par la primitive début de bloc est légèrement différent de celui effectué dans le modèle statique (§ II.4.1.1.). En effet, l'exécution des primitives execbloc et attendre entraîne des modifications de la *liste de correspondance d'entrée* que la primitive début de bloc n'est plus seule à utiliser. Et de plus, la relation R1 (§ II.4.1.1.) ne sera plus toujours vérifiée. En effet, lors de l'exécution de la primitive début de bloc, il sera possible d'avoir :

card (*liste de correspondance d'entrée*) ≠ card (*domaine d'entrée*).

6.1. PRIMITIVE DÉBUT DE BLOC

Si DE est le *domaine d'entrée* du BLOC
LCE la *liste de correspondance d'entrée*
DP le *domaine propre*

alors

$$DP = DP + \{(NI_k, V_j) \mid (NC_j, NI_k) \in LCE \text{ et } (NC_j, V_j) \in DE\}$$

$$- \{(NI_k, V_k) \mid (NC_j, NI_k) \in LCE \text{ et } (NC_j, V_j) \in DE\}$$

$$LCE = LCE - \{(NC_j, NI_k) \mid (NC_j, V_j) \in DE\}$$

c'est-à-dire que pour chaque transmission de valeur d'un *nom de communication* NC vers un *nom interne* NI, le couple correspondant (NC, NI) est supprimé de la *liste de correspondance d'entrée*. Ainsi il ne peut plus être réutilisé.

6.2. PRIMITIVE EXECBLOC

Les paramètres de la primitive servent à construire une *demande d'exécution*. Suivant la forme des paramètres d'entrée ou de sortie, le traitement effectué est différent; mais dans tous les cas, à partir des paramètres d'entrée, il faut fabriquer un *domaine d'entrée* pour le BLOC demandé, et à partir des paramètres de sortie, il faut fabriquer un *domaine de sortie* pour ce même BLOC. Quant au nom du BLOC, il est utilisé sans modification.

A - Cas général

1) les paramètres d'entrée sont des expressions $E_1,\ldots, E_n$.

Soient V_i, i = 1,..., n les valeurs calculées à l'aide des expressions $E_1,\ldots,E_n$; alors le *domaine d'entrée* est DE = $\{(NC_i, V_i)\}$ où les NC_i sont des *noms de communication* tous distincts délivrés par un Fournisseur de Noms propre au processeur de traitement ou commun à tous les processeurs (voir § 9).

Il peut sembler curieux et peu utile d'associer un nom de communication à chaque valeur V_i. La logique voudrait que l'on donne simplement la liste des valeurs et une simple transmission par position pourrait être effectuée par la primitive début de bloc au début de l'exécution d'un BLOC. Mais dans la solution qui a été choisie, cela obligerait à associer à chaque *BLOC exécutable* un indicateur spécial permettant de savoir si le BLOC est déjà passé par l'état *en attente* ou non afin de faire une transmission de valeurs différente pour chaque cas. Une autre possibilité peut être envisagée afin de ne pas avoir à faire cette distinction. Nous en parlerons à propos de la primitive attendre.

2) les paramètres de sortie sont des *noms internes* $N_1,\ldots,N_p$.

Pour chaque *nom interne* N_i, on modifie la *liste de correspondance d'entrée* LCE en lui ajoutant un couple (NC_i, N_i), où NC_i est un *nom de communication* délivré par le Fournisseur de Noms, et après avoir vérifié qu'il n'existe pas déjà dans la *liste de correspondance d'entrée* un couple (NC', N_i). Cela signifierait que le nom N_i est déjà apparu en paramètre de sortie d'une primitive execbloc précédemment exécutée sans qu'une primitive attendre ait été exécutée, comme dans l'exemple suivant :

Exemple 9

```
BLOC BO ; entrée ... ; sortie ... ;
   .
   .
   .
   execbloc (B1 ; ... ; i) ;
   .
   .
   .
   execbloc (B2 ; ... ; i) ;
   .
   .
   .
   attendre (i) ;
   .
   .
   .
FIN BLOC
```

Ce cas pose des problèmes de déterminisme du programme, et est donc détecté au moment de l'exécution de la primitive execbloc.

Au niveau des paramètres de sortie, la primitive execbloc modifie donc la *liste de correspondance d'entrée* LCE :

$$LCE = LCE + \{(NC_i, N_i) \mid i = 1,\ldots,p\}$$

et élabore en même temps le troisième élément de la *demande d'exécution* qui constituera le *domaine de sortie* DS du BLOC demandé :

$$DS = \{(NC_i, \textit{non-évalué})\}$$

B - Cas des opérandes marqués

1) en paramètre d'entrée

execbloc (B2 ; * I ; J)

L'opérande marqué a déjà du figurer en paramètre de sortie, et non marqué, dans une primitive execbloc exécutée précédemment, par exemple

execbloc (B1 ; ... ; I)

Cela signifie que l'on désire transmettre le résultat calculé par le BLOC B1, directement au BLOC B2 et sans passer par l'intermédiaire du BLOC dans lequel se trouvent ces deux execbloc.

Le *nom interne* I a été associé à un *nom de communication* NC lors de l'exécution du premier execbloc, et on a ajouté le couple (NC, I) à la *liste de correspondance d'entrée*. Le deuxième execbloc désigne un destinataire pour l'objet qui sera calculé par le BLOC B1 : c'est le BLOC B2. Pour respecter la règle d'utilisation unique (§ II.4.1.2.), le BLOC B2 doit être le seul destinataire de cet objet. Il faut donc supprimer le couple (NC, I) de la *liste de correspondance d'entrée*, afin d'éviter l'utilisation possible du nom I dans une primitive attendre avant que ce même nom ait figuré en paramètre de sortie d'une autre primitive execbloc.

En résumé, pour chaque opérande marqué I figurant en paramètre d'entrée, le traitement à effectuer est le suivant :

```
LCE = LCE - (NC, I)
DE = DE + (NC, non-évalué)
```

où DE est le deuxième élément de la *demande d'exécution* et constituera le *domaine d'entrée* du BLOC demandé.

2) en paramètre de sortie

execbloc (B1 ; I ; * J)

L'opérande marqué désigne un des *objets de sortie* que doit élaborer le BLOC dans lequel se trouve cette primitive execbloc, c'est-à-dire que l'écriture du BLOC doit être du type suivant, avec les problèmes déjà évoqués au § 3.1.

Exemple 10

```
BLOC BO ; entrée ... ; sortie J, K ;
   .
   .
   .
   execbloc (B1 ; ... ; * J) ;
   .
   .
   .
FIN BLOC
```

Cette possibilité correspond à faire une transmission de *noms de communication* à travers le BLOC B1. Ce *nom de communication* figure dans la *liste de correspondance de sortie* du BLOC BO où il est associé au *nom interne* J. Afin que la règle d'Assignation Unique soit respectée, c'est-à-dire que le BLOC B1 soit le seul à produire une valeur pour le *nom de communication* associé à J, il faut supprimer le couple (NC, J) de la *liste de correspondance de sortie* du BLOC BO.

En résumé, pour chaque opérande marqué J figurant en paramètre de sortie, le traitement à effectuer est le suivant :

LCS = LCS - (NC, J)
DS = DS + (NC, *non-évalué*)

où DS est le troisième élément de la *demande d'exécution* et constituera le *domaine de sortie* du BLOC demandé.

6.3. PRIMITIVE ATTENDRE

attendre (liste de *noms internes*)

La primitive attendre a pour but d'arrêter l'exécution d'un BLOC B en attendant des valeurs calculées dans d'autres BLOCS. Le BLOC qui va être suspendu correspond donc à la notion de *BLOC en attente* : son *domaine d'entrée* doit contenir les *noms de communication* des objets attendus. Ces noms ont été rangés dans la *liste de correspondance d'entrée* LCE lors de l'exécution de primitives execbloc. La primitive attendre constitue un nouveau *domaine d'entrée* pour le BLOC B dans lequel elle se trouve, puis range ce BLOC B dans l'ensemble A des *BLOCS en attente*, c'est-à-dire que le traitement suivant est effectué :

Si N_i i = 1,..., p sont les *noms internes* qui sont les paramètres de la primitive attendre, alors

$DE = \{(NC_i, \textit{non évalué}) \mid (NC_i, N_i) \in LCE \quad i = 1,\ldots, p\}$
$A = A + B$

Il n'est pas possible d'attendre un objet déjà reçu ou non encore demandé, puisque lors de l'exécution de la primitive début-de-bloc les couples (NC, NI) correspondants aux objets reçus sont supprimés de la *liste de correspondance d'entrée*.

Ainsi l'exemple suivant conduira à une erreur.

Exemple 11.

```
BLOC BO ; entrée ... ; sortie ... ;
     .
     .
     .
     execbloc (B1 ;...; x, y) ;
     .
     .
     .
     attendre (x, y) ;
     .
     .
     .
     execbloc (B2 ; ... ; z) ;
     .
     .
     .
     attendre (y, z) ;
     .
     .
     .
FIN BLOC
```

Au moment de l'exécution de la deuxième primitive attendre, il n'existe pas dans la *liste de correspondance d'entrée* de *nom de communication* associé à y.

Remarque

La *liste de correspondance d'entrée* sert à faire les transmission de valeurs de l'extérieur du BLOC, où elles sont associées à des *noms de communications*, vers l'intérieur du BLOC, où elles sont associées à des *noms internes*. La transmission s'effectue en recherchant dans la *liste de correspondance d'entrée*, pour chaque *nom de communication* figurant dans le *domaine d'entrée* le *nom interne* qui lui est associé. Elle ne peut pas se faire par position parce qu'il est possible de n'attendre qu'une partie des objets demandés lors de l'exécution de primitives execbloc.

Moyennant la création d'une autre liste au moment de l'exécution de la primitive attendre, il serait tout de même envisageable de faire cette transmission par position.

Dans cette solution, la *liste de correspondance d'entrée* se réduirait à une "liste de correspondance pour les *noms de communication* sans destinataire". Elle se construit de la même façon que la *liste de correspondance d'entrée* utilisée jusqu'ici, c'est-à-dire avec les paramètres de sortie des primitives execbloc. Un destinataire pour un *nom de communication*, c'est-à-dire un nom de BLOC dans le *domaine d'entrée* duquel figurera ce *nom de communication* est désigné de deux façon.

- Soit par l'utilisation d'un opérande marqué dans une primitive execbloc. Le destinataire désigné est le nom du BLOC qui figure comme paramètre dans l'execbloc. Or dans ce cas, le couple (NC, NI) est supprimé de la *liste de correspondance d'entrée* (§ 6.2.B.).

- Soit par l'exécution d'une primitive attendre. Le destinataire est alors le BLOC dans lequel se trouve cette primitive. On crée alors une nouvelle liste, analogue à une liste de paramètres formels, dans laquelle on place tous les *noms internes* figurant dans la primitive attendre. En même temps, on construit le nouveau *domaine d'entrée* du BLOC, et on supprime de la *liste de correspondance d'entrée* tous les couples correspondants. C'est-à-dire, si $N_1, \ldots, N_p$ sont les *noms internes* figurant dans la primitive attendre, et si L est la nouvelle liste créée :

$$LCE = LCE - \{(NC_i, N_i) \mid N_i \quad i = 1, \ldots, p\}$$

$$L = \{N_i, \text{ dans l'ordre de 1 à p}\}$$

$$DE = \{(NC_i, \textit{non évalué}), \text{ dans l'ordre de 1 à p}\}.$$

Dans LCE ne figurent plus que des couples où les *noms de communication* sont sans destinataire.

Cette solution petmet de faire, à la reprise de l'exécution du BLOC, une transmission des valeurs des *noms de communication* vers les *noms internes* par position. De plus, elle simplifie la mise en forme des BLOCS faisant partie du *programme initial* puisque dans ce cas, la *liste de correspondance d'entrée* est vide. Seule la liste L doit exister.

7. FONCTION DES OPERATEURS DANS LE MODELE DYNAMIQUE

Il devient maintenant possible de définir ou de redéfinir la fonction de chaque opérateur vis-à-vis du système global.

Soient DE un *domaine d'entrée*
DS un *domaine de sortie*
LCE une *liste de correspondance d'entrée*
LCS une *liste de correspondance de sortie*
DP un *domaine propre*.

7.1. OPÉRATEUR DE TRAITEMENT

Le traitement effectué par un opérateur T_i est le même que dans le cas statique, c'est-à-dire le traitement d'un BLOC, sauf :

1) lors de la transmission des valeurs de la partie communication vers la partie interne du BLOC, c'est-à-dire lors de l'exécution de la primitive début de bloc : chaque couple utilisé dans la *liste de correspondance d'entrée* pour faire la transmission est supprimé de cette liste ;

2) lors de l'exécution des primitives donnant le caractère dynamique :

* l'exécution d'une primitive execbloc provoque la modification de la *liste de correspondance d'entrée* et le rangement dans l'ensemble D d'une *demande d'exécution*

* l'exécution d'une primitive attendre provoque la modification du *domaine d'entrée* du BLOC et le rangement du BLOC complet dans l'ensemble A des *BLOCS en attente*, et donc la libération de l'opérateur.

7.2. OPÉRATEUR CONSTRUCTEUR

Son rôle est de construire un BLOC à partir d'une *demande d'exécution* trouvée dans l'ensemble D, et de placer ce BLOC dans l'ensemble A des *BLOCS en attente*. Pour cela, il utilise le nom de BLOC trouvé dans la *demande d'exécution* : il doit exister dans la BIBLIOTHEQUE un modèle de même nom.

Le modèle qui se trouve en BIBLIOTHEQUE ne comprend que :

* un ensemble "instruction + objets" destiné à former la partie interne du BLOC ;

* deux listes de noms E et S associées aux *objets d'entrée* et *objets de sortie* servant à constituer la partie interface du BLOC.

La partie communication du BLOC se trouvant dans la *demande d'exécution*, seule la partie interface est à construire, c'est -à-dire la *liste de correspondance* :

Si DE désigne le *domaine d'entrée* et DS le *domaine de sortie* fournis par la *demande d'exécution*:

Soient (NC_j, V_j) $j = 1,\ldots, n$ les couples figurant dans DE
$(NC_k,$ *non-affecté*$)$ $k = 1,\ldots,p$ les couples figurant dans DS
NI_j $j = 1,\ldots, n$ les noms de E
NI_k $k = 1,\ldots,p$, les noms de S.
le CONSTRUCTEUR construit

$$LCE = \{(NC_\ell, NI_\ell) \mid \ell = 1,\ldots, n \quad NC_\ell \in DE \text{ et } NI_\ell \in E\}$$

$$LCS = \{(NC_\ell, NI_\ell) \mid \ell = 1,\ldots,p \quad NC_\ell \in DS \text{ et } NI_\ell \in S\}$$

Les correspondances entre les *noms de communication* et les *noms internes* se font par position.

Puis l'opérateur CONSTRUCTEUR rassemble les différents éléments afin de constituer un BLOC.

Au niveau de MAUD, le traitement effectué est le suivant, si DX désigne la *demande d'exécution* pour le BLOC B :

```
D = D - DX
A = A + B
```

L'opérateur CONSTRUCTEUR est un opérateur relativement simple et peu chargé. Ce travail pourrait même être supporté par l'opérateur MAJ. Par contre, il serait plus sollicité si la solution proposée au paragraphe II.4. était retenue, c'est-à-dire si chaque BLOC était décomposé en deux :

- d'une part, le *domaine d'entrée* du BLOC
- d'autre part, le BLOC sans son *domaine d'entrée*.

Le processeur ne travaillerait plus que sur les *domaines*. Toutes les autres manipulations de BLOCS seraient à la charge du CONSTRUCTEUR.

7.3. OPÉRATEUR MAJ

La fonction de l'opérateur MAJ reste identique à celle décrite dans le modèle statique, la composante dynamique n'affectant pas le contrôle dirigé par les données.

Cependant, si la solution proposée au paragraphe 6.3. était adoptée, les transmissions de la partie communication vers la partie interne se faisant par position, les *noms de communications* ne seraient plus utiles lors de l'exécution de la primitive début-de-bloc. Dans ce cas, l' opérateur MAJ pourrait simplement remplacer, dans les *BLOCS en attente*, les *noms de communication* se trouvant dans les *domaines d'entrée* par leurs valeurs trouvées dans l'ensemble des *objets de sortie*.

8. ETAPES DE LA VIE D'UN BLOC

Tous les BLOCS présents dès le début du traitement sont des *BLOCS en attente*. Ceux dont le *domaine d'entrée* possède toutes les valeurs attendues deviendront les *BLOCS exécutables* grâce à l'intervention immédiate de l'opérateur MAJ. L'un au moins des *BLOCS en attente* doit remplir cette condition afin que le traitement puisse commencer.

* un *BLOC en attente* devient un *BLOC exécutable* lorsque son *domaine d'entrée* a été complété par l'opérateur MAJ à l'aide des *objets de sortie* obtenus par l'exécution d'autres BLOCS ; ou si son *domaine d'entrée* est vide.

* un *BLOC exécutable* qui ne fait pas de *demandes d'exécution* d'autres BLOCS est traité entièrement par le même opérateur de traitement et produit un *domaine de sortie*. Le BLOC ainsi terminé est détruit.

* l'exécution d'un BLOC B1 qui fait des *demandes d'exécution* par la primitive execbloc, peut être suspendue par une primitive attendre, lorsque cette exécution ne peut plus être poursuivie sans que le BLOC B1 obtienne les *objets de sortie* provenant des *demandes d'exécution* qu'il a faites. Le BLOC B1 est alors placé dans l'ensemble A des *BLOCS en attente*. Son *domaine d'entrée* est modifié de telle façon que le BLOC soit en attente des *objets de sortie* demandés.

Les exemples qui suivent illustrent ces différentes étapes. Pour chaque exemple, il est décrit :

- l'évolution des parties interface et communication du BLOC que l'on exécute sur MAUD ;

- l'évolution des ensembles d'objets due à l'exécution du BLOC B. Sur le diagramme ne sont portées que les modifications de chaque ensemble. Ce diagramme ne représente qu'une exécution possible. Il peut en exister d'autres suivant les instructions contenues dans B, B1 et B2 et l'occupation des différents opérateurs.

Dans ces exemples,

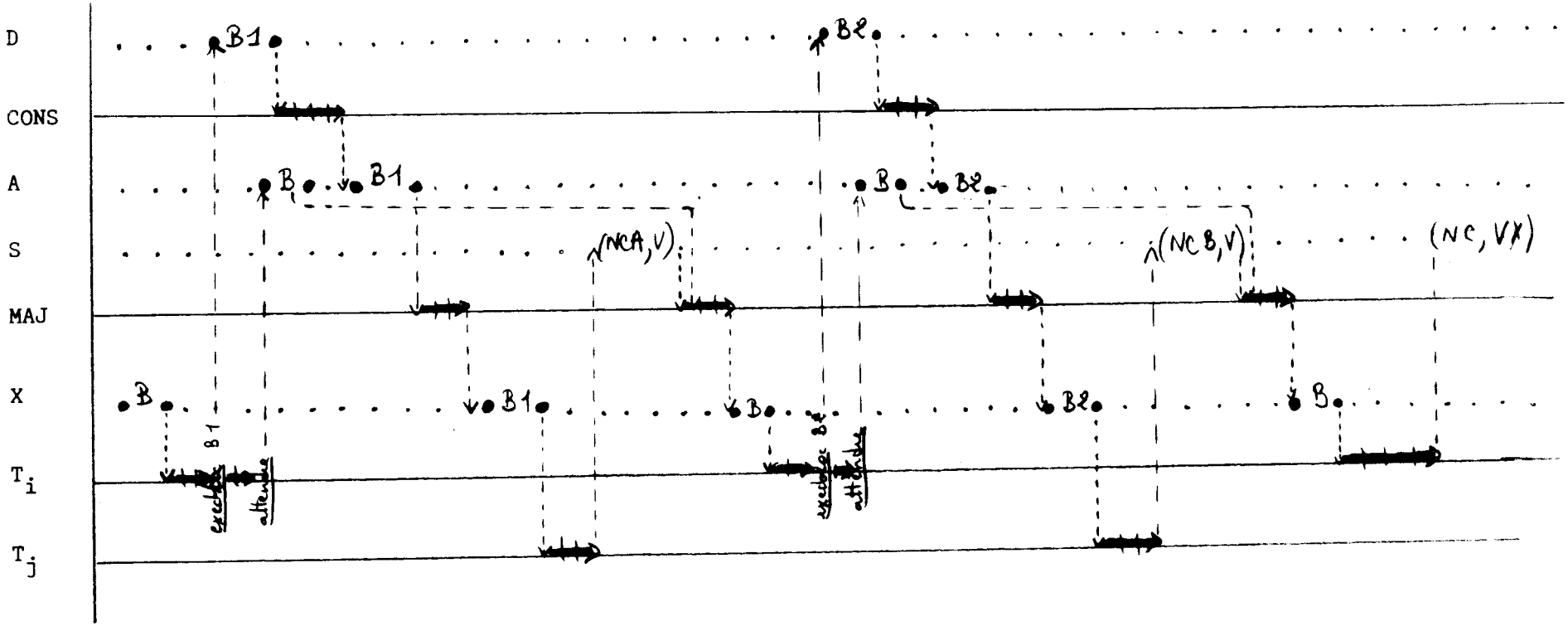
DE représente le *domaine d'entrée*
DS le *domaine de sortie*
LCE la *liste de correspondance d'entrée*
LCS la *liste de correspondance de sortie*
DP le *domaine propre*

Exemple 1

```
BLOC B
DE (NC1, V1),(NC2, V2) ; DS (NC, non-affecté) ;
LCE (NC1, A) , (NC2, B) ; LCS (NC, X) ;
DP (A, -), (B, -), (X, -), (Y, VY),(Z, VZ),(R1, -), (R2, -) ;
  .
  .
  .
  execbloc (B1 ; VY ; R1) ;
  attendre (R1) ;
  execbloc (B2 ; VZ ; R2) ;
  attendre (R2)
  .
  .
  .
FIN BLOC
```

MOMENT	DE	LCE	LCS	DS
prise en charge du BLOC B	(NC1, V1), (NC2, V2)	(NC1, A), (NC2, B)	(NC, X)	(NC, non-affecté)
début d'exécution	/	/	idem	idem
après le 1er execbloc	/	(NCA, R1)	idem	idem
après le 1er attendre	(NCA, non-affecté)	(NCA, R1)	idem	idem
reprise du BLOC B	(NCA, V)	(NCA, R1)	(NC, X)	(NC, non-affecté)
suite d'exécution	/	/	idem	idem
après le 2ème execbloc	/	(NCB, R2)	idem	idem
après le 2ème attendre	(NCB, non-affecté)	idem	idem	idem
reprise du BLOC B	(NCB, V)	(NCB, R2)	(NC, X)	(NC, non-affecté)
suite d'exécution	/	/	idem	idem
fin d'exécution	/	/	idem	(NC, VX)

Exemple 1. : Evolution des parties communication et interface du BLOC B.



Exemple 1. : Une exécution possible.

D : demandes d'exécution

A : BLOCS en attente

S : domaines de sortie

X : BLOCS exécutables

T_i, T_j, T_k : opérateurs quelconques

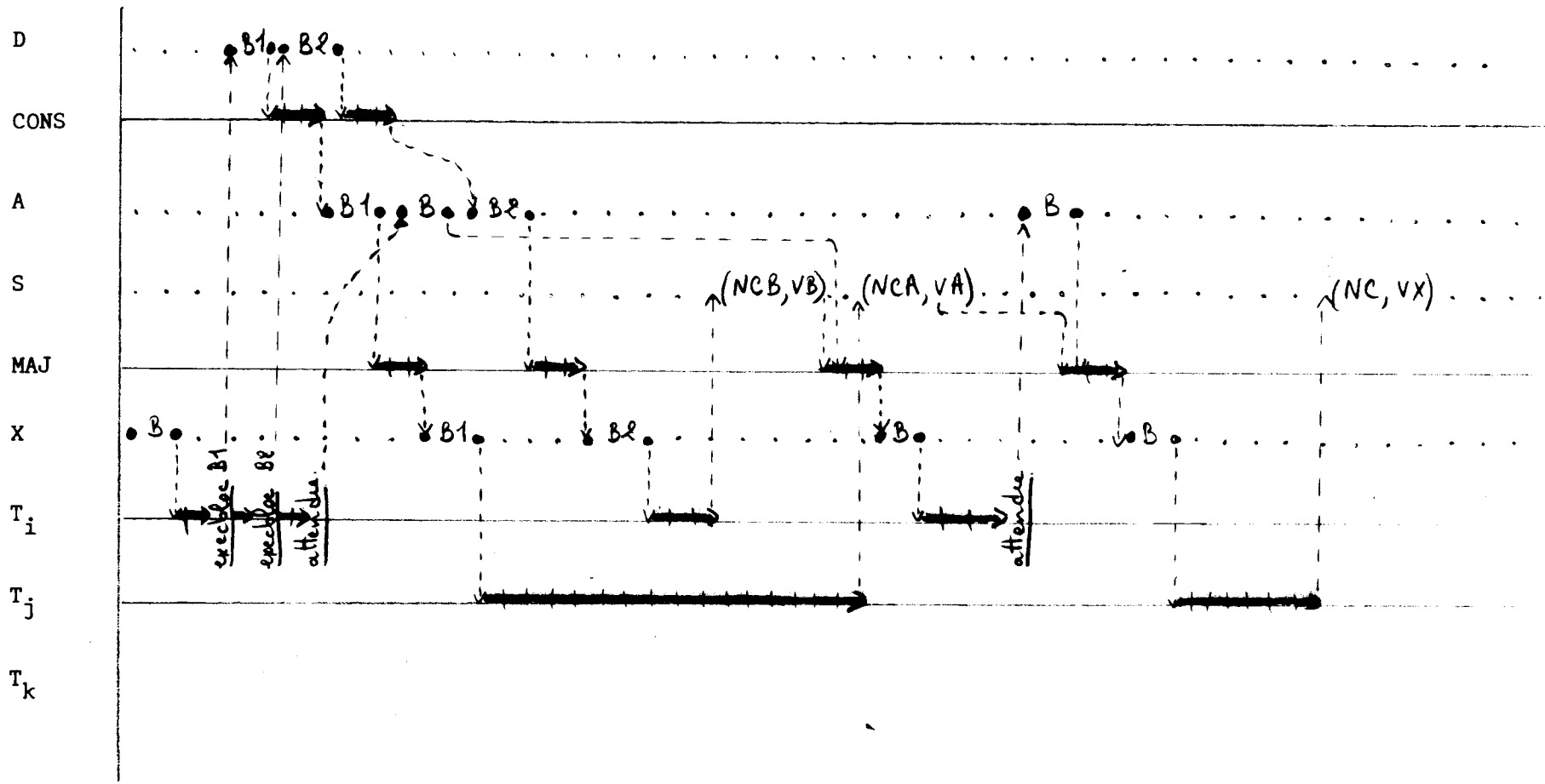
CONS : CONSTRUCTEUR

Exemple 2

```
BLOC B ;
DE (NC1, V1), (NC2, V2) ; DS (NC, non-affecté) ;
LCE (NC1, A), (NC2, B) ; LCS (NC, X) ;
DP (A, -), (B, -), (X, -), (Y, VY), (Z, VZ), (R1, -), (R2, -) ;
   .
   .
   .
   execbloc (B1 ; VY ; R1) ;
   execbloc (B2 ; VZ ; R2) ;
   attendre (R2) ;
   .
   .
   .
   attendre (R1) ;
   .
   .
   .
FIN BLOC
```

MOMENT	DE	LCE	LCS	DS
prise en charge de B	(NC1, V1), (NC2, V2)	(NC1, A), (NC2, B)	(NC, X)	(NC, *non-affecté*)
début d'exécution	/	/	idem	idem
après le 1er execbloc	/	(NCA, R1)	idem	idem
après le 2eme execbloc	/	(NCA, R1), (NCB, R2)	idem	idem
après le 1er attendre	(NCB, *non-affecté*)	(NCA, R1), (NCB, R2)	idem	idem
reprise du BLOC B	(NCB, VB)	(NCA, R1), (NCB, R2)	(NC, X)	(NC, *non-affecté*)
suite d'exécution	/	(NCA, R1)	idem	idem
après le 2ème attendre	(NCA, *non-affecté*)	(NCA, R1)	idem	idem
reprise du BLOC B	(NCA, VA)	(NCA, R1)	(NC, X)	(NC, *non-affecté*)
suite d'exécution	/	/	idem	idem
fin d'exécution	/	/	idem	(NC, VX)

Exemple 2. : Evolution des parties interface et communication du BLOC B.



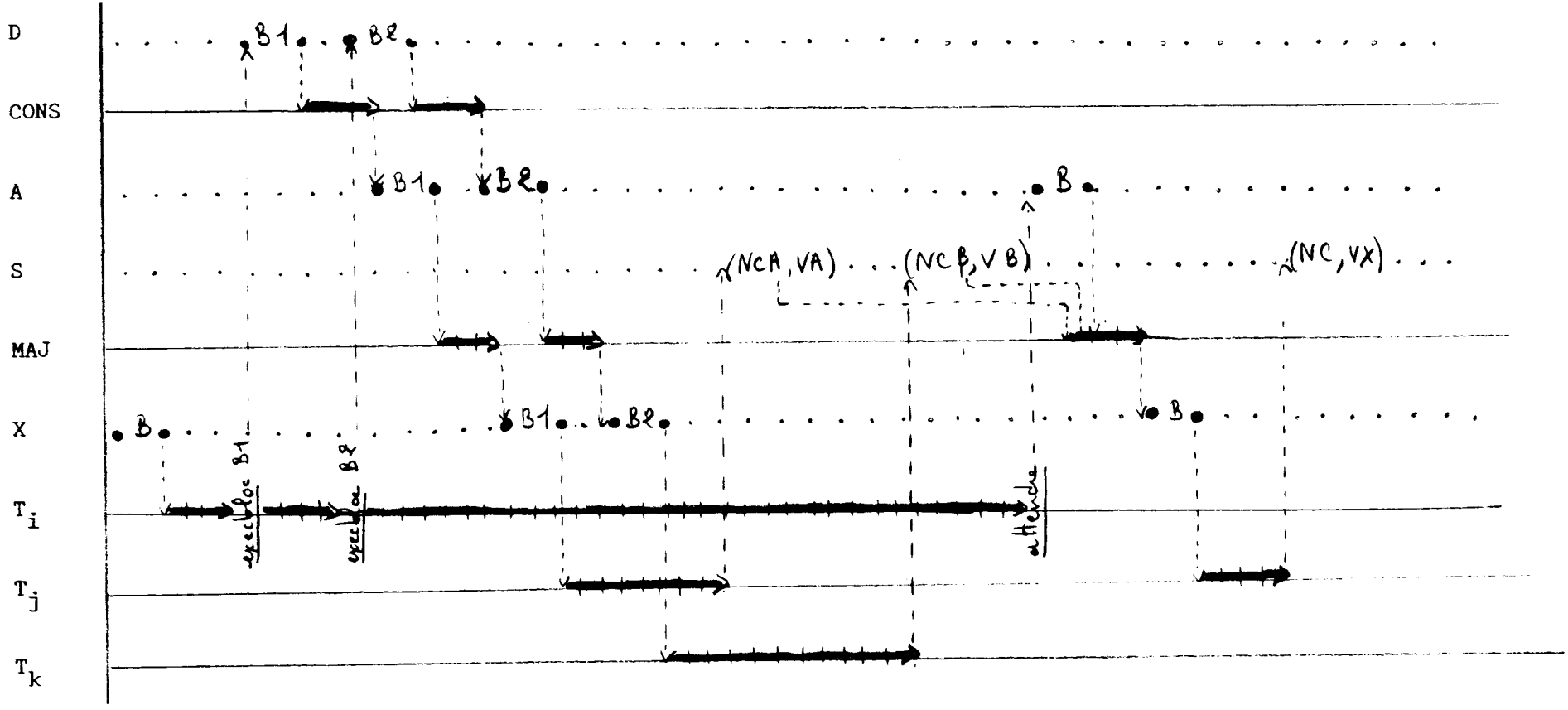
Exemple 2. : Une exécution possible.

Exemple 3

```
BLOC B ;
DE (NC1, V1), (NC2, V2) ; DS (NC, non-affecté) ;
LCE (NC1, A),(NC2, B) ; LCS (NC, X) ;
DP (A, -), (B, -), (X, -), (Y, VY), (Z, VZ), (R1, -), (R2, -) ;
   .
   .
   .
   execbloc (B1 ; VY ; R1) ;
   execbloc (B2 ; VZ ; R2) ;
   .
   .
   .
   attendre (R1, R2) ;
   .
   .
   .
FIN BLOC
```

MOMENT	DE	LCE	LCS	DS
prise en charge de B	(NC1, V1), (NC2, V2)	(NC1, A), (NC2, B)	(NC, X)	(NC, *non-affecté*)
début d'exécution	/	/	idem	idem
après le 1er execbloc	/	(NCA, R1)	idem	idem
après le 2ème execbloc	/	(NCA, R1), (NCB, R2)	idem	idem
après attendre	(NCA, *non-affecté*) (NCB, *non-affecté*)	(NCA, R1), (NCB, R2)	idem	idem
reprise du BLOC B	(NCA, VA), (NCB, VB)	(NCA, R1), (NCB, R2)	(NC, X)	(NC, *non-affecté*)
suite d'exécution	/	/	idem	idem
fin d'exécution	/	/	idem	(NC, VX)

Exemple 3. : Evolution des parties interface et communication du BLOC B.



Exemple 3. : Une exécution possible.

9. GESTION DES NOMS DE COMMUNICATION

La création de nouveaux BLOCS pendant l'exécution d'un programme ne doit pas perturber le contrôle dirigé par les données. Pourtant comme les BLOCS effectivement utilisés ne sont connus qu'au moment de l'exécution, un programmeur ou un compilateur ne sont plus capables d'appliquer la règle d'Assignation Unique pour les *objets de communication* des BLOCS qui ne forment pas le *programme initial*, c'est-à-dire tous ceux qui seront demandés par la primitive execbloc. La gestion des *noms de communication* est donc complètement prise en charge par MAUD. D'ailleurs, il est essentiel que le programmeur n'aie pas accès aux *noms de communication* : il ne peut alors les utiliser pour des usages non prévus, ce qui garantit la sécurité du déroulement.

Lors de l'exécution, il est nécessaire de créer de nouveaux *noms de communication*. Deux stratégies sont envisageables :

* soit il existe dans MAUD un fournisseur unique de *noms de communication* qui est consulté par tous les opérateurs de traitement pour obtenir des *noms de communication* lors de l'exécution des primitives execbloc.

* soit chaque opérateur de traitement possède son propre générateur de *noms de communication*. Dans ce cas, pour être tous différents les *noms de communication* devront être caractéristiques de l'opérateur qui les a construits.

Cependant, à partir d'un programme utilisateur en langage évolué, ou un programme écrit en Assignation Unique par BLOCS, il faudra bien construire un programme exécutable par MAUD, avec parties communication et interface comprenant des *noms de communication* différents de tous ceux qui seront donnés lors de l'exécution.

Si pour des raisons de sécurité de fonctionnement, il est préférable que les *noms de communication* ne soient jamais connus d'avance, même pour un compilateur, la solution suivante est la seule envisageable : elle consiste à stocker en BIBLIOTHEQUE tous les BLOCS qui seront éventuellement utilisés lors de l'exécution du programme, et à n'avoir qu'un seul BLOC au début du traitement de chaque programme. Ce BLOC initial doit être *exécutable* et comprendre des primitives execbloc pour lancer l'exécution des autres BLOCS du programme. Dans ce cas, tous les *noms de communication* sont fabriqués par MAUD.

Si cette solution n'est pas envisageable, il en existe trois autres :

* un certain nombre de *noms de communication* sont connus d'avance et utilisables par le compilateur ou le préprocesseur, avec tous les aléas que cela suppose (quantité de noms, utilisation incorrecte...)

* ui les *noms de communication* sont créés par les opérateurs de traitement, le compilateur peut être considéré comme l'opérateur T_0 par exemple

* le compilateur ou le préprocesseur peuvent être exécutés par MAUD

Cependant, nous conservons l'idée d'un fournisseur de noms, source exclusive de tous les *noms de communication* .

FOURNISSEUR DE NOMS DE COMMUNICATION

Il est possible d'envisager l'existence dans MAUD d'un opérateur fournisseur de *noms de communication*. Cet opérateur supplémentaire délivre aux opérateurs de traitement les noms dont ils ont besoin. Pour respecter les principes du modèle, les communications entre fournisseur de noms et opérateurs de traitement ne peuvent se faire que par l'intermédiaire d'un ensemble N de *noms de communication*.

* le fournisseur de noms place des *noms de communication* dans l'ensemble N,

* à chaque fois qu'un opérateur de traitement a besoin d'un *nom de communication*, il l'extrait de l'ensemble N.

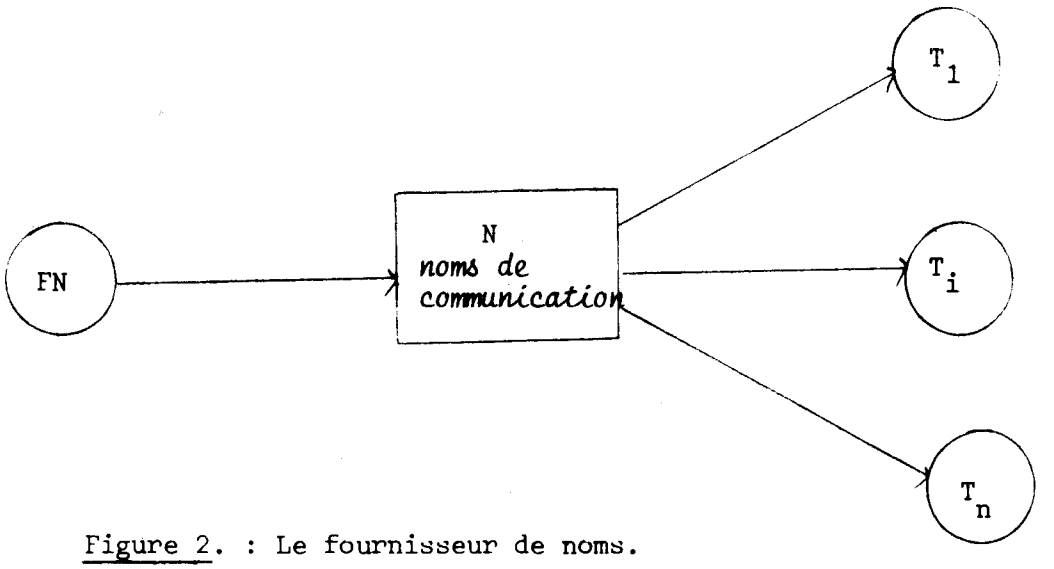


Figure 2. : Le fournisseur de noms.

Le rôle de l'opérateur FN est de maintenir à tout instant suffisamment de *noms de communication* disponibles pour les opérateurs de traitement. Cependant, il peut exister des cas où des *noms de communication* utilisés pourrait lui être rendus. Par exemple, si lors de l'exécution de la primitive execbloc la transmission des valeurs se faisait par position (§ 6.3.), les *noms de communication* ne figureraient plus dans les *domaines d'entrée* des *BLOCS exécutables* ; l'opérateur MAJ pourrait les restituer au Fournisseur de Noms dès qu'il les a utilisés.

CHAPITRE IV

PROPOSITION D'ARCHITECTURE

0 - INTRODUCTION

Le modèle décrit apparait comme suffisamment original par rapport aux architectures classiques pour justifier une étude complète. Nos objectifs sont de proposer :

- un système peu coûteux et facilement extensible, c'est-à-dire qu'une augmentation du nombre de processeurs ne devrait pas entraîner trop de modifications. Ces processeurs devront être de type classique. Compte-tenu de l'accroissement continu des performances des microprocesseurs et du fait que l'exécution se partage entre un nombre de sites qui peut être élevé, une puissance importante peut être obtenue même si l'on s'impose de ne faire appel qu'à des composants classiques.

- une gestion mémoire relativement simple avec le moins de problèmes de synchronisation possible.

1. CARACTERISTIQUES D'UNE ARCHITECTURE ADAPTEE AU MODELE

Le modèle qui vient d'être décrit présente trois caractéristiques principales :

1) les opérateurs n'ont pas de liaison directe entre eux, toutes les communications se font à l'aide d'ensembles homogènes d'informations. L'architecture choisie doit donc comporter une unité de stockage partageable servant en même temps d'outil de communication.

2) l'utilisation d'un BLOC passe par plusieurs étapes, comme l'indique la figure 1.

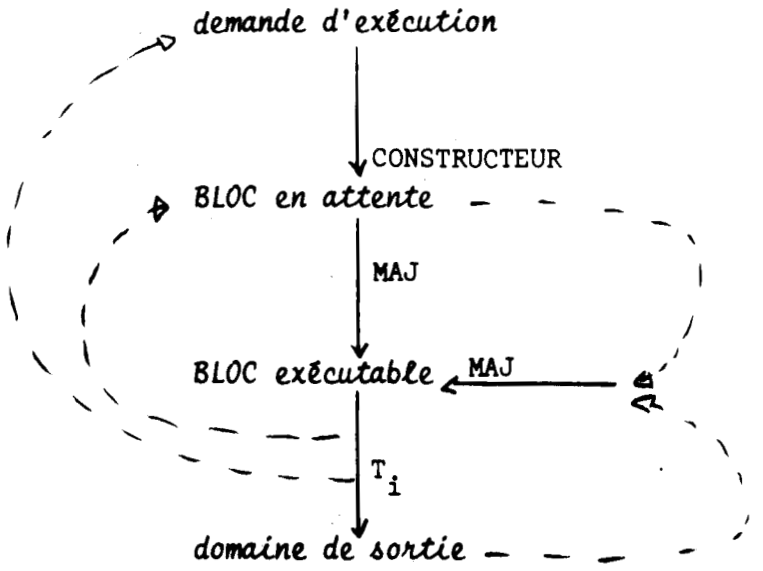


Figure 1. : Etapes de la vie d'un BLOC.

A chaque étape intermédiaire, l'objet est traité par un processeur qui le transforme en un objet d'un autre type. L'ordre des transformations n'est pas quelconque : en partant d'une *demande d'exécution*, les transformations sont faites successivement par le CONSTRUCTEUR, l'opérateur MAJ, un processeur de traitement. Il existe donc un certain recouvrement dans les opérations à effectuer et il est intéressant d'en tenir compte par une architecture présentant des caractéristiques de fonctionnement en pipeline.

3) au niveau de l'opérateur MAJ, il est souhaitable que l'accès à l'ensemble A des *BLOCS en attente* et à l'ensemble S des *domaines de sortie* puissent se faire de manière associative puisque c'est par des noms que s'effectuent les opérations de communication entre BLOCS.

Dans une solution classique, l'utilisation d'une mémoire commune contenant tous les objets de MAUD : *demandes d'exécution, BLOCS exécutables, BLOCS en attente, domaines de sortie,* est peu justifiée. Par contre, les *BLOCS exécutables* et les *BLOCS en attente* peuvent être réunis dans une même mémoire de BLOCS, un indicateur précisant l'état du BLOC, *exécutable* ou *en attente*. L'accès à un BLOC pourrait se faire de manière associative par consultation des indicateurs.

L'ensemble D des *demandes d'exécution* peut être représenté par une file d'attente.

Dans ce contexte, il est important que l'ensemble S des *objets de sortie* soit représenté dans une mémoire possédant un accès associatif pour l'opérateur MAJ. En effet, les conflits d'accès à cette mémoire risquent d'être assez nombreux : les opérateurs de traitement ne font qu'y déposer des *domaines de sortie* ; mais l'opérateur MAJ la consulte de façon permanente afin de compléter les *domaines d'entrée* des *BLOCS en attente*.

Dans une telle solution, les mémoires doivent être accessibles en lecture et en écriture simultanément par plusieurs opérateurs. De plus, le nombre d'objets qui y sont stockés est extrêmement variable. Comme ils ne sont utilisés qu'une fois, ils peuvent être supprimés dès qu'ils ont été utilisés par un opérateur ; ce qui peut conduire à une fragmentation de la mémoire et rend par conséquent délicate la gestion de la place disponible. D'autre part, nous ne souhaitons pas nous limiter au point de vue structures de données utilisables ; c'est-à-dire que nous souhaitons pouvoir traiter

non seulement des entiers, des réels, des booléens et des caractères, mais aussi des vecteurs et des structures.

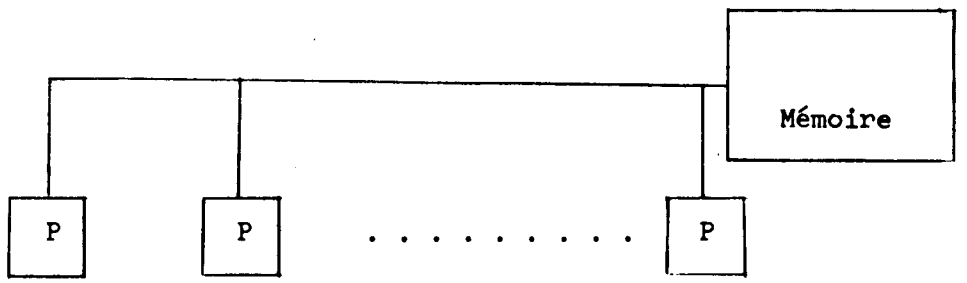
Pour faciliter l'accès de tous les opérateurs à la mémoire commune, il parait souhaitable que cette mémoire soit divisée en blocs addessibles séparément, ainsi les conflits d'accès n'auraient plus lieu qu'au niveau d'un bloc.

2. COMMUNICATION ENTRE PROCESSEURS PAR PAR UNE MEMOIRE COMMUNE

2.1. SOLUTIONS CLASSIQUES

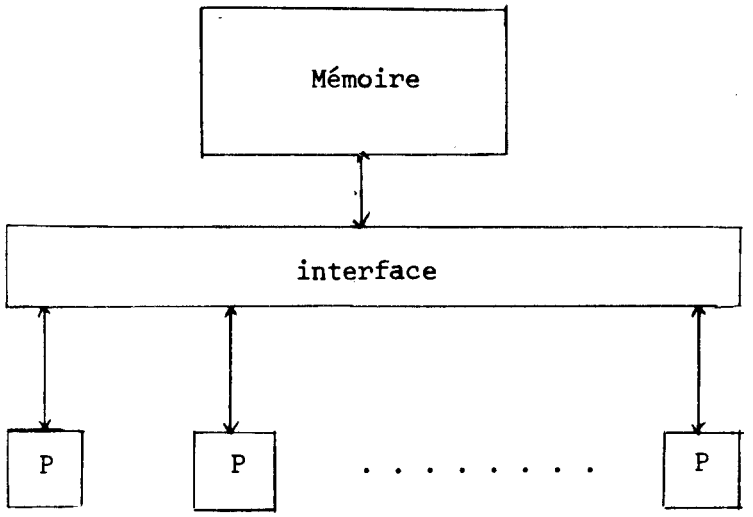
De nombreuses solutions ont été étudiées en ce qui concerne la communication entre processeurs par l'intermédiaire d'une mémoire commune. Il existe principalement trois techniques :

1) l'accès à la mémoire se fait par un bus commun



Dans ce cas, les accès à la mémoire sont règlés par les processeurs eux-mêmes. L'inconvénient ici est qu'une seule communication avec la mémoire est possible à la fois.

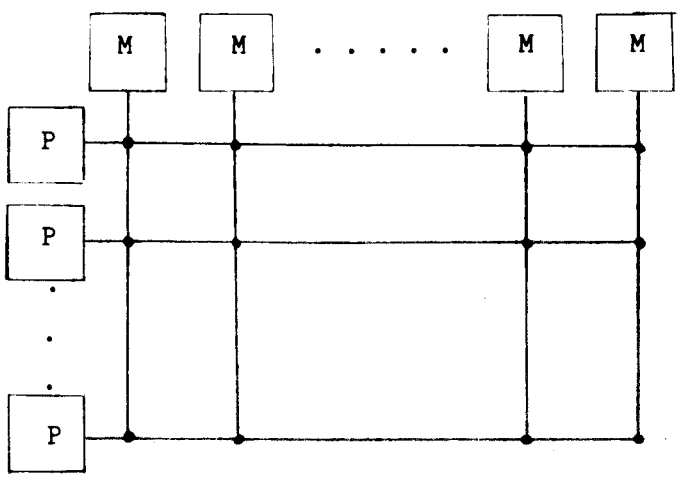
2) l'accès à la mémoire se fait par un interface auxquels sont connectés tous les processeurs, chaque processeur ayant son propre bus de communication. Dans ce cas, les accès à la mémoire sont réglés par l'interface.



Même dans le cas où le nombre d'accès à l'interface est suffisamment grand, il existe un goulot d'étranglement entre celui-ci et la mémoire proprement dite. L'entrelacement permet de résoudre en partie ce problème mais impose une gestion des conflits et par conséquent une logique complexe supplémentaire.

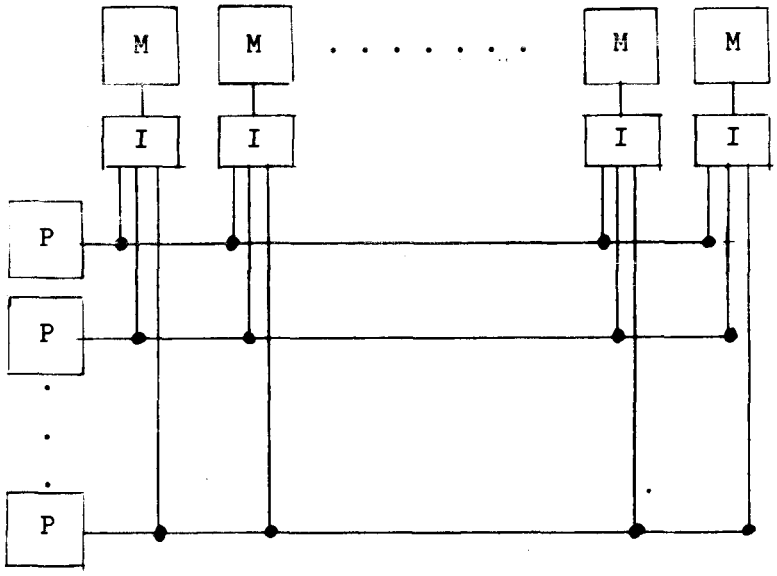
3) Afin d'augmenter le nombre d'accès simultanés à la mémoire, celle-ci est souvent divisée en un certain nombre de blocs séparés et accessibles indépendamment. L'accès à la mémoire peut se faire alors principalement de trois façons :

3.1.) par un cross-bar



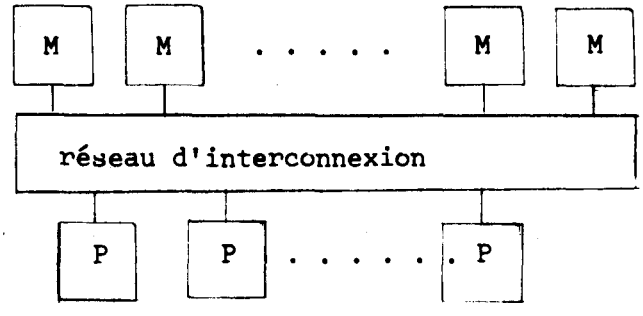
l'accès à chaque bloc de mémoire se fait au moyen d'un bus commun à tous les processeurs. Mais la complexité du cross-bar croît comme le produit du nombre de blocs mémoire par le nombre de processeurs.

3.2.) par un interface mémoire associé à chaque bloc



les conflits d'accès n'apparaissent plus qu'au niveau de chaque bloc mémoire et sont résolus par l'interface associé à chaque bloc. Cette solution présente encore l'inconvénient d'être difficilement extensible et se révèle très lourde.

3.3.) par un réseau d'interconnexion



le réseau d'interconnexion permet de relier n'importe quel processeur à n'importe quel bloc de mémoire. Cependant, le nombre maximum d'accès simultanés est limité : le taux de simultanéité est limité au maximum par le nombre de blocs mémoires indépendants.

Chacune des solutions évoquées présente des inconvénients en regard des objectifs que nous nous sommes fixés pour une réalisation matérielle, c'est-à-dire :

- facilités d'extension
- accès à la mémoire possible simultanément pour tous les processeurs.
- contrôle d'accès simple a réaliser
- compatibilité avec les types d'accès du modèle du chapitre précédent.

2.2. UTILISATION D'UNE MÉMOIRE CIRCULANTE

La solution que nous proposons utilise un anneau de mémoire circulante. La principale caractéristique d'une mémoire circulante est son mode d'accès qui est séquentiel. Un module de mémoire circulante se compose de trois parties :

* un milieu de propagation dans lequel les bits sont rangés dans des cellules qui communiquent avec leurs voisines sous contrôle d'une horloge commune ;

* un point d'entrée où les informations à écrire sont présentées séquentiellement ;

* un point de sortie où l'information apparait après avoir effectué un parcours complet à travers le milieu de propagation.

Très souvent, un circuit (externe ou interne) permet de réintroduire au point d'entrée l'information apparaissant au point de sortie, assurant ainsi une circulation permanente des données stockées. Les points d'entrée et de sortie confondus s'appellent "fenêtre d'accès". L'information est utilisable pour un usage externe uniquement lorsqu'elle traverse cette fenêtre. Des détails concernant la technologie et l'utilisation des mémoires circulantes peuvent être trouvées en [Cor 78], [Van 79].

La solution choisie est celle d'une mémoire circulante commune à tous les processeurs ; chaque processeur possède cependant une mémoire propre qui lui permet de réaliser sa fonction sans pénalisation pour l'ensemble du système.

La mémoire commune est constituée en disposant en série plusieurs modules de mémoire circulante et en bouclant le dernier module de la série sur le premier afin de former un anneau. La liaison entre deux modules constitue la fenêtre d'accès. Les différents processeurs sont répartis autour de l'anneau et placés devant les fenêtres d'accès (figure 2).

L'anneau de mémoires circulantes a d'intéressantes caractéristiques en regard de nos objectifs : il permet des accès multiples sans poser de problème de synchronisation. De plus, grâce au principe même des mémoires circulantes, il représente un outil de communication.

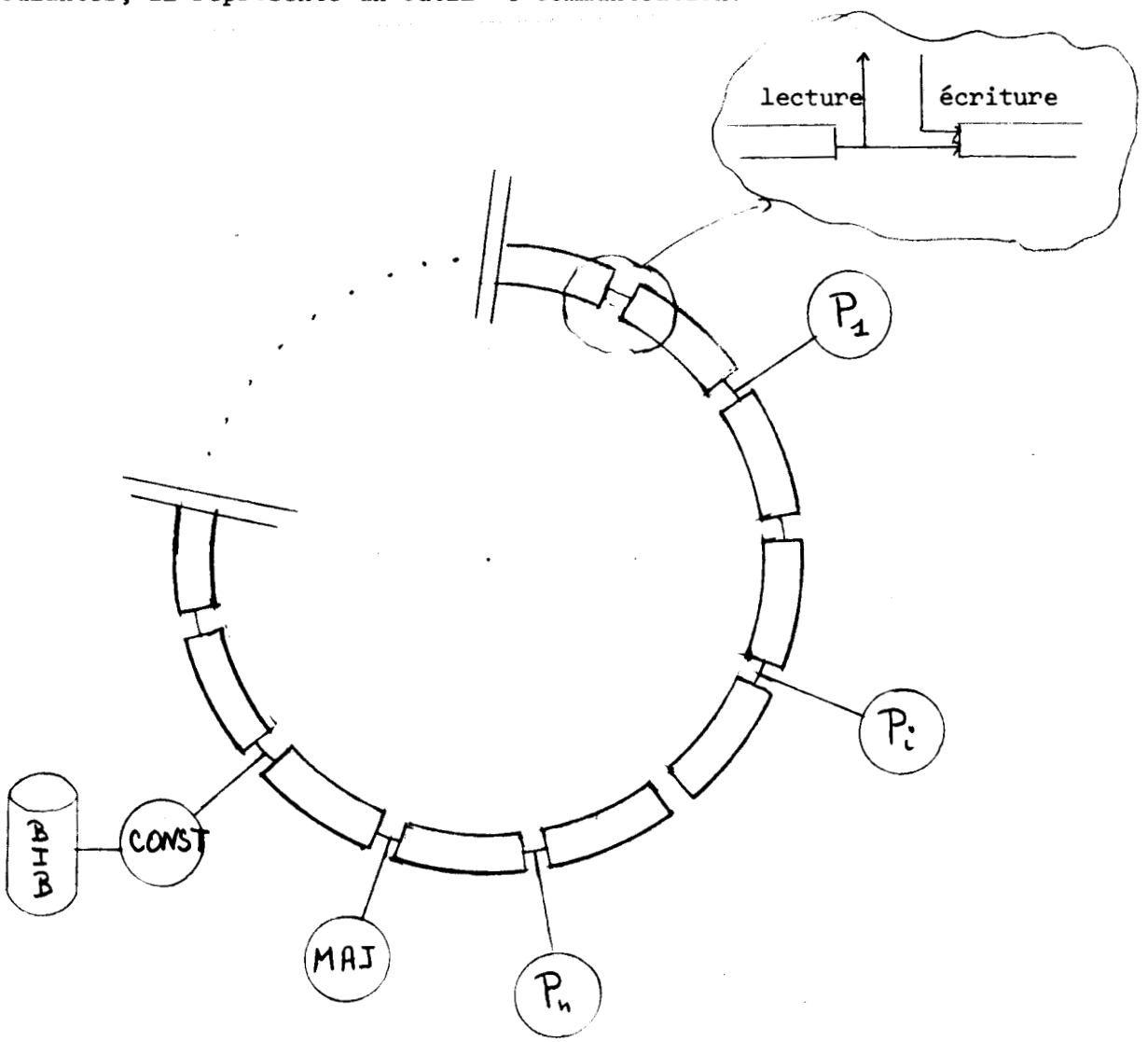


Figure 2. : Une architecture pour MAUD.

3. REALISATION DES OPERATEURS

Les opérateurs de traitement exécutent des ensembles d'instructions c'est-à-dire que leurs fonctions sont celles d'un processeur classique. Plutôt que de fabriquer un processeur répondant exactement à nos besoins, nous avons choisi d'utiliser des microprocesseurs standards, (probablement des micro-processeurs 16 bits). Ceci pour des raisons de coût, de disponibilité, et par la suite d'extensibilité, notre but étant de faire évoluer le système au fur et à mesure de l'évolution des composants. Il faudra cependant ajouter à ces microprocesseurs les quatre primitives nécessaires pour le modèle : début-de-bloc, fin-de-bloc, execbloc, attendre.

Quant aux opérateurs CONSTRUCTEUR et MAJ, ils sont spécifiques, mais pourront être réalisés à partir de microprocesseurs microprogrammables ou même de microprocesseurs standards, les fonctions du CONSTRUCTEUR et de MAJ étant relativement simples et plus liées à une gestion d'espace mémoire qu'à des contraintes très spécifiques de traitement.

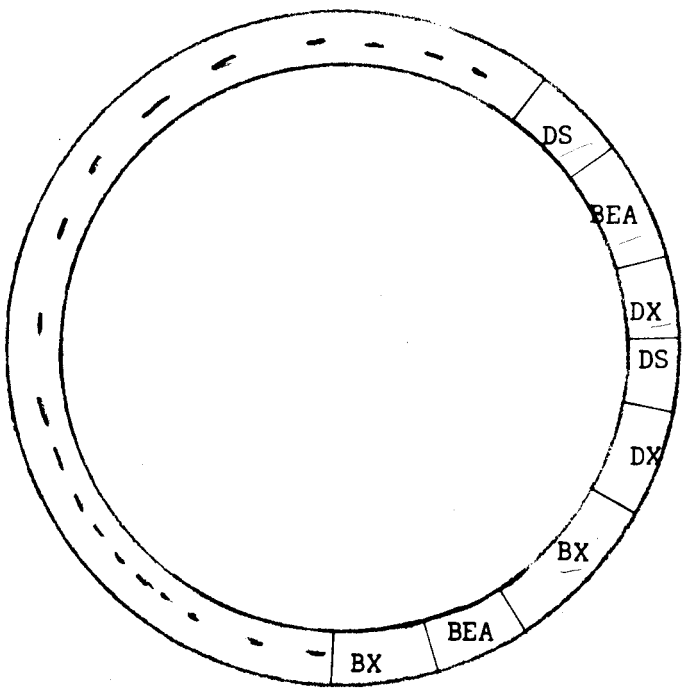
4. LA MEMOIRE DE MAUD

4.1. ORGANISATIONS POSSIBLES

La mémoire commune contient tous les objets utilisés par MAUD : *BLOCS exécutables, BLOCS en attente, domaines de sortie, demandes d'exécution.* D'autres types d'objets seront peut-être nécessaires pour les communications avec les périphériques ou les erreurs. Mais de toute façon leur forme sera très proche de celle des objets cités ci-dessus, et de plus leur passage dans l'anneau sera très transitoire. Le stockage des différents objets dans l'anneau est un problème difficile compte-tenu de leurs tailles qui sont très diverses et du nombre d'objets de chaque type qui est très variable tout au long de l'exécution du programme. Trois cas peuvent être envisagés :

1) l'anneau est divisé en quatre parties, de longueur égale ou non, correspondant aux quatre ensembles d'objets, chacune regroupant les objets d'un même type. Cette solution ne ferait que multiplier les problèmes de gestion par quatre et ne serait même pas adoptée dans une mémoire classique.

2) les objets sont placés dans l'anneau les uns derrière les autres dans un ordre quelconque. Afin de les distinguer les uns des autres, un indicateur est placé en tête de chaque objet et désigne le type de l'objet. Ces indicateurs doivent être différents de toutes les autres informations pouvant être placées dans l'anneau.



DS : *domaine de sortie*
BEA : *BLOC en attente*
BX : *BLOC exécutable*
DX : *demande d'exécution*

La gestion de l'espace disponible risque d'être difficile car on retrouvera les problèmes bien connus d'émiettement. Cependant, grâce au principe des mémoires circulantes, il serait envisageable d'avoir un processeur "compacteur" qui effectue de façon permanente le tassement des objets de façon à n'avoir qu'un seul "trou" utilisable : la place libérée par un processeur retirant un objet de l'anneau ne serait utilisable qu'après passage dans le compacteur. Ce dernier devrait alors posséder deux fenêtres d'accès à l'anneau et une assez grosse mémoire. De plus, toutes les communications entre les

processeurs et l'anneau seraient plus difficiles : les processeurs devant observer de façon permanente le contenu de leur propre fenêtre à la recherche d'un indicateur d'objet pouvant se trouver n'importe où dans l'anneau. En outre, les temps de dépôt risquent d'être longs puisqu'il faudrait attendre LE "trou". Cette solution est cependant difficile à réaliser d'un point de vue matériel.

3) l'anneau est divisé en secteurs logiques de taille identique, appelés *cadres*. Les objets sont déposés dans ces *cadres* qui circulent dans l'anneau. Le temps de défilement de la mémoire circulante étant constant, les processeurs ne doivent plus consulter l'anneau qu'à intervalles réguliers, c'est-à-dire lorsque le début d'un *cadre* passe devant leur fenêtre d'accès.

Afin de ne pas retrouver les inconvénients de la solution précédente, la répartition des objets dans les *cadres* ne doit pas être quelconque : il ne faut placer que des objets de même type dans un même cadre. Le contenu du *cadre* doit alors être désigné par un indicateur qui prend des valeurs correspondant aux différents types d'objets utilisés par MAUD :

- *BLOC exécutable* ,
- *BLOC en attente* ,
- *domaine de sortie* ,
- *demande d'exécution* ,

et quelques autres pour les échanges avec l'extérieur, etc... qui seront nécessaires par la suite.

L'indicateur d'un *cadre* n'est pas fixé une fois pour toutes : le contenu d'un *cadre* change à chaque fois qu'il est utilisé par un processeur. En particulier il doit exister un indicateur "vide" qui désigne un *cadre* utilisable par n'importe quel processeur pour le dépôt d'un objet.

Le choix d'une taille particulière pour les *cadres* doit tenir compte de considérations technologiques aussi bien que de la taille des objets qu'ils doivent contenir. Il semble souhaitable de pouvoir ranger dans un *cadre* un

BLOC tout entier, sans que la partie interne d'un BLOC, instructions + objets locaux, soit trop réduite, ce qui conduirait à multiplier le nombre de BLOCS constituant un programme et donc augmenter le volume de communications nécessaire pour son exécution. Dans cette optique, une taille de 1 K mots (16 bits) semble acceptable (elle relève sensiblement des mêmes critères de choix que la taille des pages en contexte paginé). Cependant, le choix d'une taille particulière pour les *cadres* implique des contraintes sur la taille des BLOCS fabriqués à la compilation, car tout le contexte d'exécution d'un BLOC doit pouvoir être rangé dans un *cadre* à tout moment lors de l'exécution d'une primitive attendre. Si ces contraintes paraissent trop importantes il serait alors envisageable de choisir des *cadres* plus petits, par exemple 256 mots, et d'adopter pour les BLOCS une solution du même genre que celle décrite dans le paragraphe II.4.2. : chaque BLOC est décomposé en au moins deux parties, toutes de même taille. D'une part, le *domaine d'entrée* ; d'autre part, un ou plusieurs morceaux constituant le BLOC sans son *domaine d'entrée*. Cela éviterait d'avoir des contraintes quant à la taille des BLOCS. Cette dernière serait seulement limitée par la taille de la mémoire des processeurs de traitement.

Si un BLOC tout entier tient dans un *cadre*, les *demandes d'exécution* et *domaines de sortie* sont obligatoirement plus petits qu'un *cadre* puisqu'ils sont construits avec des informations contenues dans un BLOC. Il serait donc possible dans certains cas d'en ranger plusieurs dans un seul *cadre*. Cependant, pour que les échanges entre processeurs et anneau restent très simples, nous avons choisi de ne placer qu'un seul objet par *cadre*. Dans la majorité des cas, la durée de vie dans l'anneau d'un *domaine de sortie* et surtout d'une *demande d'exécution* sera très courte par rapport à la durée de vie des BLOCS, les *cadres* contenant ces deux types d'objets seront donc vite libérés. D'autre part, les problèmes d'accès et de gestion de l'anneau étant simples, le traitement global y gagne en rapidité.

4.2. PROCÉDURES D'ÉCHANGE ENTRE LES PROCESSEURS ET L'ANNEAU

Les processeurs n'entament une communication avec l' anneau qu'à intervalles réguliers, au moment où un début de *cadre* passe devant leur fenêtre, en lisant l'indicateur de *cadre*.

Un processeur désirant déposer un objet dans l'anneau recherche un *cadre* ayant un indicateur "vide" ; un processeur libre recherche un *cadre* important un indicateur en rapport avec son activité et son état. Chaque lecture ou écriture dans l'anneau entraîne le changement d'état du *cadre* suivant le tableau 3. Si chaque processeur dispose d'un module de mémoire circulante de taille égale à celle d'un *cadre*, il est même possible de faire physiquement cet échange par commutation matérielle en isolant momentanément un module [Cor 79].

		Cadre	
Processeur	**Action**	**Indicateur avant**	**Indicateur après**
TRAITEMENT	lire	BLOC exécutable	VIDE
	écrire	VIDE	demande d'exécution
	écrire	VIDE	BLOC en attente
	écrire	VIDE	domaine de sortie
MAJ	lire	BLOC en attente	VIDE
	lire	domaine de sortie	VIDE
	écrire	VIDE	BLOC exécutable
CONSTRUCTEUR	lire	demande d'exécution	VIDE
	écrire	VIDE	BLOC en attente

Tableau 3.

La réalisation des échanges est confiée à une unité d'échange placée entre l'anneau et le processeur. Dans la plupart des solutions proposées par la suite, un automate suffira à assurer le contrôle des échanges entre la mémoire circulante et la mémoire propre du processeur (Figure 4). Cet automate, outre les fonctions d'accès à l'anneau, assure un accès direct à la mémoire locale du processeur.

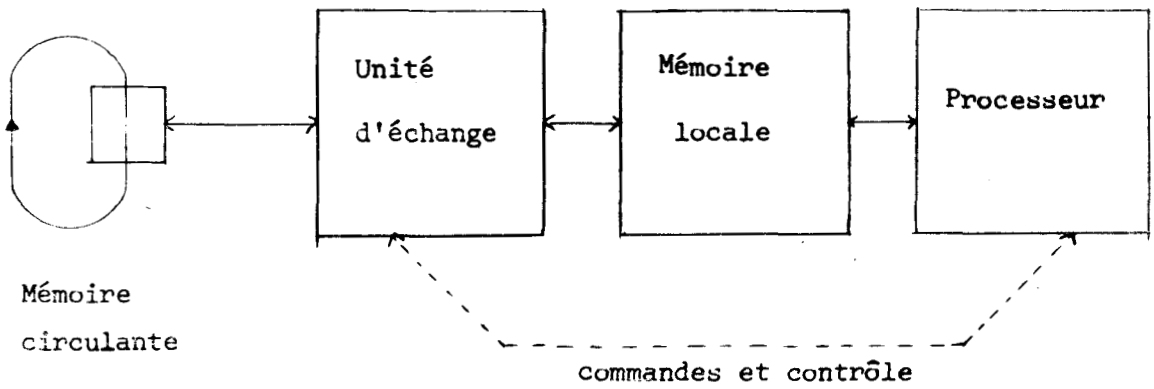


Figure 4. : Echanges entre les processeurs et l'anneau.

4.2.1. ECHANGES ENTRE PROCESSEURS DE TRAITEMENT ET ANNEAU

Les procédures d'échange entre les processeurs de traitement et l'anneau sont extrêmement simples :

Ecriture

* recherche d'un *cadre* possédant l'indicateur "VIDE", c'est-à-dire lecture de l'indicateur de tous les *cadres* passant devant la fenêtre d'accès à l'anneau.

* lorsqu'il est trouvé, dépôt dans le *cadre* d'un objet avec l'indicateur du type de l'objet : *domaine de sortie, BLOC en attente, demande d'exécution.*

Lecture

* recherche d'un *cadre* possédant l'indicateur "*BLOC exécutable*", c'est-à-dire lecture de l'indicateur de tous les *cadres* passant devant la fenêtre.

* lorsqu'il est trouvé, remplacement de l'indicateur par "vide" puis lecture du *cadre* tout entier.

4.2.2. ECHANGES ENTRE PROCESSEUR CONSTRUCTEUR ET L'ANNEAU

Les procédures d'échange avec l'anneau sont identiques à celles des processeurs de traitement, aux indicateurs près.

Ecriture

* recherche d'un *cadre* possédant l'indicateur "VIDE", c'est-à-dire lecture de l'indicateur de tous les *cadres* passant devant la fenêtre d'accès à l'anneau.

* lorsqu'il est trouvé, dépôt dans le *cadre* d'un *BLOC en attente* avec l'indicateur adéquat.

Lecture

* recherche d'un *cadre* possédant l'indicateur "*demande d'exécution*", c'est-à-dire lecture de l'indicateur de tous les *cadres* passant devant la fenêtre.

* lorsqu'il est trouvé, remplacement de cet indicateur par "vide" puis lecture du *cadre*.

Le CONSTRUCTEUR pourrait contrôler le type du BLOC qu'il construit : si le *domaine d'entrée* est *complet*, c'est un *BLOC exécutable*, sinon c'est un *BLOC en attente* (il suffit de regarder s'il existe ou non une valeur *non-affecté* dans le *domaine d'entrée*) ; et le BLOC serait alors placé dans l'anneau avec l'indicateur correspondant. Cela permettrait de décharger partiellement le processeur MAJ sans ajouter trop à la charge du CONSTRUCTEUR : le contrôle sur l'état du BLOC et la recherche du BLOC en BIBLIOTHEQUE peuvent être effectués simultanément.

4.2.3. ECHANGES ENTRE PROCESSEUR MAJ ET ANNEAU

Le processeur MAJ est le processeur principal de MAUD, bien qu'il ne soit pas un processeur maître. Ses performances risquent d'être critiques vis-à-vis de la vitesse de traitement du système. Il se distingue des autres

processeurs, au niveau des communications avec l' anneau, car il est concerné en lecture par deux types d'objets : les *BLOCS en attente* et les *domaines de sortie*.

Si les seuls échanges possibles avec l'anneau sont la lecture et l'écriture d'un *cadre* entier, le processeur MAJ doit ranger dans sa mémoire propre tous les *BLOCS en attente* et tous les *domaines de sortie* qui passent devant sa fenêtre. Dans ce cas, l'anneau est réduit pratiquement à un simple outil de communication : les seuls objets qui peuvent rester dans l'anneau plus d'un tour sont des *BLOCS exécutables*. Cela n'arrive que lorsque tous les processeurs sont occupés, ce qui traduit d'ailleurs le fait que le traitement se poursuit à son potentiel maximum d'exécution, mais ce qui peut également produire un interblocage par manque de *cadres* vides.

La fonction de stockage de l'anneau étant pour nous essentielle, nous n'étudierons pas cette solution. Deux possibilités restent envisageables :

solution 1) : le processeur MAJ range dans sa mémoire propre tous les *domaines de sortie* et laisse dans l'anneau les *BLOCS en attente* ;

solution 2) : le processeur MAJ range dans sa mémoire propre tous les *BLOCS en attente* et laisse les *domaines de sortie* dans l'anneau.

Une variante de la solution 2 consiste à avoir un certain nombre de *cadres* réservés dont l'indicateur est toujours le même : *domaines de sortie*. Nous l'étudierons dans la solution 3.

5. SOLUTION 1 : BLOCS EN ATTENTE DANS L'ANNEAU

> Le processeur MAJ range dans sa mémoire propre tous les *domaines de sortie* et laisse les *BLOCS en attente* dans l'anneau.

Il possède donc des couples (*nom de communication*, valeur) et doit consulter tous les *BLOCS en attente* qui passent devant sa fenêtre pour savoir s'il ne détient pas un ou plusieurs des *noms de communication* que ces BLOCS attendant, c'est-à-dire que pour chaque *BLOC en attente*, il doit effectuer les opérations suivantes :

> . pour chaque *nom de communication* NC_i du *domaine d'entrée* de ce BLOC
> .. comparer le nom NC_i à tous les noms contenus dans la mémoire propre
> .. s'il existe un couple (NC_i, V_i) dans la mémoire propre, recopier la valeur V_i dans le *domaine d'entrée* du BLOC, dans l'anneau.

Deux problèmes se posent :

1) le processeur MAJ doit comparer un *nom de communication*, se trouvant dans un *cadre* d'indicateur *BLOC en attente*, à tous les *noms de communication* qu'il possède dans l'ensemble des *domaines de sortie* contenus dans sa mémoire propre.

2) le processeur MAJ a besoin d'une procédure d'échange avec l'anneau, autre que lire ou écrire un *cadre* entier, pour faire des lectures et des modifications partielles dans les *BLOCS en attente*. Cette procédure nécessite des accès associatifs à l'anneau.

Ce dernier problème se résoud aisément car il est possible d'ajouter des fonctions associatives au niveau de la fenêtre dans un modèle où les bits d'un mot sont traités en parallèle et les mots en série [Yau 77].

Par contre, la première opération semble à première vue impossible à faire à la volée, c'est-à-dire uniquement pendant le temps de défilement d'un nom dans la fenêtre du processeur MAJ. Une solution consiste à doter le processeur MAJ d'une mémoire hiérarchisée : cette mémoire contenant tous les *noms de communication* utilisables dans MAUD, chaque *nom de communication* étant associé à un pointeur vers les informations que possède éventuellement le processeur MAJ à propos de ces *noms de communication* ; c'est-à-dire la valeur associée à ce *nom de communication* (Figure 5).

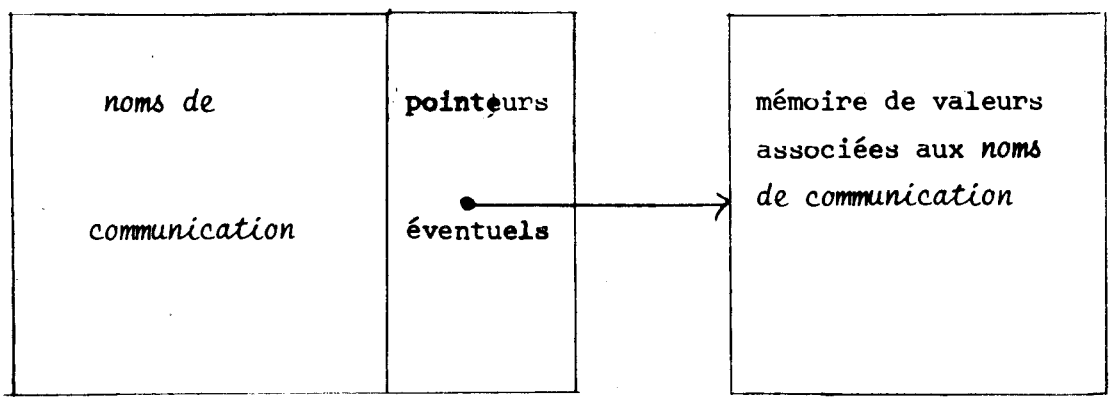


Figure 5.: Une organisation possible de la mémoire de MAJ (solution 1).

Suivant la taille choisie pour un *nom de communication*, l'espace réservé à ces noms est plus ou moins grand : ainsi un nom sur 16 bits demande un espace de 64 K x 16 bits. Cela est très coûteux car en réalité, à ce stade seule l'existence du pointeur est intéressante : elle signifie que le processeur MAJ possède des informations sur le *nom de communication*. Il suffit donc de 1 bit pour savoir si ces informations existent. Dans le cas de noms sur 16 bits, le nom lu dans l'anneau par le processeur MAJ est alors considéré comme une adresse dans une mémoire de 64 K x 1 bit. Si le bit adressé est à 1, le processeur MAJ possède, dans sa mémoire propre, des informations concernant le nom lu, sinon il ne possède rien à son sujet. Toutefois, l'opération d'adressage dans la mémoire de 64 K x 1 bit suivie du test doit se faire à la volée, et il faudra choisir pour la réalisation une mémoire suffisamment rapide.

En outre, deux difficultés propres à la solution 1 apparaissent :

1) la lecture des *noms de communication* dans l'anneau et l'écriture des valeurs qui leur sont associées. En effet, les couples (NC_i, V_i) représentent des objets de types différents donc de tailles différentes. La place occupée par la valeur doit être prévue et laissée libre dans le *domaine d'entrée* du BLOC. Cela peut parfois être difficile, par exemple pour les chaînes de caractères ; mais il est rare qu'on ne connaisse pas une borne maximum pour la taille d'une donnée. De ce fait, la lecture des noms NC_i ne peut se faire à intervalles réguliers si on suppose que les couples (NC_i, V_i) sont placés les uns derrière les autres dans le *domaine d'entrée*. Mais cela peut se résoudre en adoptant une autre organisation pour le *domaine d'entrée*.

2) la mise à jour de l'indicateur du BLOC si le processeur MAJ travaille à la volée. En effet, après avoir vérifié qu'un *domaine d'entrée* est *complet*, le processeur MAJ doit remplacer l'indicateur du *BLOC en attente* concerné par un indicateur *BLOC exécutable*. Or on sait que le BLOC est devenu *exécutable* après avoir vu défiler son *domaine d'entrée*. A ce moment, il est trop tard pour changer l'indicateur qui est en tête du *cadre*. Par conséquent :

* si le processeur MAJ ne possède qu'une seule fenêtre d'accès à l'anneau, il doit recopier complètement le *BLOC en attente* dans sa mémoire propre pour pouvoir éventuellement changer l'indicateur. Dans ce cas, il n'est plus obligatoire de faire la mise à jour du *domaine d'entrée* à la volée puisque le BLOC se trouve dans sa mémoire propre. Cependant, il faudra trouver un *cadre* vide pour ranger le BLOC éventuellement modifié.

* si le processeur MAJ possède deux fenêtres d'accès à l'anneau situées de part et d'autre d'un même module de mémoire circulante, il peut lire le *BLOC en attente* à travers la première fenêtre, et écrire le BLOC éventuellement devenu *exécutable* dans la deuxième fenêtre. Il dispose alors du temps de défilement de la mémoire entre les deux fenêtres pour préparer toutes les mises à jour. On peut alors considérer que le processeur MAJ dispose pour lui seul d'une partie de l'anneau. Dans ce cas, il ne doit pas trouver d'autre *cadre* pour restituer le BLOC éventuellement modifié.

Il faut remarquer que la mémoire propre du processeur MAJ n'est pas nécessairement très importante dans cette solution : elle ne contient que des *domaines de sortie* et de plus dès qu'un couple (NC_i, V_i) a été utilisé, il peut être effacé en vertu de l'utilisation unique (§ II.4.1.).

Particularités de la solution 1

- les procédures d'échange des processeurs avec l'anneau sont très simples ;

- aucune gestion de la place occupée dans un *cadre* n'est supportée par les processeurs ;

- afin d'éviter les problèmes matériels soulevés par le traitement à la volée, il semble souhaitable d'avoir deux fenêtres d'accès à l'anneau pour le processeur MAJ. Mais cela n'est pas un inconvénient majeur s'il en résulte un accroissement des performances de ce processeur. En effet, les mises à jour doivent être faites rapidement pour ne pas freiner la suite du traitement.

6. SOLUTION 2 : DOMAINES DE SORTIE DANS L'ANNEAU

Le processeur MAJ range dans sa mémoire propre tous les *BLOCS en attente* et laisse les *domaines de sortie* dans l'anneau.

On désignera par espace de sortie l'ensemble formé par tous les *cadres* de l'anneau contenant des *domaines de sortie*. Lorsqu'un *domaine de sortie* passe devant la fenêtre, le processeur MAJ doit :

. pour chaque couple (NC_i, V_i) de ce *domaine de sortie*

.. comparer le nom NC_i à tous les noms se trouvant dans les *domaines d'entrée* DE_j des *BLOCS en attente* se trouvant dans sa mémoire propre.

.. si ce nom NC_i figure dans un DE_j, lire la valeur V_i dans le *cadre* et la recopier dans ce DE_j.

Des problèmes du même genre que ceux rencontrés dans la solution 1 se posent, c'est-à-dire :

1) la nécessité d'avoir des accès associatifs à l'anneau.

2) la comparaison à la vitesse de défilement de l'anneau d'un *nom de communication*, figurant dans un *cadre* possédant l'indicateur *domaine de sortie*, à tous les *noms de communication* possédés par le processeur MAJ dans les *BLOCS en attente* que contient sa mémoire propre. Une solution symétrique à celle de la solution 1 peut être envisagée (Figure 6).

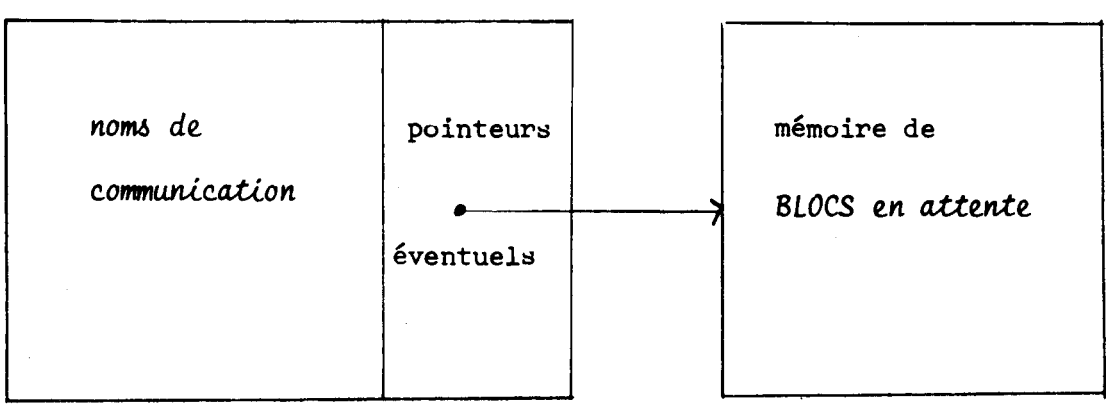


Figure 6. : Une organisation possible de la mémoire de MAJ (solution 2).

Par contre, la mise-à-jour de l'indicateur de l'état du BLOC ne pose pas de problème puisque le contrôle sur le *domaine d'entrée complet* peut se faire en différé, les *BLOCS en attente* se trouvant dans la mémoire propre du processeur MAJ ; de même, si ce dernier dispose d'un tampon (suffisamment grand), la lecture de la valeur V_i peut se faire dans le tampon et la recopie dans le BLOC concerné peut s'effectuer en différé aussi. Dans ce cas, le traitement à la volée semble très intéressant.

Toutefois, tous les couples d'un *domaine de sortie* ne sont pas nécessairement utilisés dès leur premier passage devant le processeur MAJ car il est possible que le BLOC auquel ils doivent être transmis ne soit pas encore *en attente*. De plus, chaque couple (NC_i, V_i) n'est utilisé qu'une fois (chapitre II). Dès qu'un couple est utilisé par le processeur MAJ il faut donc le rendre inacessible, par exemple en remplaçant le nom N_i par un nom *inutilisé* signifiant que ce nom n'est désormais plus à prendre en compte. Il est également possible de faire ce remplacement à la volée.

Cependant, si les processeurs de traitement déposent un seul *domaine de sortie* dans un *cadre*, il risque d'y avoir dans l'anneau beaucoup de *cadres* contenant des *domaines de sortie* : en effet, le processeur MAJ n'utilise pas toujours tous les couples d'un *domaine de sortie* en une seule fois. De plus, lorsqu'il a épuisé tous les couples, il devrait modifier l'indicateur du *cadre* en le remplaçant par "vide".

Par conséquent, dans cette solution 2 , il semble intéressant de regrouper plusieurs *domaines de sortie* dans un même *cadre*. Deux questions se posent :

i) qui place l'indicateur *domaine de sortie* dans un *cadre* ? et qui l'enlève ?

ii) qui gère la place occupée dans un *cadre* contenant un *domaine de sortie* ? Nous désignerons dans la suite ces *cadres* par *cadre* DS.

Nous allons envisager successivement plusieurs possibilités en supposant toujours que les *domaines de sortie* sont déposés dans l'anneau en une seule fois.

1) les processeurs de traitement

Lorsqu'un processeur de traitement a besoin d'un *cadre* DS, s'il n'y a plus assez d'espace libre dans ceux qui passent devant sa fenêtre, ou s'il n'en a pas trouvé au bout d'un certain temps T, il pourrait décider d'en créer un lui-même à partir d'un *cadre* vide. Puisqu'il n'existe pas de communication directe entre processeurs, à la limite tous les processeurs peuvent créer un *cadre* DS en même temps. Ce qui revient à un excès de *cadres* DS dans l'anneau.

D'autre part, les temps de recherche d'un *cadre* DS utilisable risquent d'augmenter considérablement ; en cas de création, il faut au moins le temps T, puis le temps de recherche d'un *cadre* vide.

De même, si les processeurs de traitement doivent gérer eux-mêmes l'espace disponible dans un *cadre* DS, même si les *domaines de sortie* sont tassés au début du *cadre* (par le processeur MAJ par exemple), ils finiront par passer beaucoup de temps en communication avec l'anneau au détriment du traitement proprement dit.

2) le processeur MAJ

Le processeur MAJ est le seul processeur retirant des éléments des *cadres* DS et il semblerait normal que la gestion complète (création, libération, tassement) de ces *cadres* lui soit confiée. En effet, puisqu'il crée des "trous" dans l'espace des *cadres* DS en retirant régulièrement les couples (NC_i, V_i), il pourrait effectuer le tassement, et même à la volée s'il possède deux fenêtres sur l'anneau.

De plus, lorsqu'il constate qu'il reste peu d'espace disponible dans un *cadre* DS, il peut créer un nouveau *cadre* DS à partir d'un *cadre* vide. De la même façon, lorsqu'il utilise l'unique couple (NC_i, V_i) d'un *cadre* DS, il peut remplacer l'indicateur de ce *cadre* par "vide".

Toutefois, le processeur MAJ est déjà très sollicité pour qu'on lui confie ce travail supplémentaire.

3) un processeur spécialisé

En réalité, le travail à effectuer est un contrôle du nombre de *cadres* dans l'anneau, et cela correspond au rôle d'un "GERANT" d'espace de sortie.

Le GERANT ne peut se rendre compte qu'il faut ajouter un *cadre* DS à l'anneau que s'il contrôle le contenu des *cadres* DS, et dans ce cas, en contrôlant le taux d'occupation des *cadres* DS il peut effectuer le tassement des couples (NC_i, V_i), de manière analogue à celle qui est décrite au § 4.1.2.

Ce GERANT semble représenter la meilleure solution en ce qui concerne la gestion des *cadres* DS. Il peut d'ailleurs avoir d'autres fonctions, et nous en reparlerons au § 11.

Toutefois, la gestion de la place disponible dans un *cadre* DS et la récupération des trous faits par MAJ peut être facilitée en ne plaçant dans un *cadre* DS que des objets de même type. Cette nouvelle distribution des *domaines de sortie* dans l'anneau entraîne des modifications au niveau des procédures d'échange entre les processeurs de traitement et l'anneau, et représente une variante de la solution 2.

7. SOLUTION 3 : CADRES DOMAINES DE SORTIE "TYPÉS"

Il est possible de ne placer dans un *cadre* DS que des objets de même type : il y a alors des *cadres* DS réservés aux entiers, d'autres aux réels, d'autres aux booléens, etc... (les *cadres* DS seraient alors mieux nommés *cadres objets de sortie*). D'un point de vue matériel, cette solution est très satisfaisante : les objets d'un *cadre* DS étant tous de même type, il est possible de leur fixer une taille : il devient alors très facile de consulter chaque couple d'un *cadre* DS, à intervalles réguliers, en synchronisme avec la longueur des objets. Les procédures d'échange avec les processeurs et l'anneau deviennent beaucoup plus simples au niveau matériel :

1) échanges entre processeur MAJ et anneau

Ils sont fonctionnellement identiques à ceux décrits au début de la solution 2 (§ 6). Seule la réalisation matérielle est différente et beaucoup plus simple. Le tassement des objets dans le *cadre* DS n'est pas neeéssaire : les processeurs de traitement pourront réutiliser les "trous" créés par MAJ.

2) échanges entre processeur de traitement et anneau

Les processeurs de traitement ne déposent plus un *domaine de sortie* en une seule fois, mais écrivent des *objets de sortie* (NC_i, V_i). Ils doivent :

> . pour chaque couple (NC_i, V_i)
> .. chercher un *cadre* DS possédant le même type que la valeur V_i
> .. lorsque ce *cadre* est trouvé, le balayer en cherchant la première place libre, c'est-à-dire un couple (NC_j, V_j) où NC_j est le nom *inutilisé*
> .. remplacer (NC_j, V_j) par (NC_i, V_i).

Il est évident que le temps de dépôt dans l'anneau d'un *domaine de sortie* va considérablement augmenter, car le travail précédent est à effectuer pour tous les couples du *domaine de sortie*. S'ils sont tous de même type, les temps d'attente seront probablement assez courts Sinon, il faut redouter une forte pénalisation par rapport au temps de traitement des processeurs. En effet, il faut d'abord supposer qu'au moment d'écrire le *domaine de sortie*, le processeur est encore capable d'accéder au type de l'objet à écrire (peut être serons nous obligés de l'ajouter à chaque objet). D'autre part, aucune précision n'a été donnée jusqu'ici sur l'ordre des objets dant le *domaine de sortie* : il n'avait aucune importance. Envisageons le *domaine de sortie* suivant:

(N_a, 1), (N_b, 3.14), (N_c, 2), (N_d, D), (N_e, 3)
entier reel entier caractère entier

* ou bien, les couples sont pris dans l'ordre où ils se trouvent dans le *domaine de sortie*, et il faut effectuer 5 recherches de *cadres*, alors que les 3 entiers peuvent peut-être être rangés dans le même *cadre* ;

* ou bien, le processeur effectue un tri sur les types avant de rechercher les *cadres*, ce qui ne garantit d'ailleurs pas qu'il en cherchera moins de cinq.

Dans tous les cas, le processeur de traitement va passer beaucoup de temps à mener des activités peu intéressantes par rapport à sa vraie fonction qui est l'exécution d'un BLOC. Ceci est un argument en faveur d'une unité d'échange évoluée qui effectuerait une partie des opérations qui sont fonctionnellement à la charge du processeur : décision sur les catégories de *cadres* à rechercher, tri des *objets de sortie* par type. Toutefois, l'unité d'échange peut posséder une petite mémoire tampon. En effet, lorsque le processeur a transmis le *domaine de sortie* à l'unité d'échange, il devient libre pour le traitement d'un *BLOC exécutable*, que doit lui fournir l'unité d'échange. Aussi cette dernière doit elle stocker le *domaine de sortie* dans une mémoire locale, pendant la recherche et la lecture d'un *BLOC exécutable*.

Particularités de la solution ?

- le temps de dépôt d'un *cadre* DS dans l'anneau peut être très important.

- il circule dans l'anneau des *cadres* DS "typés", ce qui peut imposer des contraintes quant au nombre de types utilisables pour les objets des programmes.

- il existe dans l'anneau un certain nombre de *cadres* à indicateur prédéfini. Pour certaines catégories de problèmes, le nombre de *cadres* DS d'un certain type, par exemple "entier", peut se révéler insuffisant. Ce besoin ne peut être détecté et satisfait que par un processeur spécialisé assurant la gestion de l'espace de sortie.

Comparaison des 3 solutions proposées

Il faut évaluer chacune des solutions vis-à-vis de quatre critères :

- complexité matérielle
- vitesse des échanges
- souplesse par rapport à des catégories diverses de programmes
- possibilités d'extensions

Vis-à-vis de ces quatre critères, les trois solutions présentent chacune des avantages et des inconvénients :

- la première solution apparait plus systématique, plus efficace quant à la rapidité du traitement puisque les processeurs de traitement sont complètement déchargés de la gestion des *cadres*. Mais la réalisation des procédures d'échange entre le processeur MAJ et l'anneau pose quelques problèmes matériels.

- dans les deux autres, l'anneau joue mieux son rôle de mémoire. Mais les procédures d'échange des processeurs de traitement avec l'anneau sont plus complexes et plus longues.

Le choix entre ces trois solutions dépendra des choix matériels qui seront faits pour la réalisation.

8. ECHANGES AVEC L'EXTERIEUR

Les échanges avec l'extérieur se font par l'intermédiaire de *demandes d'exécution* pour des BLOCS particuliers dont la partie interne correspond à des sous-programmes d'entrées-sorties. Ces BLOCS ont des noms qui peuvent être prédéfinis et qui sont reconnus par le processeur CONSTRUCTEUR. Ce dernier place alors un indicateur "échanges" en tête du BLOC construit. Ce BLOC sera intercepté par un processeur spécialisé dans les échanges avec l'extérieur.

Les *demandes d'exécution* pour ces BLOCS d'échange se font à l'aide des opérations execbloc, de manière tout-à-fait analogue à ce qui a été défini dans le chapitre III. Suivant les caractéristiques de l'échange demandé, le processeur spécialisé renverra un *domaine de sortie* ou non.

Les échanges avec l'extérieur ne posent donc pas de problème particulier, si ce n'est peut-être au niveau de la taille des informations échangées dans certains problèmes de gestion.

9. INTERRUPTIONS

Dans un monoprocesseur, les interruptions servent à prévenir la machine qu'un évènement extérieur s'est produit et qu'elle doit momentanément abandonner le travail en cours pour traiter cet évènement.

Dans un multiprocesseur, une interruption peut concerner toute la machine ou un processeur particulier. Dans un système où il existe des communications directes entre processeurs, un processeur P_i peut envoyer une interruption à un processeur P_j particulier.

Dans un système comme MAUD où les processeurs n'ont aucune communication directe entre eux et où les processeurs de traitement sont banalisés, une interruption ne peut concerner directement le traitement en cours dans la machine : on serait incapable de savoir sur quel processeur se déroule la tâche qui doit être interrompue. Par contre, une interruption pourrait concerner le processeur CONSTRUCTEUR, ou des processeurs spécialisés dans les échanges avec l'extérieur, par exemple pour l'introduction de nouveaux programmes dans la BIBLIOTHEQUE.

10. TRAITEMENT DES ERREURS

Lorsqu'une erreur est décelée au cours de l'exécution d'un BLOC, plusieurs traitements peuvent être envisagés :

1) faire automatiquement une *demande d'exécution* pour un BLOC particulier de traitement d'erreur, le même pour tous les utilisateurs de MAUD. Cette fonction serait alors une primitive à ajouter aux processeurs de traitement.

2) faire une *demande d'exécution* d'un BLOC particulier à l'utilisateur que ce dernier doit effectuer à l'aide d'une **opération** execbloc explicite écrite dans le BLOC. Dans ce cas, la suite du traitement peut être prévue **par l'utilisateur** lui-même, avec reprise ou non du BLOC où l'erreur a été décelée.

De toute façon, il faut s'assurer qu'il ne reste pas de BLOCS dont l'exécution a été demandée ou dépend du BLOC BO dans lequel il y a une erreur, et qui ne **seront** donc pas utilisés.

Dans le cas de BLOCS demandés par un execbloc, si leurs résultats n'ont pas été communiqués au BLOC erroné BO, les *noms de communication* des objets qu'ils doivent évaluer figurent encore dans la *liste de correspondance d'entrée* de ce BLOC BO.

Dans le cas de BLOCS dépendants du BLOC erroné, ce sont les *noms de communication* figurant dans le *domaine de sortie* de BO qui permettront de retrouver les BLOCS dont l'exécution ne pourra avoir lieu.

Toutes ces informations **doivent** être transmises à un processeur qui peut obtenir suffisamment de renseignements sur l'état du traitement en cours pour effectuer la récupération des BLOCS qui ne serviront pas. En effet, une erreur dans un BLOC donné peut entraîner l'annulation de toute une cascade de BLOCS. Ce travail consiste en un examen des *domaines d'entrée* et *domaines de sortie* et peut donc être confié au processeur MAJ. Mais il serait possible, et peut-être préférable, de le confier à un processeur ne s'occupant que des problèmes de gestion de MAUD.

11. REGULATION DE LA CHARGE DE L'ANNEAU

Contrairement aux mémoires classiques, il semble très difficile de connaître le taux d'occupation de l'outil de stockage qu'est l'anneau de mémoire circulante. Dans les solutions 2 et 3 ci-dessus, une partie de la gestion de l'anneau est supportée par les différents processeurs connectés.

Cependant il peut dans tous les cas y avoir des situations de blocage dues à un manque de *cadres* libres dans l'anneau. Il n'est pas raisonnable de confier la gestion du nombre de *cadres* vides à un des processeurs définis jusqu'ici : l'exécution des programmes dans MAUD doit être totalement indépendante de ce problème. Aussi semble-t-il préférable d'attribuer ce travail à un processeur spécialisé, le GERANT, dont la seule fonction est la gestion du nombre de *cadres* libres.

Nous n'étudierons le GERANT que dans l'optique de la solution 1 (*BLOCS en attente* restant dans l'anneau), où les processeurs sont déchargés d'une gestion quelconque des *cadres*, ce qui semble plus intéressant au point de vue vitesse d'exécution des programmes.

11.1. RÔLE DU GERANT

Le rôle du GERANT est de surveiller la charge de l'anneau et d'intervenir lorsqu'il y a encombrement, en dirigeant provisoirement sur des "voies de garage" un certain nombre d'objets en circulation dans l' anneau, et en les replaçant lorsque la charge aura diminué.

Une solution possible est la suivante : le GERANT contrôle la charge de l'anneau en comptant le nombre de *cadres* vides au cours d'un tour de l'anneau. Si ce nombre est compris entre PETIT NOMBRE et GRAND NOMBRE, il n'intervient pas. Si ce nombre devient inférieur à PETIT NOMBRE, il doit décharger le contenu d'un certain nombre de *cadres* occupés dans une mémoire qui lui est propre afin de placer sur ces *cadres* un indicateur "vide". Si le nombre de *cadres* vides devient supérieur à GRAND NOMBRE, le GERANT replace dans des *cadres* vides les objets qu'il a stockés momentanément, s'il en possède.

11.2. CHOIX DES OBJETS À ÉCARTER DE L'ANNEAU

Les objets qui peuvent rester dans l'anneau sont ceux qui n'y sont que de façon transitoire, c'est-à-dire les *demandes d'exécution* et les *domaines de sortie* qui sont normalement interceptés dès leurs premier passage dans la fenêtre respectivement du CONSTRUCTEUR et du processeur MMJ.

Pour les *BLOCS exécutables*, il est inutile d'en laisser trop dans l'anneau s'il n'y a pas de processeur de traitement libre pour les exécuter. Pour ceux laissés dans l'anneau, il faut être sûr qu'ils seront interceptés rapidement par un processeur de traitement, donc savoir s'il existe des processeurs libres. Ce qui implique que le GERANT doit connaître à tout moment le nombre de processeurs occupés.

Quant aux *BLOCS en attente*, ce sont les seuls objets actifs puisque ce sont les seuls qui peuvent être lus et modifiés plusieurs fois au cours de leur vie dans l'anneau. Cependant leur durée de vie dans l'anneau peut être très importante et on sera parfois obligé de les écarter temporairement. Dans ce cas, il faudra faire un choix non pénalisant pour la poursuite du traitement, c'est-à-dire ne pas sortir de l'anneau des *BLOCS en attente* d'objets déjà apparus dans des *domaines de sortie*.

Au moment où il pourra remettre des BLOCS dans l'anneau, le GERANT choisira des *BLOCS exécutables* s'il en possède. Sinon il pourra utiliser pour les *BLOCS en attente* un critère symétrique de celui qui a été choisi pour les écarter de l'anneau.

BIBLIOGRAPHIE

[*AG* 77] ARVIND et GOSTELOW, K.P.
"a computer capable of exchanging processors for time"
Proceedings of IFIP Congress 1977.

[*AG* 78] ARVIND et GOSTELOW, K.P.
"data flow computer architecture : research and goals"
TR 113. University Of California-Irvine. Fév. 78.

[*AGP* 77] ARVIND, GOSTELOW, K.P. et PLOUFFE, W.
"indeterminacy, monitors and dataflow"
Proceedings of sixth ACM symposium on Operating systems principles. Nov. 77.

[*AGP* 77] ARVIND, GOSTELOW, K.P. et PLOUFFE, W.
"the ID preliminary report : an asynchronous programming language and computing machine"
TR 118 - University of California-Irvine. Sept. 78.

[*B*er 66] BERNSTEIN, A.J.
"analysis of programs for parallel processing"
IEEE Trans. EC 15, n° 5, Oct. 66.

[*B*ou 72] BOUKNIGHT, W.J. et al.
"the ILLIAC IV system"
Proceedings of IEEE. Avril 72.

[*B*ri 73] BRINCH HANSEN, P.
"operating systems principles"
Prenctice-Hall, Englewood cliffs, new Jersey 1973.

[*C*ha 71] CHAMBERLIN, D.D.
"parallel implementation of a single assignment language"
Ph-D-Thesis. Stanford University. Jan. 71.

[Con 63] CONWAY, M.E.
"a multiprocessor system design"
Proceedings FJCC 1963.

[Cor 78 a] CORDONNIER, V.
"architectures pipeline"
Support du cours IRIA sur le traitement parallèle. Lille, 29 Mai, 2 Juin 1978.

[Cor 78 b] CORDONNIER, V.
"l'emploi des mémoires circulantes dans l'architecture des ordinateurs"
Publication du Laboratoire de Calcul de l'Université de Lille, 1978.

[Cor 79] CORDONNIER, V.
"la mémoire circulante de MAUD"
Publication du Laboratoire de Recherches en Architecture des systèmes et Machines Informatiques. Université de Lille, n° 2, 1979.

[CS 76] COMTE, D., SYRE, J.C. et al
"the LAU parallel system : software definition and implementation through a multimicroprocessor architecture"
Congrés Euromicro-Venise, 1976.

[Dav 78] DAVIS, A.L.
"Data driven nets : a maximally concurrent, procedural, parallel process representation for distributed control systems"
Department of Computer Science, University of Utah. TR 108. Jul. 1978.

[Den 74] DENNIS, J.B.
"First version of a data flow programming language"
Proceedings of symposium on programming languages. Paris April 1974.

[*Dij* 68] DIJSKTRA, E.W.
"cooperating sequential processes"
Programming Languages. F. Genuys.ed. 1978.

[*DM* 74] DENNIS, J.B. et MISUNAS, D.P.
"a preliminary architecture for a basic data flow processor"
Proceedings ACM SIGARCH, IEEE Symposium on Computer Architecture. Dec. 1974.

[*DML* 77] DENNIS, J.B., MISUNAS, D.P. et LEUNG, C.K.
"a highly parallel processor using a data flow machine language"
Computation structures Group Memo 134, MIT. Jan 77.

[*Dur* 77] DURRIEU, G.
"étude et définition d'une machine parallèle asynchrone à assignation unique"
Thèse Université Paul -Sabatier-Toulouse. 1977.

[*GW* 75] GELLER, D.P. et WEINBERG, G.M.
"The principle of sufficient reason : a guide to language design for parallel processing"
Sig plan. Notices. n° 10.

[*Lec* 78] LECOUFFE, M.P.
"MAUD : une machine d'assignation unique dynamique"
Publication du Laboratoire de Calcul de l'Université de Lille. N° 116. Sept. 78.

[*Lec* 79] LECOUFFE, M.P.
"MAUD : a dynamic single-assignment system"
Computers and digital techniques. April 79. Vol. 2, n° 2.

[*Leg* 78] LEGUY, B. et LEGUY, J.
"mémoire active limitée aux besoins de l'utilisateur"
Publication du Laboratoire de Calcul de l'Université de Lille, n° 114, Août 1978.

[*Pan* 77] PANIGRAHI, G.
"the implications of electronic serial memories"
Computer. July 1977.

[*Pla* 77] PLAS, A.
"étude et définition d'un langage machine à assignation unique"
Thèse Université Paul Sabatier-Toulouse 1977.

[*RG* 70] RAMAMOORTHY, C.V. et GONZALEZ, M.J.
"recognition and representation of parallel processable streams in computer programs"
parallel processor systems, Technologies and applications
L.C. Hobbs, ed. 1970.

[*RV* 77] ROBERT, P. et VERJUS, J.P.
"towards autonomous descriptions of synchronization modules"
Proceedings IFIP 77.

[*RW* 73] ROUCAIROL, G., et WIDORY, A.
"programmes séquentiels et parallélisme"
RAIRO, B.2., Juin 1973.

[*Syr* 76] SYRE, J.C. et al
"techniques et exploitation de l'assignation unique"
Contrat Sesori 74-167, Vol 9 (tomes 1 à 7). Rapport final.
Oct. 1976.

[*Syr* 78] SYRE, J.C. et al
"parallelism, control and synchronization expression in a single assignment language"
Sig plan Notices n° 13, Jan 78.

[*TE* 68] TESLER, L.G. et ENEA, H.J.
"a language design for concurrent processes"
Proceedings SJCC 1968.

[*T*re 78 a] TRELEAVEN, P.C. et al
"the design of highly concurrent computing systems"
University of Newcastle upon Tyne. TR. July 1978.

[*T*re 78 b] TRELEAVEN, P.C.
"principal components of a data flow computer"
Congrés Euromicro-Munich. Oct. 1978.

[*T*re 79] TRELEAVEN, P.C.
"exploiting program concurrency in computing systems"
Computer. Jan. 1979.

[*U*rs 73] URSCHLER, G.
"the transformation of flow diagrams into maximally parallel forms"
Conference on parallel processing. Sagamore 1973.

[*V*an 79] VANLAER, P.
"étude d'une architecture à flux simultanés"
Université de Lille I, Thèse de docteur-Ingénieur, Avril 1979.

[*WDD* 75] WORKSHOP on Data Driven Languages and Machines-Toulouse
compte-rendu des journées d'étude des 12 et 13 février 1979.

[*W*en 75] WENG, K.S.
"stream-oriented computation in recursive data flow schemas"
Laboratory for Computer-Science. MIT. Oct. 1975.

[*Y*au 77] YAU, S.S. et FUNG, M.S.
"associative processor architecture a survey"
Computing Surveys Vol. 9. n° 1. Mars 1977.