50376
1986
93

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNIQUES DE LILLE

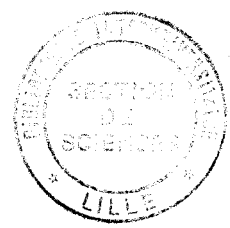
pour l'obtention du titre de

Docteur - Ingénieur
en
Automatique
par

L. Teixeira de Carvalho

Soutenue le 24 Janvier 1986

ÉTUDE D'UNE STATION LOCALE MULTIPROCESSEURS



AVANT - PROPOS.

Nous sommes très heureux de pouvoir exprimer ici notre gratitude à Monsieur J.M. TOULOTTE, qui nous a suivi et guidé durant cette étude.

Nous remercions Messieurs les Membres du Jury, d'avoir bien voulu siéger à cette Commission d'Examen.

T A B L E D E S M A T I E R E S

I N T R O D U C T I O N G E N E R A L E

Dans le cadre du travail que nous présentons dans ce mémoire nous avons été amené à étudier les systèmes multiprocesseurs dont le développement, lié à celui des composants électroniques intégrés, est très certainement un sujet d'actualité.

Beaucoup de stations multiprocesseurs ont été décrites dans la littérature, mais peu sont celles qui sont complètement exploitées, en particulier si l'on n'a pas considéré, dés le début, le logiciel correspondant à leur architecture. (19).

Le parallélisme est en effet le point clé de l'utilisation de plusieurs processeurs;La première partie de notre présentation porte sur l'étude du parallélisme dans les automatismes logiques.

Dans une deuxième partie, on fait un tour d'horizon des stratégies d'ordonnancement d'un ensemble de tâches dans le cas de plusieurs processeurs. Une stratégie particulière est proposée et mise en oeuvre.

Dans une troisième partie on fait le choix d'un outil de description d'une application et un langage permettant son édition est proposé. L'implantation de la description est alors étudiée ainsi que la technique permettant d'élaborer à chaque instant la liste des tâches à ordonnancer.

Enfin dans une quatrième et dernière partie on présente quelques stations multiprocesseurs existantes. On en propose une et l'on donne les résultats des essais effectués.

Afin de pouvoir controler et eventuellement revoir nos idées, nous avons du chercher les moyens de réaliser une station d'essai hors site. La société ABSY de Bruxelles a accueilli favorablement notre demande d'aide et nous a fourni le matériel nécessaire.

C H A P I T R E I

E T U D E D U P A R A L L E L I S M E D A N S L E S A U T O M A T I S M E S L O G I Q U E S

I. 1. INTRODUCTION.

Résoudre un problème d'automatisation de processus à évolution séquentielle revient à réaliser un automatisme logique pouvant être représenté comme l'interconnexion de deux parties qui coopérent:

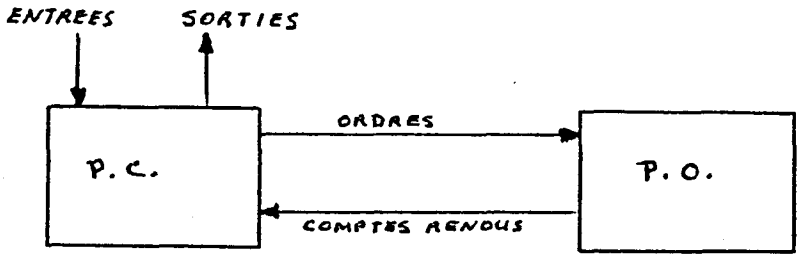


Fig. I.1

La partie opérative (P.O.) représente le procédé à commander tandis que la partie commande (P.C.) représente l'automatisme à concevoir (5) (32) (33).

La partie commande a été matérialisée jusqu'à ces dernières années par des architectures essentiellement séquentielles. en fait cette organisation est issue de l'introduction des composants électroniques (16). La supériorité de l'électronique sur le plan de la vitesse d'exécution par rapport aux composants mécaniques et électromécaniques a permis un accroissement des performances par un facteur compris entre 10^3 et 10^4 . Le traitement séquentiel qui en résulte permet une économie appréciable de matériel. Cependant les progrès récents de la technologie électronique ont amené des applications dont la taille et la complexité vont en grandissant. La conséquence en est un retour aux machines parallèles.

I. 2. LE PARALLELISME AU NIVEAU FONCTIONNEL.

I. 2. 1. Parallélisme géographique et parallélisme local.

Un problème d'automatisme se résoud par l'envoi d'un ensemble d'ordres au procédé moyennant la vérification d'un ensemble de conditions. L'élaboration et l'application des ordres ainsi que la vérification des conditions essentiellement liées aux comptes-rendus, constituent l'information traitante. Les comptes-rendus eux-mêmes sont associés tout naturellement à l'information traitée.

Les sources de l'information traitée se localisent aux différentes parties du procédé qui, aujourd'hui, est multi-machines, multi-ateliers et donc réparti sur des distances non négligeables. Normalement sur cette répartition, qui apparait au niveau des spécications, on calque une décomposition de l'information traitante et traitée.

La partie commande éclate donc en sous-parties, interconnectées entre-elles mais relativement indépendantes, ce qui correspond à un parallélisme naturel géographique.

Au niveau de chacune des "sous-parties" de commande on peut ajouter un parallélisme local.

En effet localement les spécifications déterminent un ensemble de tâches et les relations qui les lient. Ces relations correspondent aux contraintes de précédence . Ainsi, par exemple, une tâche T_2 ne peut démarrer qu'après l'accomplissement de la tâche T_1 et doit démarrer en même temps qu'une tâche T_3 . Ces contraintes mettent en évidence un ou plusieurs sous ensembles de tâches pouvant être exécutées simultanément ou non suivant le choix du concepteur.

On donne potentiellement par les contraintes de précédences un certain degré de parallélisme. Mais deux tâches parallèles peuvent être, au niveau de la commande, séquentialisées. Il suffit que la partie commande soit très rapide par rapport à la vitesse d'évolution du procédé. Plusieurs stuctures sont alors possibles, on peut soit tout placer en séquence, soit mettre au maximum en parallèle.

Le choix se fait soit au niveau de la description pour faciliter l'analyse en étudiant une somme de machines, plutôt que leur produit, soit au niveau de l'implantation pour des problèmes de vitesse d'exécution ou pour améliorer la sécurité.

I. 2. 2. Parallélisme au niveau de l'algorithme.

Une tâche est un programme correspondant à l'implantation d'un algorithme pouvant posséder un certain parallélisme intrinsèque.

Un algorithme consiste en un ensemble d'opérations entre lesquelles existent des règles de précédence comme pour les tâches. Suivant ces règles, pour un algorithme donné, il y a un maximum d'opérations qui peuvent être exécutées en parallèle. Le concepteur aura ici aussi le choix de tout séquentialiser ou de tirer parti de la parallélisation possible. Les critères qui le guident sont ici surtout le gain de temps. Certains algorithmes sont sur le plan de la possibilité de parallèlisation nettement meilleurs que d'autres.

Afin d'illustrer ces notions nous allons reprendre les exemples donnés en (3).

1) Soit à multipier deux vecteurs B et C pour obtenir le vecteur A :

```
      DO  100  I = 1,N

100   A(I) = B(I) * C(I)
```

Il est évident qu'au lieu d'exécuter la boucle séquentiellement pour chaque valeur de I, les N opérations peuvent être faites en une seule fois en parallèle. Ce cas représente le type même des boucles parallélisées.

D'autres boucles récurentes comme dans le cas suivant peuvent être traitées, partiellement du moins en mode parallèle:

2) Soit à additionner les éléments d'un vecteur A pour obtenir la somme scalaire S :

```
      S = 0
      DO  200  I = 1,N
200   S = S + A(I)
```

Cette boucle peut être "parallélisée" en additionnant d'abord les paires adjacentes, puis les paires de sommes partielles....

1° pas : $T(1) = A(1) + A(2)$ $\quad T(3) = A(3) + A(4)$
$\quad\quad\quad\quad T(5) = A(5) + A(6)$ $\quad T(7) = A(7) + A(8)$

2° pas : $T'(1) = T(1) + T(3)$ $\quad T'(5) = T(5) + T(7)$

3° pas : $S = T'(1) + T'(5)$

I. 3. LE PARALLELISME AU NIVEAU MATERIEL

I. 3. 1. Le parallélisme de type réparti.

Nous avons vu au niveau fonctionnel qu'il y avait un parallélisme naturel géographique. Pour s'adapter le mieux possible à la partie opérative il y a lieu d'envisager une structure de machine de type reparti: ce sont les réseaux.

En pratique un réseau correspond à un ensemble de sous-machines reliées entre elles par des liens sériels.

Il correspond aussi à un système multiprocesseurs à couplage lâche, c'est à dire sans mémoire commune, où l'information d'une sous-machine n'est pas directement accessible à une autre via une simple unité de gestion mémoire.

On parle de:

Réseau local (Local Area Network, LAN) s'il est confiné à un batiment ou une usine et n'emprunte pas de systèmes publics de communications (exemple: l'Ethernet)

Réseau ouvert (Open System Interconnexion, OSI) quand il emprunte des voies de communications publiques (téléphone, télex, satellites,....).

Les distances mises en jeu entre sous-machines obligent à utiliser une transmission série qui oblige à mettre au point des protocoles d'échange de données pour éviter les aléas de fonctionnement du à des prises en comptes tardives d'événements (30).

Dès que le temps de transfert des données n'est plus négligeable, il faut l'ajouter au temps d'exécution des traitements. On augmente le temps de réponse du système, il en résulte une difficulté plus grande du respect des contraintes imposées par le fonctionnement en temps réel. Un autre problème délicat est celui de la datation des événements transmis au réseau.

Néanmoins dans le cas d'un système réparti, toutes les informations ne doivent pas être transmises sur longues distances, la plupart étant traitées localement, la situation se présente sous un jour plus favorable que dans celui des systèmes non répartis.

Un système réparti minimalise la quantité d'informations à transmettre, augmente la sécurité et la vitesse de travail, mais pose des problèmes de synchronisation.

La décentralisation favorise un "cloisonnement" des pannes et la redondance de processeurs peut augmenter la tolérance aux fautes.

I. 3. 2. Le parallélisme de type local.

Le parallélisme dans les machines locales tends à répondre au parallélisme fonctionnel local. Il apporte un gain

certain du point de vue vitesse de traitement et est aussi un facteur de sécurité supplémentaire en augmentant la tolérance aux pannes de par la redondance des processeurs.

La littérature présente plusieurs classifications des machines locales, la plus connue est celle de FLYNN (16) qui donne les quatre types suivants de machines (figure I.2)

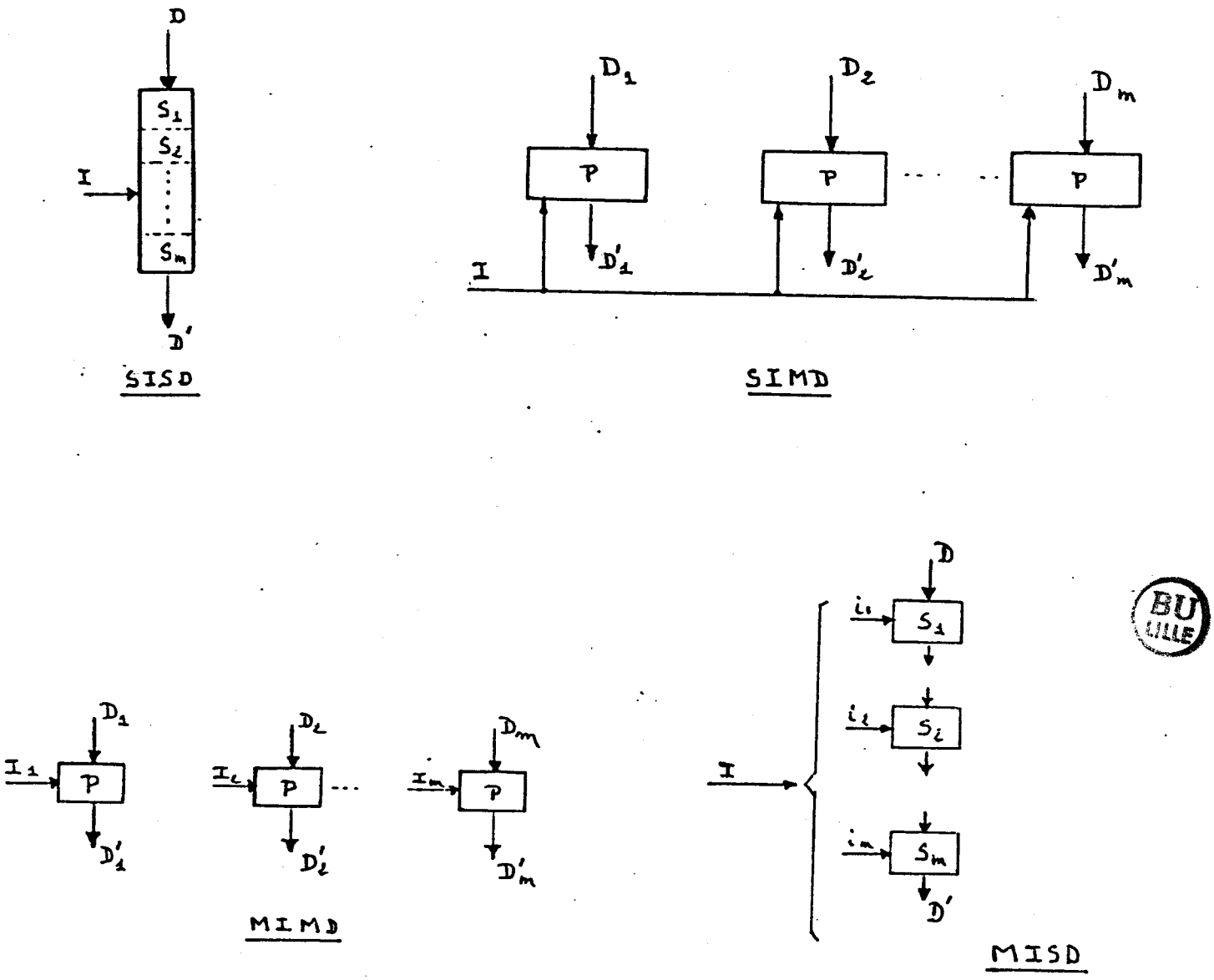


Fig. I.2

1) SISD : machines Simple Instruction, Simple flot de Données.

2) SIMD : machines Simple Instruction, flots Multiples de Données.

3) MIMD : machines Multiples Instructions, flots Multiples de Données.

4) MISD : machines Multiples Instructions, Simple flot de Données.

Les machines SISD correspondent aux machines séquentielles classiques.

Les machines MIMD incluent les systèmes multiprocesseurs à couplage lâches, donc les machines reparties en réseaux.

D'autres terminologies existent également.

Les machines vecteurs (array processors) sont des machines à unité de traitement multiples mais identiques et où chacune exécute en parallèle la même instruction, mais sur des données différentes. C'est en fait un cas particulier des machines SIMD.

Les systèmes multiprocesseurs à couplage étroit correspondent au cas ou plusieurs processeurs exécutent des instructions différentes sur des données également différentes, tout en participant à un travail commun. Chaque processeur travaille à un programme distinct et à accès à la mémoire commune du calculateur, qui lui sert de boite aux lettres pour correspondre avec les autres processeurs. Les processeurs peuvent être identiques ou spécialisés, s'ils sont complétement interchangeables on parle d'une famille (POOL) de processeurs. Cette classe de machines entre dans l'ensemble des machines MIMD.

Dans les machines pipelines, cas particuliers de machines MISD, les processeurs sont mis en cascade et effectuent chacun une phase d'une tâche donnée. Les résultats fournis par un processeur servent de données au suivant.

I. 3. 3. Remarques.

Nous avons utilisé les termes "unité de traitement", "processeur " et "calculateur" il est bon de les définir.

Ainsi, une carte avec un microprocesseur comme le 8085 sans mémoire locale peut être considérée comme une "unité de traitement".

Un "processeur" se compose d'une ou plusieurs unités de traitement, d'une mémoire locale et d'organes d'entrée - sortie.

Un "calculateur" (computer) est composé d'un certain nombre de processeurs et d'une mémoire centrale commune. S'il est de grosse taille, on parle d'ordinateur. Nous parlerons aussi de station multiprocesseurs.

Nous avons également dit, au début du paragraphe sur les machines locales, que le parallélisme assure un gain certain de vitesse de traitement, ce qui revient à dire qu'il améliore le temps de réponse de l'automatisme. Or ce dernier est la somme de deux composantes: le temps d'exécution des traitements d'une part et le temps de transfert des données d'autre part.

Le temps de transfert des données est lié au mode de transmission utilisé mais aussi et surtout à leur localisation relative et à leur volume. Le temps d'exécution lui est lié à l'information

traitante et à la structure de la machine.

Dans un premier temps, nous allons définir et évaluer les performances de ces structures sans tenir compte des temps d'acces aux données et nous nous servirons pour cela d'un exemple que l'on retrouve fréquemment dans la littérature (16)

I. 3. 4. Mesure des performances.

Une mesure de la performance d'une structure peut être définie par le nombre maximum de résultats obtenu par unité de temps. On l'appelle la largeur de Bande de Données b_D (Data Band-Width).

Considérons l'exemple de l'addition de deux vecteurs en virgule flottante:

$$Xi + Yi = Zi \quad (i = 1, 2, \ldots, n)$$

L'opération d'addition de chacune des paires d'éléments ci-dessus ($X = e.2^p$ et $Y = f.2^q$) peut être divisée en quatre sous-opérations (figure I.3) qui, nous le supposons pour simplifier, demandent un temps de traitement identique.

Si chaque sous opération demande un seul cycle horloge, alors:

- dans le cas d'une machine SISD nous avons naturellement un résultat pour quatre cycles horloge, soit
  $b_D = 1$
- dans le cas d'une machine fonctionnant en mode pipeline nous avons un résultat par cycle horloge, soit
  $b_D = 1$.

Mais remarquons que le premier résultat n'est obtenu qu'après le quatrième cycle horloge et qu'il faut encore quatre cycles horloge après la dernière prise de données.

- dans le cas d'une machine vecteur, s'il y a N processeurs, nous avons N résultats pour quatre cycles horloge, soit $b_D = 0,25.N$

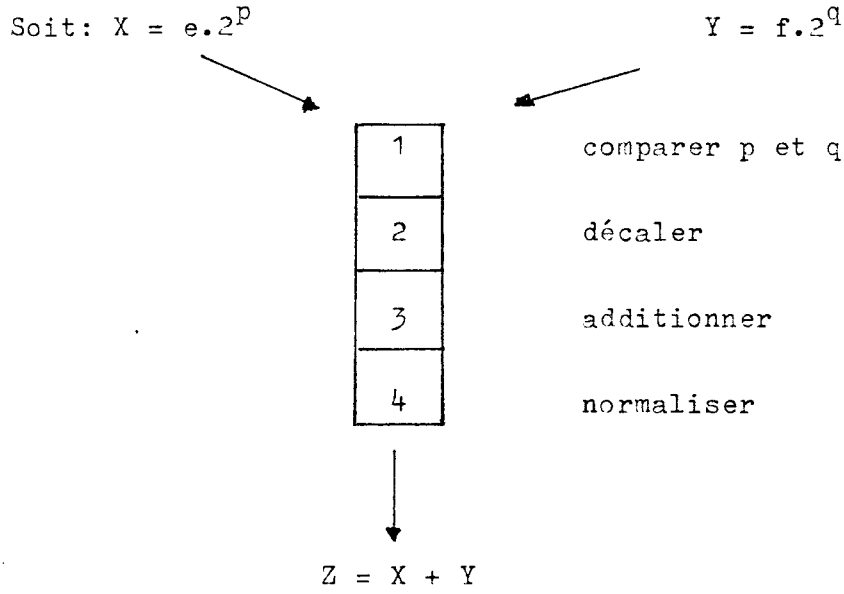


Fig. I. 3

Il semble donc que par rapport à la machine SISD, les performances de la machine pipeline sont multipliées par le nombre de segments et celles de la machine vecteurs par le nombre de processeurs, ce qui en première approximation d'ailleurs revient au même.

Toutefois, si nous introduisons les temps d'acces aux données, la situation devient nettement moins optimiste.

Ainsi, si nous supposons un système multiprocesseurs ou les programmes sont en mémoire locale et les données en mémoire commune et si nous appelons:

- α : la portion de temps passée par chaque processeur à lire ou écrire des données en mémoire par rapport au temps total d'exécution.

- N : le nombre de processeurs concurrents pour l'accès à la mémoire commune.

Alors, il faut que $\alpha.N \leqslant 1$ pour qu'il n'y ait pas de saturation de manière certaine, c'est à dire, pour qu'il n'y ait pas de processeurs en attente d'acces à la mémoire commune.

Ainsi, soit $\alpha = 0,1$ (c'est à dire quatre-vingt dix pour-cent du temps passé à des opérations locales et dix pour-cent du temps pour la lecture des données et l'écriture des résultats) le système ne peut assumer plus de dix processeurs sans diminution d'efficacité. De plus, ce résultat est optimiste car il faut que les demandes d'acces à la mémoire de chaque processeur se repartissent de manière optimale. En réalité, il faut tenir compte du fait que les demandes surviennent de manière quasi aléatoire. Il y a donc lieu d'envisager des files d'attente et des systèmes d'arbitrage ce qui réduit encore l'efficacité.

En définitive, la fonction qui représente l'augmentation des performances avec le nombre des processeurs n'est pas une droite comme on aurait pu l'espérer au départ. C'est une courbe qui tend vers une assymptote horizontale et qui peut même, dans certains cas, s'infléchir à nouveau vers le bas. Avec une dizaine de processeurs, on peut déjà arriver à saturation (16) (30).

I. 4. CLASSIFICATION DU PARALLELISME.

Nous pouvons résumer ce qui précède et en déduire la classification suivante:

1) Au niveau fonctionnel

   a) parallélisme géographique naturel

   b) parallélisme local:
      1) au niveau des relations entre tâches
      2) au niveau de l'algorithme choisi comme solution de chaque tâche.

A chacun des niveaux précédents on distingue le parallélisme correspondant à l'information traitante et celui correspondant à l'information traitée (les données).

2) Au niveau matériel

Les structures des machines capables d'exploiter le parallélisme fonctionnel se ramène fort bien à celles données par FLYNN, c'est à dire les machines: SISD, SIMD, MISD et MIMD. Cette classification peut être néanmoins quelque peu détaillée; Ainsi on a:

- Les machines MIMD se répartissant en
  - multiprocesseurs à couplage lâche ou réseaux répondant au parallélisme géographique
  - multiprocesseurs à couplage étroit répondant au parallélisme local tant sur le plan de l'information traintante que traitée.

- Les machines MISD correspondent essentiellement aux machines de structure également appelée "pipeline" et répondant aux problèmes ou le parallélisme se situe au niveau de l'information traitante

- Les machines SIMD correspondent essentiellement aux machines de structure également appelée "vecteur" et répondant aux problèmes ou le parallélisme se situe au niveau de l'information traitée.

- Les machines SISD correspondent aux machines sans parallélisme.

Rappelons encore que si le temps de réponse de l'automatisme est satisfaisant avec une machine SISD rien n'oblige le concepteur à exploiter le parallélisme possible si ce n'est pour répondre à un besoin de sécurité.

I. 5. CONCLUSIONS.

Nous venons de voir que la forme la plus générale d'un automatisme est le réseau, ensemble organisé de stations interagissantes. Ces dernières pouvant être elles mêmes constituées de multiprocesseurs à couplage étroit.

Il existe à l'heure actuelle un grand nombre de réseaux fonctionnant parfaitement. Mais les stations les constituant sont généralement monoprocesseur. Il n'y a encore que très peu de stations locales multiprocesseurs, c'est ce qui nous incline à poursuivre notre étude sur ce sujet.

C H A P I T R E II

S T R A T E G I E D' O R D O N N A N C E M E N T D' U N E N S E M B L E D E T A C H E S P A R U N G R O U P E D E P R O C E S S E U R S.

II. 1. INTRODUCTION.

Nous nous attachons dorénavant à l'élaboration d'une station locale multiprocesseurs, connectable à un réseau.

Une telle station comprend une "structure d'exécution locale" et un logiciel de gestion de tâches et d'exécution.

Les "intercommunications" se répartissent aurant sur la structure matérielle que sur le logiciel. Dans ce chapitre nous laissons de coté ce problème pour nous occuper de celui de la répartition d'un ensemble de tâches entre processeurs. Nous supposons pour cela la structure matérielle de la station constituée d'un groupe de processeurs banalisés et d'une mémoire commune. Pour cette structure nous allons nous efforcer de développer un Exécutif (operating system).

En pratique (19) l'exécutif est un programme résident dont les procédures peuvent être invoquées par les tâches utilisateurs. L'ensemble des conventions d'appel de ces procédures constitue ce qu'on appelle l' "interface utilisateur".

II. 2. NOTION DE TACHE.

Une tâche correspond à l'action découlant de l'exécution d'un programme sur une machine donnée. Le temps d'exécution de ce programme doit pouvoir être évaluable, ce qui suppose une complexité suffisamment réduite.

Une tâche a besoin du programme qui lui correspond, d'une zone mémoire pour ce dernier et de son point d'entrée. Elle a besoin de données et de l'endroit où les trouver, elle fournit des résultats et doit savoir où les placer.

Une tâche est existante s'il y a un descripteur qui donne pour elle les informations précédentes et s'il existe le programme correspondant. Sinon elle est inexistante.

Une tâche est dite active, c'est à dire que le programme qui lui correspond doit être exécuté, dans le cas contraire elle est dite non active ou désactivée.

Une tâche active peut être:

- Bloquée: en attente d'une donnée
- Prete : en attente d'être allouée à un processeur.
- En cours : en exécution par un processeur.

Dans le cas ou l'on considère un ensemble de tâches elles peuvent être indépendantes ou dépendantes. Elles ont besoin de communiquer entre elles et aussi, dans le cas des tâches dépendantes de se synchroniser.

La communication (29) entre deux tâches consiste en un échange d'informations suivant un protocole. Celui-ci fixe la forme que doit avoir cet échange pour que les deux tâches concernées se "comprennent" correctement. La communication entre deux tâches se réalise d'autant plus simplement que ces tâches sont voisines: l'utilisation de la mémoire commune permet de réduire le protocole de communication à des opérations de synchronisation et des manipulations de données partagées contenues dans la mémoire commune.

La synchronisation (29) des actions entreprises par une famille de tâches a pour rôle d'organiser l'exécution de ces actions. Un mécanisme de synchronisation est un moyen par lequel une tâche peut influer sur l'ordre des actions entreprises par les tâches avec lesquelles elle interagit. Parmis ces mécanismes nous notons les sémaphores, les queues, le monitor de Hoares et le "rendez-vous" du langage Ada.

Un sémaphore est une partie de mémoire qui se compose d'un emplacement pouvant contenir un JETON et éventuellement d'une liste de tâches en attente du JETON.

Le monitor de Hoares se comporte un peu comme un sémaphore. Une tâche propriétaire du monitor peut se mettre en attente d'une condition associée à ce monitor. Cette attente a pour effet de libérer le monitor pour les tâches qui veulent en prendre possession. Chacune des conditions spécifiques associées a un monitor est évaluée par les tâches qui en deviennent tour à tour propriétaires.

Lorsqu'une condition est devenue vraie, la tâche qui l'évalue le signale à la première tâche en attente de cette condition. Celle ci peut alors reprendre son exécution.

Le "rendez-vous" du langage Ada fournit un mécanisme permettant à une tâche "client" de faire exécuter un service par une tâche "serveur". Le service a la forme d'une procédure qui en Ada prend le nom " d'entry". Le client doit attendre que le serveur soit en mesure de le servir pour exécuter cette procédure. Lorsque c'est le cas, le client exécute la procédure et reprend son cours normal après cette exécution. Lorsque cette procédure a été exécutée le serveur reprend son activité.

Un ensemble de tâches demande un ordonnanceur chargé de gérer leur exécution par une famille de processeurs.

Cette gestion consiste soit à attribuer les processeurs (libres ou libérés) aux tâches prêtes à commencer ou à continuer leur exécution, soit à reprendre le processeur d'une tâche à la demande explicite de celle-ci (exemple: attente d'une condition) ou impérativement.

Cette gestion se fait suivant une certaine stratégie dont nous allons reparler dans la suite de ce chapitre.

Auparavant nous voudrions encore introduire la notion d'échéancier.

L'ordonnanceur agit pour le compte d'un ensemble de tâches activées. L'échéancier décide des tâches devant être activées ou désactivées. En effet lors du déroulement d'une application toutes les tâches lui correspondant ne sont pas active simultanément. C'est l'apparition d'un événement et la situation dans laquelle on

se trouve qui permet de décider des tâches à activer et à désactiver ce qui est fait par l'intermédiaire de l'échéancier. En réalité en terme de Grafcet, par exemple, l'échéancier décide s'il y a lieu de franchir une ou plusieurs transitions. Nous en reparlerons dans le chapitre suivant.

Classiquement l'échéancier active les tâches et les désactive en fonction de règles de priorité portant soit sur les temps d'exécutions soit sur la périodicité de leur répétitivité par exemple.

Pour qu'une tâche existe il faut la céer nous devons donc prévoir le mécanisme correspondant.

II. 3. STRATEGIE D'ORDONNANCEMENT D'UN ENSEMBLE DE TACHES.

II. 3. 1. Modèle général.

Les problèmes d'ordonnancement peuvent être traités en utilisant un modèle ou l'on considère les ensembles suivants (8) :

- Les Ressources
- Les Structures de tâches
- Les contraintes de précédence
- Les critères de performances.

II. 3. 1. 1. Les Ressources.

Nous considérons deux types distincts de ressources, d'une part l'ensemble $P = (P_1, \ldots, P_m)$ de processeurs et d'autre part l'ensemble $R = (R_1, \ldots, R_s)$ de ressources supplémentaires.

Suivant le problème traité les processeurs sont identiques sur le plan des possibilités fonctionnelles mais différents du point de vue vitesse de travail ou différents sur les deux plans.

Durant l'exécution d'une tâche par un processeur un sous ensemble de R, éventuellement vide, est requis et la disponibilité totale d'une ressource R_i donnée est indiquée par le nombre positif m_i . De telles ressources sont constituées en pratique par les mémoires internes, les mémoires de masse ou les ports d'entrée/sortie, etc.

II. 3. 1. 2. Les Structures de tâches.

Une structure générale de tâches pour un ensemble donné de ressources peut être défini par $S (\mathcal{T}, \prec, [\tau_{ij}], \{R_j\}, \{W_j\})$ où:

a) $\mathcal{T} = \{T_1, \ldots, T_n\}$ est un ensemble de tâches devant être exécutées.

b) $\prec$ est un ordre partiel défini sur $\mathcal{T}$ qui spécifie les contraintes de précédence des tâches. Ainsi $T_i \prec T_j$ signifie que T_i doit être exécuté avant de commencer T_j.

c) $[\tau_{ij}]$ est une matrice m X n de temps d'exécution, où $\tau_{ij} > 0$ est le temps requis pour exécuter T_j, $1 \leqslant j \leqslant n$, sur le processeur P_i, $1 \leqslant i \leqslant m$.
$\tau_{ij} = \infty$ signifie que T_j ne peut pas être exécuté sur P_i, nous supposons qu'il existe au moins un i pour chaque j tel que $\tau_{ij} < \infty$
Lorsque tous les processeurs sont identiques nous notons τ_j comme étant le temps d'exécution de T_j commun à chacun des processeurs.

d) $R_j = [R_1 (T_j), \ldots, R_s (T_j)]$, $1 \leqslant j \leqslant n$, spécifie les quantités des ressources R_k , $1 \leqslant k \leqslant s$, nécessaires à l'exécution de T_j.
Nous supposons toujours que $R_k (T_j) \leqslant mk$ pour tous k et j.

e)Les poids W_i, $1 \leqslant i \leqslant n$, correspondent aux coûts des tâches T_i :
Le coût pour finir T_i au temps t sera $W_i \cdot t$, avec W_i constant.

Chaque problème étudié peut-être considéré comme étant un cas particulier de cette représentation générale. Il existe une limitation cependant dont il faut tenir compte: l'ordre partiel défini sur τ ne permet pas la représentation des boucles. Donc une boucle éventuelle doit être introduite dans son ensemble comme une seule tâche.

II. 3. 1. 3. Les contraintes de précédence.

Ces contraintes nous amènent à considérer les deux modes principaux d'ordonnancement ci-après:

a) Ordonnancement avec ou sans préemption des tâches.

Dans l'ordonnancement sans préemption de tâches l'exécution d'une tâche ne peut être interrompue une fois qu'elle a débuté. Par contre l'ordonnancement avec préemption des tâches le permet en garantissant toutefois son achèvement, par le même processeur ou par un autre.

b)Ordonnancement suivant une liste de tâches.

Dans ce type d'ordonnancement on dresse une liste ordonnée des tâches qui est souvent appellée liste des priorités. La séquence suivant laquelle les tâches sont assignées au processeurs est alors obtenue par balayages successifs de cette liste. Pratiquement, lorsqu'un processeur devient disponible, la liste est parcourue jusqu'a ce que l'on trouve la première tâche T_i non exécutée qu'il puisse assumer. C'est à dire que la tâche T_i doit être exécutable par ce processeur, et l'exécution de tous ses prédécesseurs doit être achevée, il faut également suffisament de ressources pour satisfaire R_i. On affecte alors T_i au processeur.

Si plusieurs processeurs deviennent disponible simultanément on affecte les tâches d'abord à P_i, puis à P_j et ainsi de suite, avec $i < j$.

II. 3. 1. 4. Les Critères de performances.

Les deux principaux critères de performances sont la durée de l'ordonnancement ou temps maximum d'achèvement $W(O)$ et le temps d'achèvement moyen pondéré $\overline{W}(O)$.

$$W(O) = \max_{1 \leqslant i \leqslant n} \{f_i(O)\}$$

où f_i est le temps d'achèvement de la tâche T_i

$$\overline{W}(O) = \frac{1}{n} \sum_{i=1}^{n} W_i \cdot f_i(O)$$

Il convient donc de trouver des algorithmes efficaces qui minimisent ces quantités pour tous les ordonnancements O appartenant à une même classe.

Nous pouvons encore définir les critères suivants:

- Le temps d'attente $W_i(O)$ (de T_i dans O):

$$W_i(O) = f_i(O) - \tau_i$$

- La date d'achèvement d_i :

Supposons que le modèle général soit étendu de manière à ce qu'un nombre d_i ($1 \leqslant i \leqslant n$) appelé la date d'achèvement soit donné pour chaque tâche T_i . d_i exprime le moment auquel on souhaite voir la tâche T_i achevée. On peut alors définir:

a) le retard d'achèvement de T_i dans O:

$$\tau_i(O) = f_i(O) - d_i$$

b) le retard maximum d'achèvement:

$$\tau_{max}(O) = \max \{0, f_i(O) - d_i\}$$

Lorsque l'on traite des problèmes où les dates d'achèvement doivent être respectées ($f_i(0) \leqslant d_i$, $1 \leqslant i \leqslant n$), alors on les appelle dates critiques.

II. 3. 2. Résultats relatifs aux problèmes d'ordonnancement de tâches (7), (), (31).

II. 3. 2. 1. Efficacité relative des différents modes d'ordonnancement.

L'efficacité de minimisation des critères de temps maximum d'achèvement ou de temps d'achèvement moyen pondéré, diffère selon les modes d'ordonnancement considérés. Elle décroit dans l'ordre suivant: ordonnancement avec préemption, sans préemption et avec liste de tâches.

Ce qui vient d'être dit peut être illustré par l'exemple ci-après:

Soient deux processeurs identiques, s = 0, n = 6

ce qui signifie six tâches avec $\tau_1 = 1$, $\tau_2 = 2$, $\tau_3 = 4$ et $\tau_4 = \tau_5 = \tau_6 = 3$

de plus $T_1 \prec T_3$, $T_2 \prec T_4 \prec T_6$, $T_2 \prec T_5 \prec T_6$

a) Dans le cas d'un ordonnancement avec préemtion on arrive à la solution optimale suivante

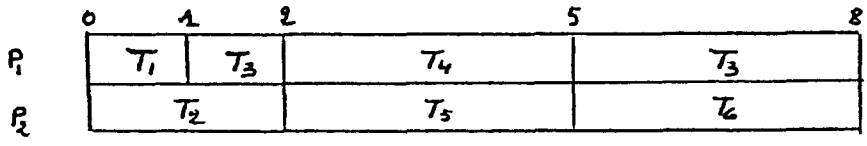


Fig. II-1

donc $W = 8$

$$\overline{W} = \frac{1}{6}(1 + 2 + 5 + 5 + 8 + 8) = \frac{29}{6}$$

b) Dans le cas d'un ordonnancement sans préemption on arrive à la solution optimale suivante:

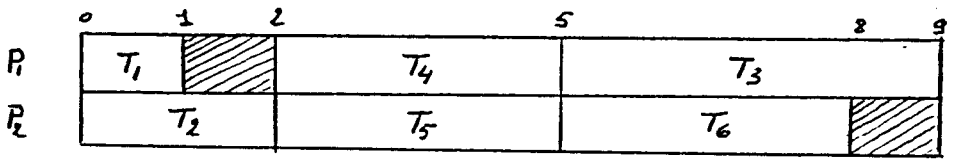


Fig. II-2

donc $W = 9$

$$\overline{W} = \frac{1}{6}(1 + 2 + 5 + 5 + 8 + 9) = \frac{30}{6}$$

c) Dans le cas d'un ordonnancement suivant une liste de tâches:

la liste des tâches sera : T_1

T_2

T_3 après T_1

T_4 après T_2

T_5 après T_2

T_6 après T_5 et T_4

ce qui donne:

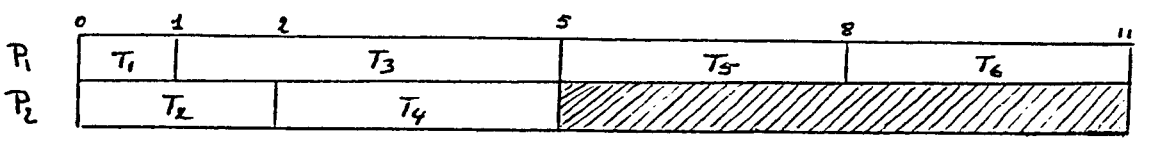


Fig. II-3

donc $W = 11$

$$\overline{W} = \frac{1}{6} (1 + 2 + 5 + 5 + 8 + 11) = \frac{32}{6}$$

II. 3. 2. 2. Possibilités d'ordonnancement de tâches.

Nous nous apercevons (7), (8), (31) qu'il n'est possible de trouver un algorithme efficace, c'est à dire donnant une solution optimale, que dans quelques cas particuliers.

Le premier de ces cas est celui ou les relations ou contraintes de précédence entre les tâches à ordonnancer se présentent sous la forme d'un arbre, voir figure II-4. Les relations de précédence entre tâches impliquent le fait qu'une tâche dont un resultat au moins sert de donnée à une autre doit être exécutée avant cette dernière.

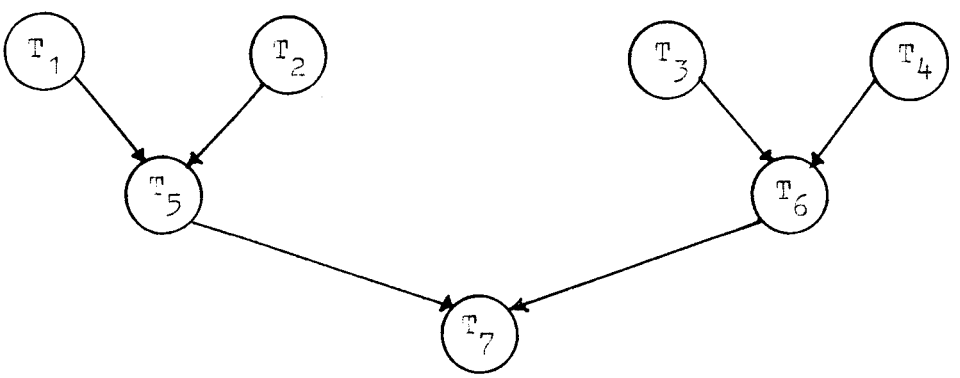


Fig. II-4 exemple de contraintes de précédence en arbre

Notons que cette condition de raltions de précédence en arbre ne suffit pas il faut encore que le temps d'exécution de chaque tâche soit le même.

Une variante consiste a avoir une structure pour les relations de précédence qui ne soit pas un arbre, mais un graphe acyclique orienté quelconque, et seulement deux processeurs.

Nous avons aussi le cas ou les tâches sont de longueurs quelconques (temps d'exécution quelconques) mais à condition qu'elles soient indépendantes et que l'on autorise la préemtion: c'est à dire l'arret de l'exécution d'une tâche en cours au profit d'une autre.

Sans donner l'ensemble complet des cas particuliers pour lesquels il est possible de trouver, à chaque fois, une solution optimale d'ordonnancement nous avons cité les principaux.

Mais notre but n'étant pas de mettre sur pied une machine ne pouvant assumer qu'un type particulier d'ensemble de tâches nous voyons qu'il ne nous sera pas possible d'atteindre une solution optimale.

Autrement dit si le temps d'exécution d'un ensemble de tâches ordonnancées sur une processeur est égal à W, avec m processeurs nous avons un temps $W' > \frac{W}{m}$

Mais en (8) on nous dit aussi qu'il est toujours possible de trouver une solution d'ordonnancement tel que le temps d'exécution W de l'ensemble de n tâches sur m processeurs ($n > m$) soit tel que

$W < W_0 (1 + k)$ avec W_0 le temps d'exécution optimal pour le cas donné et ou k peut s'exprimer comme suit:

$$k = (m - 1)\ \frac{\tau_{i\ max}}{\sum_{i=1}^{n} \tau_i}$$

où $\sum_{i=1}^{n} \tau_i$ est la somme des temps d'exécution des n tâches,

$\tau_{i\ max}$ est le plus grand de ces temps.

Nous voyons que plus le temps d'exécution d'une tâche quelconque est petit vis à vis de la somme des temps d'exécution de toutes les tâches (temps d'exécution de l'ensemble des tâches sur un processeur), plus on se rapproche de la solution optimale. Au contraire plus le nombre de processeurs augmente plus on s'en éloigne.

Cette formule nous permet plus loin d'évaluer le nombre de processeurs souhaitables afin de pouvoir rendre compte des contraintes de temps réel.

C'est le principal bénéfice que nous pouvons retirer des études théoriques sur l'ordonnancement d'un ensemble de tâches par une famille de processeurs.

II. 4. PROPOSITION D'UNE STRATEGIE D'ORDONNANCEMENT.

II. 4. 1. Introduction.

Nous allons faire une proposition de stratégie d'ordonnancement dans le cadre suivant:

- Nous supposons travailler avec des processeurs identiques sur tous les plans, c'est à dire banalisés.

- Nous n'envisageons comme ressources supplémentaires que la mémoire locale de chaque processeur et une mémoire commune.

- Nous n'envisageons aucune restriction relative aux contraintes de précédence autre que l'absence de boucles.

- Nous supposons évalué les temps d'exécution de chacune des tâches, nous les supposons également petit par rapport à la longueur de l'ordonnancement.

Nous ne retenons pas la préemption, estimant que les tâches étant supposées relativement courtes leur suspension en cours d'exécution entrainerait une perte de temps non négligeable.

- Nous comptons proposer une stratégie d'ordonnancement basée sur une liste de tâches. Cette liste est fournie par l'échéancier Nous comptons corriger l'efficacité moindre de cette méthode en envisageant de pouvoir traiter simultanément deux jeux de données.

Comme au départ de la description des tâches il n'est pas évident de connaitre leur temps d'exécution ni de voir clairement les contraintes de précédence nous allons envisager ces problèmes avant de passer effectivement à la proposition de la stratégie d'ordonnancement.

II. 4. 2. Evaluation du temps d'exécution des tâches.

Le temps d'exécution des tâches est généralement variable et dépend essentiellement des valeurs des données traitées. En ce qui concerne l'ordonnancement c'est le temps moyen d'exécution qui doit intervenir. Mais comme nous le verrons la connaissance du temps d'exécution est principalement importante pour l'évaluation du nombre m de processeurs et dans ce cas il convient de prendre le temps d'exécution le plus défavorable. Cela n'est certe pas évident et demande l'étude séparée de chaque tâche que l'on doit spécifier avec précision et pour laquelle il faut faire avec attention le choix de l'algorithme qui l'implémente.

Notons que cela n'a rien d'impossible, ainsi les constructeurs donnent ces temps pour les tâches que les processeurs spécialisés sont à même de réaliser.

Naturellement cela implique de préciser les processeurs utilisés et pour ceux-ci de constituer une bibliothèque de tâches. Le découpage de ces tâches doit être suffisamment fin.

II. 4. 3. Recherche des contraintes de précédence

Il existe des recherches sur les méthodes de détection automatique de séquences parallèles et en particulier, nous pouvons citer celles de BERNSTEIN (37). Les méthodes qui en résultent s'appliquent à des segments de programmes existant et travaillent sur les variables communes à chacun d'entre eux.

Notre cas correspond à la situation où rien n'existe sinon une bibliothèque de tâches considérées sans possibilité de découpage supplémentaire au niveau du parallélisme.

A condition que l'outil de description des tâches en tienne compte, il est possible de savoir pour chacune d'elles, si les données utilisées sont des données d'entrées ou des résultats produits par une ou plusieurs autres tâches et par lesquelles, nous pouvons dès lors établir les contraintes de précédence en utilisant la méthode suivante:

a) Nous considérons toutes les tâches terminales au niveau 1.

b) D'une manière générale, pour chaque tâche au niveau i nous recherchons la ou les tâches lui fournissant des données, ces tâches sont placées au niveau i + 1, même si elles

ont déjà été assignées à un niveau inférieur. Dans ce dernier cas dependant, il faut relever aussi le niveau des tâches déjà assignées et qui sont reconnues comme prédécesseurs de la tâche dont le niveau vient d'être relevé ainsi que leur propres prédécesseurs.

A chaque tâche de niveau i + 1, nous associons les tâches de niveau i qui sont pour elle des successeurs immédiats. Les données d'entrée sont considérées, le cas échéant, comme prédécesseurs et de ce fait peuvent être associées à certaines tâches.

c) Lorsque toutes les tâches du dernier niveau n'ont comme prédécesseurs immédiats que des entrées, toutes les tâches doivent avoir été assignées et la procédure prend fin.

Il convient néanmoins de faire les remarques suivantes:

a) Après application de la procédure déterminant les contraintes de précédence entre tâches, toutes les tâches d'un même niveau sont indépendantes.

b) Certaines tâches assignées à un niveau intermédiaire i peuvent n'avoir comme prédécesseurs que des entrées ce qui permet de les réassigner à n'importe quel niveau supérieur à i.

c) Il se peut que les niveaux des prédécesseurs immédiats de certaines tâches assignées au niveau j aient été relevées au moins en j + n avec $n > 1$.
Ces tâches peuvent alors être réassignées à n'importe quel niveau j' avec $j \leqslant j' < j + n$. On utilise ces possibilités de réassignation pour avoir de préférence au moins m tâches par niveau, principalement pour les premiers niveaux.

II. 4. 4. Stratégie d'ordonnancement

Les tâches à réaliser étant classées par niveau, nous pouvons définir une stratégie permettant de les allouer aux m processeurs.

Il n'est pas rentable d'effectuer l'ensemble des tâches niveau par niveau car normalement, tous les processeurs ne termineront pas simultanément et il y en aura donc souvent en attente. Il faut alors pouvoir commencer les tâches du niveau suivant, même plus généralement d'un des niveaux suivants, avant d'avoir terminé celles du niveau actuel. A la limite, lorsqu'un processeur ne trouve plus de tâches dans aucun des niveaux suivants, il commence celles du premier niveau de l'ensemble suivant de données, ce qui est, nous semble-t-il, la façon la plus naturelle et la plus efficace d'introduire le mode pipeline.

Il y a néanmoins lieu d'éviter d'exécuter simultanément trop de tâches de niveaux différents, on cherche pour cela à donner la priorité à chaque niveau aux tâches qui ont le plus de successeurs au niveau suivant.

Pratiquement, l'on peut procéder comme suit:

1) Construire une liste des tâches classées niveau par niveau et pour chaque niveau faire le classement par ordre décroissant du nombre de successeurs immédiats. En cas d'égalité, placer d'abord les tâches dont les successeurs ont le plus long temps d'exécution.

2) On alloue les m premières tâches aux m processeurs.

3) Lorsqu'un processeur termine une tâche:

a) si cette tâche est terminale on inscrit le résultat dans la table correspondante, si de plus, c'est le dernier

résultat d'un ensemble, on applique les résultats.

b) on alloue à ce processeur la première tâche de la liste dont tous les prédécesseurs sont effectués. Si l'on ne trouve aucune tâche exécutable dans la liste, on autorise l'acquisition d'un nouvel ensemble de données et l'on alloue au processeur la première tâche de la liste avec ces nouvelles données.

c) on alloue évidemment par priorité, les tâches exécutables avec le plus ancien des ensembles de données.

A titre d'exemple, nous traitons l'ensemble de calculs présentés en (39) que nous considérons exécutés par 2 processeurs du type MM57109 spécialisé dans le calcul numérique.

Soit, donc à calculer l'ensemble $(Y_1, Y_2, Y_3, Y_4, Y_5)$ à partir de données (X_1, X_2) et défini par:

$$Y_1 = \sin (X_1)$$

$$Y_2 = \cos (X_2)$$

$$Y_3 = \sqrt{X_1 + Y_2 X_2}$$

$$Y_4 = (Y_1 + Y_2)^2$$

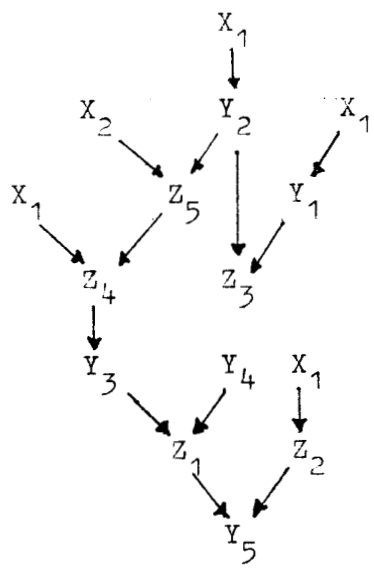
$$Y_5 = \frac{Y_3}{Y_4} + X_1^2$$

Certains de ces calculs doivent être décomposés et l'on obtient ainsi l'ensemble de tâches suivant:

$Y_1 = \sin(X_1)$	562
$Y_2 = \cos(X_2)$	562
$Z_4 = X_1 + Z_5$	22
$Z_5 = Y_2 X_2$	32
$Y_3 = Z_4$	70
$Y_4 = Z_3^2$	30
$Z_3 = Y_1 + Y_2$	22
$Y_5 = Z_1 + Z_2$	22
$Z_1 = \frac{Y_3}{Y_4}$	78
$Z_2 = X_1^2$	30

La colonne de droite donne pour chaque tâche le temps d'exécution moyen exprimé en centaines de cycles et repris par le tableau relatif au MM57109.

Les contraintes de précédences font apparaître le classement suivant des tâches par niveau de précédence:



niveau	
1	Y_2 (1)
2	Y_1 (1 ou 2), Z_5 (2)
3	Z_3 (3 ou 2 (Y_1)), Z_4 (3)
4	Y_4 (4 ou 3 (Z_3), Y_3 (4)
5	Z_2 (5 à 1), Z_1 (5)
6	Y_5 (6)

Finalement, comme on travaille avec deux processeurs on cherche à avoir au moins deux tâches par niveau, principalement, pour les premiers niveaux:

niveau 1 : Y_1, Y_2

2 : Z_3, Z_5

3 : Y_4, Z_4

4 : Y_3, Z_2

5 : Z_1

6 : Y_5

Ce qui donne la liste des tâches:

$$L\ (Y_2, Y_1, Z_3, Z_5, Y_4, Z_4, Y_3, Z_2, Z_1, Y_5)$$

En considérant que $(X_1, X_2, Y_1 \ldots Y_5, Z_1 \ldots Z_5)$ correspondent à un premier ensemble de données et que $(X'_1, X'_2, Y'_1 \ldots Y'_5, Z'_1 \ldots Z'_5)$ correspondent à un second ensemble de données, nous avons le déroulement suivant des allocations de tâches aux processeurs P_1 et P_2:

t	P_1	P_2
0	Y_2	Y_1
562	Z_3	Z_5
584	Y_4	Z_5
594	Y_4	Z_4
614	Z_2	Z_4
616	Z_2	Y_3
644	Y'_2	Y_3
686	Y'_2	Z_1

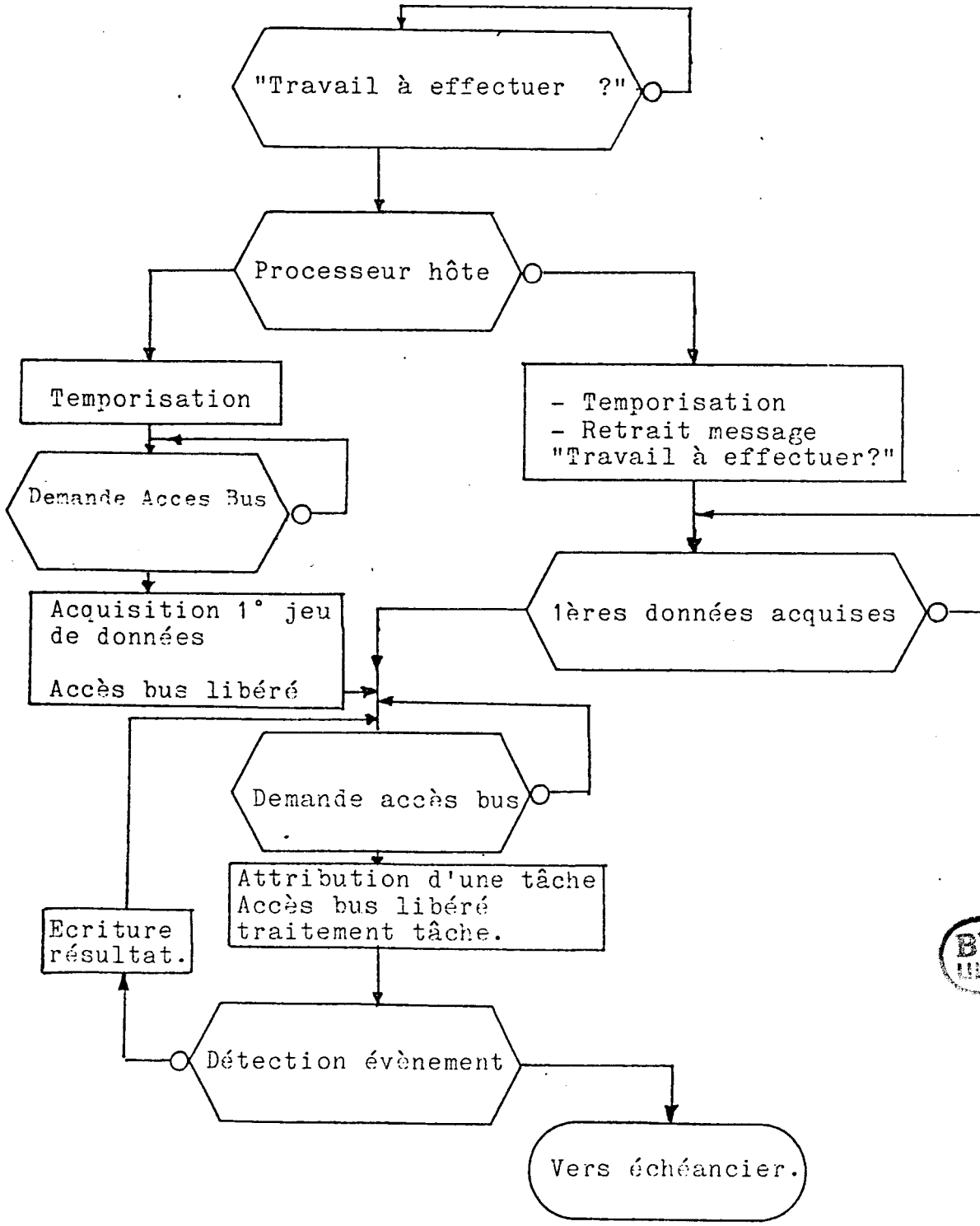
764	Y'_2	Y_5
786	Y'_2	Y'_1
1206	Z'_5	Y'_1
1238	Z'_4	Y'_1
1260	Y'_3	Y'_1
1330	Z'_2	Y'_1
1348	Z'_2	Z'_3
1360	Y''_2	Z'_3
1370	Y''_2	Y'_4
1400	Y''_2	Z'_1
1478	Y''_2	Y'_5
1500	Y''_2	Y''_1
....	....	....

Notons qu'après un certain nombre de tours il y a lieu de faire une pause pour resynchroniser le tout. Ce nombre de tours est à définir expérimentalement sur divers ensembles de tâches.

II. 5. IMPLANTATION DE LA STRATEGIE D'ORDONNANCEMENT.

En fait, l'implantation de la stratégie d'ordonnancement vue précédemment se fait par l'intermédiaire d'un programme résidant sur chacun des processeurs et d'une liste d'attente en mémoire commune.

Ce programme répond à l'ordinogramme ci-après:


"Travail à effectuer ?"
Processeur hôte
Temporisation
Demande Acces Bus
Acquisition 1° jeu de données
Accès bus libéré
- Temporisation
- Retrait message "Travail à effectuer?"
1ères données acquises
Demande accès bus
Attribution d'une tâche
Accès bus libéré
traitement tâche.
Ecriture résultat.
Détection évènement
Vers échéancier.

Afin de rendre cet ordinogramme plus compréhensible, il nous faut donner quelques précisions:

1 - La notion de processeur hôte provient de ce que parmi tous les processeurs, un est relié à un opérateur via un terminal, ce qui lui confère un statut et un rôle quelque peu différent des autres. Un sémaphore identifie le processeur hôte et cette identification a lieu lors de l'initialisation.

2 - L'accès au bus est normalement géré par un circuit arbitre que nous étudions au dernier chapitre, mais son arbitrage est remis en question à chaque lecture et écriture en mémoire commune. Un accès plus prolongé doit donc être géré par logiciel à l'aide d'un sémaphore et d'une routine qui permet à chaque processeur de savoir si le bus est "libre" et de signaler dans l'affirmative qu'il en prend possession.

3 - Les temporisations introduites sont là pour donner le temps à chaque processeur de prendre connaissance du message avant de l'effacer.

4 - Parallèlement au travail du processeur, il y a lieu de prévoir une horloge temps réel activée lors de l'initialisation et qui l'interrompe périodiquement pour l'amener à vérifier si, suite à la détection d'un événement par un autre processeur il n'y a pas lieu de retourner en attente de travail à effectuer. Notons que la période de répétition de ces interruptions ne peut être de beaucoup supérieur au temps d'exécution de l'échéancier.

Cela étant dit, il est bon de remarquer que la partie importante de cet ordinogramme est le bloc "attribution d'une tâche" que nous allons détailler maintenant.

Pour s'attribuer une tâche le processeur doit pouvoir déterminer les points suivants:

- 1) Y a-t-il un ou deux jeux de données en cours de traitements?

- 2) Quel est le jeu le plus ancien?

- 3) Quelle tâche a déjà été effectuée pour l'un et l'autre jeu (deux jeux maximum) ?

- 4) Quel est le niveau d'une tâche?

- 5) Tous les prédécesseurs d'une tâche donnée ont-ils été effectués ?

- 6) Combien de données sont nécessaires à la tâche attribuée ?

- 7) Où se localisent ces données ?

- 8) Le résultat de la tâche permet-il de vérifier l'occurence d'un événement ou est-il un résultat intermédiaire devant être écrit dans une table ?

- 9) Où écrire ce résultat ?

-10) Quelle est l'adresse d'entrée de la tâche ?

Comme l'on dispose de deux jeux de données possible, on commence par définir deux pointeurs de début de pile de données: le pointeur A et le pointeur B, ces pointeurs sont localisés à des adresses fixes, bien entendu, et c'est leur contenu qui donne les adresses de départ des deux piles. Puisqu'il faut savoir quel jeu est en cours de traitement et lequel est le plus ancien on utilise trois bits d'un octet:

le bit 0 : pointeur A en traitement (1) ou non (0)

1 : pointeur B en traitement (1) ou non (0)

2 : pointeur A le plus ancien (1) ou inverse (0)

On appelle cet octet, l'octet d'état des pointeurs de piles de données, ou plus simplement: octet d'état.

Le pointeur A, le pointeur B et surtout l'octet d'état permettent aux processeurs de répondre aux questions 1 et 2. Les informations leur permettant de répondre aux autres questions sont données par la structure d'un objet de la liste d'attente.

Pour répondre aux questions 6 à 10, il n'y a aucun problème, il suffit de réaliser la structure suivante:

- un octet donne le nombre de données correspondant à la tâche.

- cet octet est suivi d'un mot (16 bits) donnant le déplacement par rapport au début de la pile de données permettant de localiser la première donnée de la tâche considérée.

- ce mot est suivi par un octet dont le code indique si le résultat est "à écrire" ou à "tester".

- cet octet est suivi d'un mot qui n'est pris en considération que dans le cas d'un résultat "à écrire" et qui contient le déplacement par rapport au début de la pile des résultats ce début étant contenu par le pointeur R (pointeurs RA et RB).

- un dernier mot contient l'adresse du point d'entrée de la tâche.

Il nous reste à donner le niveau d'une tâche et la possibilité de trouver l'information concernant l'exécution des

prédécesseurs, ainsi que de savoir pour quel jeu de données la tâche a déjà éventuellement été exécutée ou est en cours d'exécution.

Pour cela, la structure décrivant une tâche dans la liste d'attente commence par un octet dont les deux bits de poids fort disent si la tâche a été ou non attribuée pour les jeux de données actuels (bit 7, jeu A; bit 6, jeu B; 1 pour "déjà attribué"). Le restant de l'octet contient un code indiquant si nous avons affaire à un tâche indépendante (3F) ou à une sous-tâche (3E).

Les octets suivant ne donnent des indications réellement que dans le cas de "sous-tâches".

- le premier donne le nombre de prédécesseurs.

- les suivants, groupés en mots, donnent par rapport aux pointeurs R la situation des résultats de chacun des prédécesseurs.

Notons que nous utilisons encore un pointeur de début et un pointeur courant pour la lecture de la liste d'attente. Ce pointeur courant permet, comme nous allons le voir ci-après, de ne pas devoir recommencer à chaque fois, le test de toutes les tâches de la liste.

Notons également que l'information niveau de la tâche a disparu de la structure précédente mais, comme nous allons le voir aussi ci-après, elle n'est pas absolument nécessaire.

Cette structure est utulisée pour réaliser "l'attribution d'une tâche" par l'algorithme suivant:

1 - Tester l'octet d'état pour trouver le pointeur le plus ancien.

2 - Lire le pointeur courant le plus ancien.

3 - Lire le contenu correspondant à ce pointeur, donc dans la liste d'attente:

a1) Si 00, c'est que l'on est en fin de liste, vérifier à partir du début de liste si toutes les tâches ont été effectuées pour le jeu de données correspondant et par test du bit convenable:

b1) Si non, à la première tâche rencontrée, non effectuée, on se l'attribue (saut en d1), le pointeur courant ayant été réactualisé.

b2) Si oui, on vérifie si l'autre jeu de données est en cours de traitement (par test de l'octet d'état)

c1) Si oui, on permute l'ancienneté (octet d'état) et l'on fait l'acquisition de nouvelles données pour le pointeur qui vient de terminer, on réinitialise les marqueurs (liste d'attente) ainsi que la pile des résultats correspondante puis on se rebranche en 2 .

c2) Si non, on fait la même chose qu'en c1 sauf qu'on ne touche pas à l'ancienneté.

a2) Si différent 00, lire le premier octet pour confirmation que la tâche n'a pas encore été exécutée avec ce jeu de données et vérifier si l'on a affaire à une tâche et non à une sous-tâche:

d1) Si oui: - charger les données dans le processeur (lire le nombre de donnée et la position de la première, puis charger).

- vérifier si le résultat est à "tester" ou à "écrire" placer un "drapeau"

local en position correspondante.

- s'il y a lieu charger l'adresse où écrire le résultat.

- charger l'adresse du point d'entrée de la tâche.

-"marquer" la tâche.

- sortir de l'"attribution d'une tâche".

d2) Si non, vérifier si tous les résultats des prédécesseurs sont "prêts" en contrôlant que les seconds bytes des résultats soient au code ad hoc.

e1) si oui, on se branche en d1.

e2) si non, on envisage la tâche suivante, on se branche en 3.

Notons que les résultats sont écrits sur un octet chaque fois doublé d'un octet de contrôle. Ainsi, à la limite, on pourrait prévoir des résultats sur 15 bits, le 16ème servant au contrôle.

II.6. CONCLUSIONS.

Dans ce chapitre, nous avons pu définir l'essentiel de ce qui nous est nécessaire comme structure matérielle de la station et qui est détaillée au dernier chapitre. Nous avons précisé le logiciel global nécessaire et détaillé la stratégie d'ordonnancement. Nous avons vu que l'ordonnancement faisait appel à des routines comme:

- l'attribution d'une tâche.
- la demande d'accès au bus.
- l'acquisition de données.

La première de ces routines qui est aussi la principale et la plus complexe a été étudiée avec attention et s'appuie sur une structure de la liste d'attente qui a été elle aussi détaillée.

Dans le chapitre suivant, nous allons nous occuper plus particulièrement du logiciel permettant de passer de l'énoncé de l'application à implanter à la liste d'attente. Nous devrions dire aux listes d'attente car l'exécution d'une application fait apparaître une succession de liste d'attente, à chaque fois mise à jour par le programme échéancier que nous verrons également.

C H A P I T R E III

D E S C R I P T I O N E T I M P L A N T A T I O N D' U N E A P P L I C A T I O N.

III. 1. INTRODUCTION.

Au cours de ce chapitre nous allons, a partir de la description de l'application à implanter, chercher à obtenir les différents ensembles successifs de tâches à exécuter.

Nous allons d'abord choisir l'outil de description de l'application. Parmi les outils graphiques existant pour décrire un système séquentiel possédant un certain degré de parallèlisme nous avons essentiellement le Grafcet et les réseaux de Pétri.

L'introduction de la description de l'application dans la machine servant au développement se fait de la manière la plus aisée par l'intermédiaire d'un langage que nous définissons.

Le texte ainsi édité doit être transposé dans la mémoire de la machine cible sous une forme directement exploitable. C'est à

dire que nous transposons une partie en structure de données: le graphe est traduit en un ensemble de tables et à chaque tâche est associé un descripteur. Le restant est traduit en un ensemble de programmes correspondant chacun à une tâche.

Enfin nous élaborons un programme qui en phase d'exécution puisse manipuler les tables précédentes afin d'établir à chaque instant l'ensemble des tâches devant être exécutées. Ce programme nous l'appelons l'échéancier.

III. 2. OUTIL DE DESCRIPTION.

III. 2. 1. Choix de l'outil.

Comme nous l'avons laissé entendre précédemment le choix se limite pratiquement au Grafcet ou aux réseaux de Pétri.

Pour le détail de ces outils nous renvoyons en (5), (33) et (34). Sachons cependant que chacun d'eux possède ses caractéristiques et donc ses qualités propres.

Le Grafcet est un outil puissant mais difficile à maitriser. Les réseaux de Pétri sans être simples, loin de là, ont toutefois l'avantage de ne présenter qu'un seul cas de conflit. Ce cas est celui ou il existe une place d'entrée commune à deux ou plusieurs transitions validées simultanément, afin que l'évolution reste unique il est nécessaire d'établir une règle de priorité entre les diverses transitions validées.

Dans le cadre d'un système multiprocesseurs il est particulièrement interessant de n'avoir à tenir compte que d'un

seul cas de conflit, c'est pourquoi nous orientons notre choix vers les réseaux Pétri.

III. 2. 1. 1. Réseaux de Pétri ordinaires.

En reprenant (33) nous allons rappeler brièvement les définitions de bases des réseaux Pétri:

Un réseau de Pétri est un graphe orienté défini par un quadruplet $\{ P, T, A, Mo \}$ où:

- $P = \{P_1, P_2, \ldots, P_m\}$ est un ensemble fini de places représentées par des cercles.
- $T = \{t_1, t_2, \ldots, t_1\}$ est un ensemble fini de transitions représentées par des tirets.
- $A = \{a_1, a_2, \ldots, a_n\}$ est un ensemble fini d'arcs orientés qui assurent la liaison d'une place vers une transition et inversément.
- $Mo = \{P \longrightarrow N_+\}$ est le marquage initial du graphe précisé par la présence à l'intérieur des cercles représentant les places d'un nombre nul ou fini de marqueurs. Une place peut donc être vide ou marquée. Les places immédiatement en amont d'une transition en sont les places d'entrées, celles immédiatement en aval en sont les places de sortie. Une place peut être à la fois place d'entrée et de sortie d'une transition.

Règle de validation et de tir d'une transition.

L'application de la règle de validation et de tir d'une transition permet de faire évoluer séquentiellement le marquage d'un réseau.

Par définition, une transition est validée si chaque place d'entrée de cette transition comporte au moins un marqueur. Une transition validée peut être tirée ce qui revient à enlever un marqueur à chaque place d'entrée de la transition et à en ajouter un à chaque place de sortie. Lorsque plusieurs transitions d'un graphe sont validées, leurs tirs s'effectuent simultanément. C'est ici qu'intervient la possibilité de conflit dont nous avons fait part précédemment.

Notion de réseau sain.

Un réseau est dit sain (ou sauf) pour un marquage initial Mo si quel que soit le marquage obtenu à partir de Mo par une séquence finie de tirs, aucune place ne possède plus d'un marqueur.

III. 2. 1. 2. Réseaux de Pétri Interprétés.

Toute machine séquentielle peut être représentée par un réseau de Pétri "sauf" et "sans conflit" en utilisant l'une des deux interprétations suivantes:

Réseau de type S:

A toute transition est associée une proposition logique relative aux entrées de la machine séquentielle. Pour que le tir d'une transition puisse avoir lieu il faut non seulement qu'elle soit validée mais en plus la proposition logique associée doit être vraie.

A toute place du graphe est associée une ou plusieurs variables de sortie conditionnées ou non par des grandeurs d'entrées de la structure séquentielle. Ces variables sont affirmées lorsque les places correspondantes sont marquées.

A tout marquage du réseau est associé un état de la machine séquentielle.

Réseau de type t:

Ce réseau ne diffère du réseau de type S que par l'interprétation relative aux grandeurs de sortie affectées aux transitions et indiquées entre parenthèses.

Notons qu'il n'y a pas de contre indication a avoir une interprétation de type S et t simultanément.

III. 2. 2. Langage d'introduction en machine d'un réseau de Pétri Interprété.

La plupart des applications correspondent à des réseaux de Pétri trop complexes pour être représentés de manière intelligible sur l'écran d'un terminal. Il convient donc de transposer le réseau de Pétri en un langage approprié afin de pouvoir l'introduire en machine à l'aide d'un programme éditeur.

Le texte ainsi édité doit être traduit de manière exploitable par la machine en phase d'exécution de l'application. Cette traduction donne finalement soit un programme directement exécutable soit une structure de donnée utilisée par un programme interpréteur.

La première solution nous semble difficilement réalisable dans le cadre d'un système multiprocesseurs de par la difficulté que poserait la décomposition en tâches.

En fait, sachant que nous allons travailler avec des réseaux de Pétri Interprétés, nous constatons avoir un ensemble de tâches correspondant aux différentes propositions logiques associées aux transitions. Nous avons un deuxième ensemble de tâches correspondant à l'application des variables de sortie associées soit à chaque transition, soit à chaque place, suivant le type d'interprétation.

Ces différentes tâches doivent à notre avis être directement compilées et de plus, pour chacune d'elles, doit être créé le descripteur correspondant dont nous avons parlé au chapitre précédent.

Le réseau de Pétri proprement dit va servir essentiellement à déterminer parmis l'ensemble des tâches précédentes quel est le sous-ensemble qui à chaque instant doit être effectivement exécuté.

Nous estimons dès lors qu'il vaut mieux traduire le réseau de Pétri en une structure de données qui est interprétée par un programme servant à dresser les listes de tâches successives devant être activées. Il s'agit du programme échéancier dont nous avons déjà parlé.

Ce qui vient d'être dit nous amène a envisager la traduction de la description partiellement par un langage interprété et par un langage compilé pour le restant. La description comporte donc bien deux parties distinctes qui peuvent être formulées en des langages différents.

III. 2. 2. 1. Langage de description du réseau de Pétri proprement dit.

Il est nécessaire avant tout d'indiquer le début de la description, de donner le nombre total de places du graphe et le numéro des places initialement marquées.

Comme indicateur de début de description nous pouvons utiliser le symbole " *". Dès lors la première phrase de la description que l'on termine par le mot " ; " prend l'allure suivante:

*⟨nom de l'application⟩⟨espace⟩⟨nombre de places totales⟩
⟨espace⟩ PL⟨ numero ⟩, PL ⟨numero ⟩,...,
PL ⟨ numero ⟩ , ⟨ espace ⟩ ;

Par exemple:

* TRAV1 7 PL1 ; (voir figure III-1)

Ensuite viennent une série de phrases, chacune décrivant une place, c'est à dire donnant pour cette place le nom de la tâche associée (qui peut être complexe, mais peut être en particulier l'activation des sorties) et les numéros des transitions qui la suivent immédiatement:

PL ⟨numero⟩⟨espace⟩⟨nom de tâche⟩⟨espace⟩ T ⟨numero ⟩,...,
T ⟨numero⟩ ⟨espace⟩ ;

Par exemple:

PL2 MD T2 ; (voir figure III-1)

Notons que si MD est conditionnelle cette condition est incluse dans la définition de la tâche qui lui correspond.

Nous pouvons constater que le nombre de ces phrases doit être égal à celui donné dans la phrase initiale ce qui est un premier contrôle possible.

Enfin vient un ensemble de phrases décrivant les transitions du graphe, c'est à dire donnant le nom de la proposition logique associée, les numéros des places d'entrée (PE) et ceux des places de sortie (PS):

TR <numéro><espace><nom proposition associée>
<espace> PE <numéro> ,..., PE <numéro><espace>
PS <numéro> ,..., PS <numéro><espace> ;

Notons que la dernière de ces phrases se termine par le mot "*" plutot que par le mot " ; " de manière à indiquer la fin de cette partie de la description.

Un exemple peut être le suivant:

TR1 D PE1 PS2,PS3 ;
(voir figure III-1).

Pour compléter ce qui vient d'être dit nous allons donner la transposition du réseau de Pétri de la figure III-1:

* TRAV1 7 PL1 ;

PL1 T1 ;

PL2 MD T2 ;

PL3 MH T3 ;

PL4 T4 ;

PL5 T4 ;

PL6 MG T5 ;

PL7 MB T6 ;

TR1 D PE1 PS2,PS3 ;

TR2 V2 PE2 PS4 ;

TR3 H2 PE3 PS5 ;

TR4 E PE4,PE5 PS6 ;

TR5 V1 PE6 PS7 ;

TR6 H1 PE7 PS1

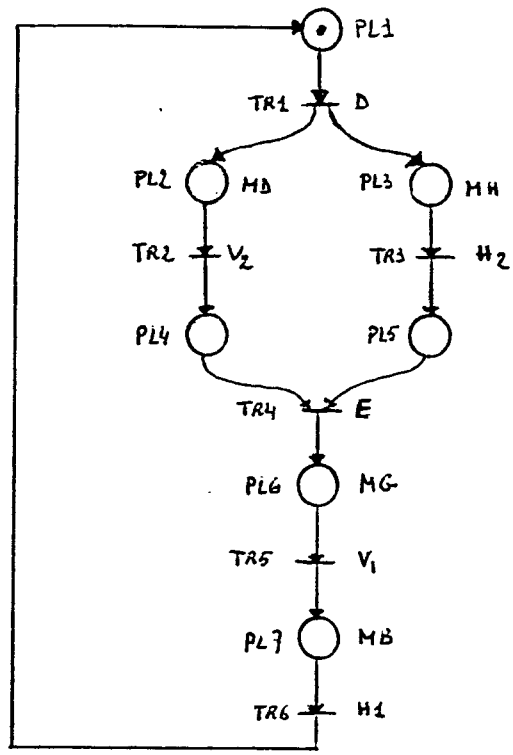


Fig. III-1

III. 2. 2. 2. Langage d'introduction de l'interprétation.

Les tâches correspondant aux propositions logiques associées aux transitions, comme les affirmations des variables de sortie, sont introduites en machines à l'aide d'un langage compilé ou rapide en tous cas. Ce langage ne doit pas être trop différent de celui servant à introduire le réseau de Pétri proprement dit.

Il y a un langage existant qui semble répondre parfaitement à ces critères, c'est le FORTH. Ce langage travaille par création de "mots" ou enrichissement de vocabulaire. Nous ne le décrivons pas en détail, nous renvoyons pour cela à la littérature spécialisée et en particulier à (6), (9) et (26).

Toutefois nous allons décrire un de ses aspects montrant que l'on peut développer un mot FORTH (une tâche) avec les facilités qu'offre un langage interprété de haut niveau et obtenir un produit tournant presque aussi rapidement qu'un programme assembleur. Cette particularité est bien sur de première importance dans le cadre de notre travail.

Un mot FORTH est construit en mémoire selon une structure constituée d'une "tête" et d'un "corps". La "tête" est essentiellement constituée de la longueur du nom, des codes ASCII des lettres formant le nom et d'un pointeur de lien permettant de chaîner les différents mots définis dans le dictionnaire. Le "corps" commence par un code définissant sa nature c'est à dire le CFA (Code Field Address) du mot. Ce CFA est suivi des pointeurs des CFA des mots constituant le mot en question. Le dernier pointeur de CFA étant celui du mot " ; " qui termine tout mot écrit en FORTH.

Il est interessant de savoir que les codes CFA sont en fait des adresses de routines assembleur qui "décident" du traitement à réserver au mot selon sa nature.

Il est donc aisé en lisant les CFA des mots constituant le mot étudié de déterminer leur nature. Or la perte en temps d'exécution provient de ce qu'un mot FORTH peut être constitué de mots FORTH eux-mêmes constitués d'autre mots FORTH et ainsi de suite sur plusieurs niveaux. A l'exécution, il y a donc perte de temps du au passage de piles de mots en piles de mots.

En lisant les codes CFA dont on vient de parler il est possible, avant exécution, de procéder à l'opération que nous avons appelé "dépliage". Cette opération transforme la pile de pointeurs de CFA d'un mot en une pile de pointeurs se branchant tous sur des routines assembleur. Cette opération donne donc pour le mot concerné une structure très proche de celle constituée d'un programme principal assembleur faisant appel à des sous-routines à un seul niveau. Les temps d'exécution FORTH et assembleur sont alors fort comparables.

A présent nous avons l'outil permettant d'introduire en machine l'ensemble des tâches correspondant aux propositions logiques associées aux transitions ainsi qu'aux activations des sorties.

Si nous reprenons l'exemple de la figure III-1 et que l'on suppose avoir:

V2 = a.b avec a et b deux variables d'entrée

Nous pourrons alors définir V2 de la manière suivante:

```
< adresse 1 > CONSTANT  a
< adresse 2 > CONSTANT  b
: V2  a @ b @ AND ;
```

Nous pourrions envisager de déposer le résultat dans une adresse de la mémoire commune, soit PV2 le nom de cette adresse dans le cas présent:

```
<adresse X> CONSTANT  PV2
: V2  a @  b @  AND PV2 ! ;
```

Bien entendu nous traitons comme cela des variables de seize bits. Or généralement ce genre de variable est d'un seul bit. Alors a, b, ... ne seront plus définis comme des constantes, mais seront des mots FORTH écrit en assembleur. Ces mots iront lire un bit précis de la mémoire d'entrée et déposeront le contenu sur la pile paramètre. Nous aurons alors:

```
: V2  a b AND PV2 ! ;
```

III. 2. 2. 3. Décomposition des propositions logiques en sous-tâches

Si l'on a une proposition logique particulièrement complexe et dont le temps d'exécution est jugé trop long, elle peut être décomposée en plusieurs tâches de la façon suivante:

```
: PR101  a b AND c d AND OR ;
```

Dans cet exemple nous avons trois niveaux de procédure avec les contraintes comme le montre la figure III-2 (les fonctions AND et OR étant à deux entrées).

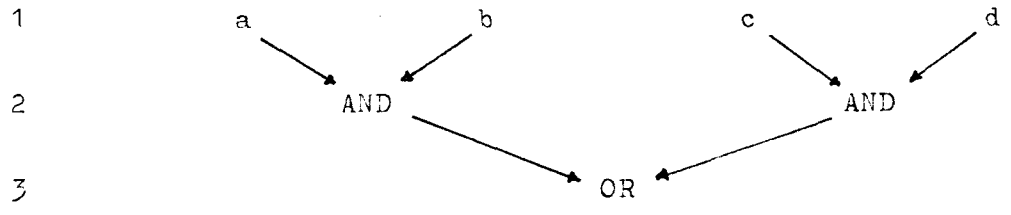


Fig. III-2

Notons que la structure de pile du langage FORTH et le fait que tout nouveau mot est défini par une suite d'autres mots préalablement définis permet une décomposition très simple en niveaux de précédence.

Il faut remarquer que nous n'avons ici que deux types de mots:

a) ceux correspondant à une acquisition de données

b) ceux correspondant à une opération logique.

Alors, en commençant l'analyse de la gauche vers la droite, le premier mot est placé au niveau 1. Chaque fois que l'on rencontre un mot de type b on descend d'un niveau. Chaque fois que l'on passe d'un mot de type b à une suite de mots de type a, on remonte d'un niveau. Cela si le mot de type b suivant ne nécessite pas un nombre d'entrées supérieur au nombre de mots de la suite, sinon on reste au même niveau.

Une deuxième passe peut permettre de remonter les mots de type a concernant l'acquisition de variables d'entrée de manière à équilibrer le contenu des différents niveaux.

Il est évident qu'il ne sert à rien de trop décomposer. Il faut en effet que le temps d'exécution des sous-tâches reste, en moyenne, grand devant celui de la routine d'attribution d'une tâche. Nous avons pu déterminer expérimentalement ce dernier à cent microsecondes environs ceci sur notre système particulier qui est basé sur le μp 68000

III. 2. 3. Implantation de l'outil de description.

En réalité il reste essentiellement à trouver une structure de données représentant un réseau de Pétri et qui

soit efficace pour la mise en oeuvre du programme échéancier.

Nous pouvons distinguer d'une part le marquage initial Mo et d'autre part le graphe proprement dit constitué de P, T et A.

Mo est représenté par une suite d'emplacement mémoire ou chacun correspond à une place du réseau. Le ième emplacement coïncide avec la ième place. Si nous ne considérons qu'un seul marqueur par place un seul bit par emplacement suffit. Alors les bits à 1 indiquent les places marquées et évidemment les bits à 0 les places non marquées.

Il faut encore un pointeur de début et un pointeur de fin pour localiser Mo, soit respectivement PDMo et PFMo.

Dans la mesure ou les mémoires sont généralement organisées en mots de plus de un bit, normalement en octets, il faut associer à PFMo un nombre donnant les bits du dernier mot dont il faut tenir compte (a partir du bit de poids le moins élevé) ou veiller à mettre à zéro les bits non significatifs.

La figure III-3 donne la représentation de Mo pour le réseau de la figure III-1.

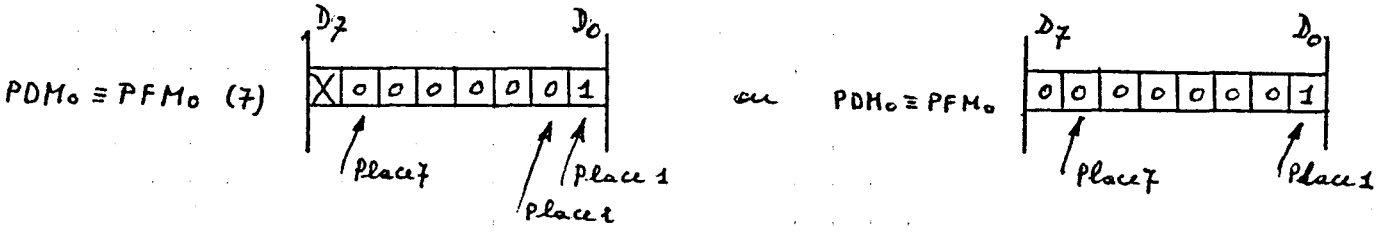


Fig. III-3

Une table supplémentaire est nécessaire et pourrait peut-être suffire pour représenter le réseau de Pétri. Il s'agit de la table que nous appelons la table transitions-places (TTP).

Un objet de cette table est organisé de la manière suivante (il décrit en fait une transition du réseau):

- Un premier octet donne la longueur du nom de la proposition logique associée à la transition.
- Les octets suivants comprennent les codes ASCII des lettres formant le nom de la proposition.
- Puis viennent deux octets, le premier contenant le nombre de places d'entrée de la transition et le deuxième le nombre de places de sortie.
- Ensuite viennent des octets comprenant d'abord les numéros des places d'entrées, puis les numéros des places de sortie.

Bien entendu au ième objet de la table correspond la ième transition du réseau. Il faut aussi des pointeurs de début et de fin de table, soit respectivement PDTTP et PFTTP et un pointeur courant PCTTP.

La figure III-4 représente la table TTP dans un cas général et la figure III-5 représente cette même table dans le cas du réseau de la figure III-1.

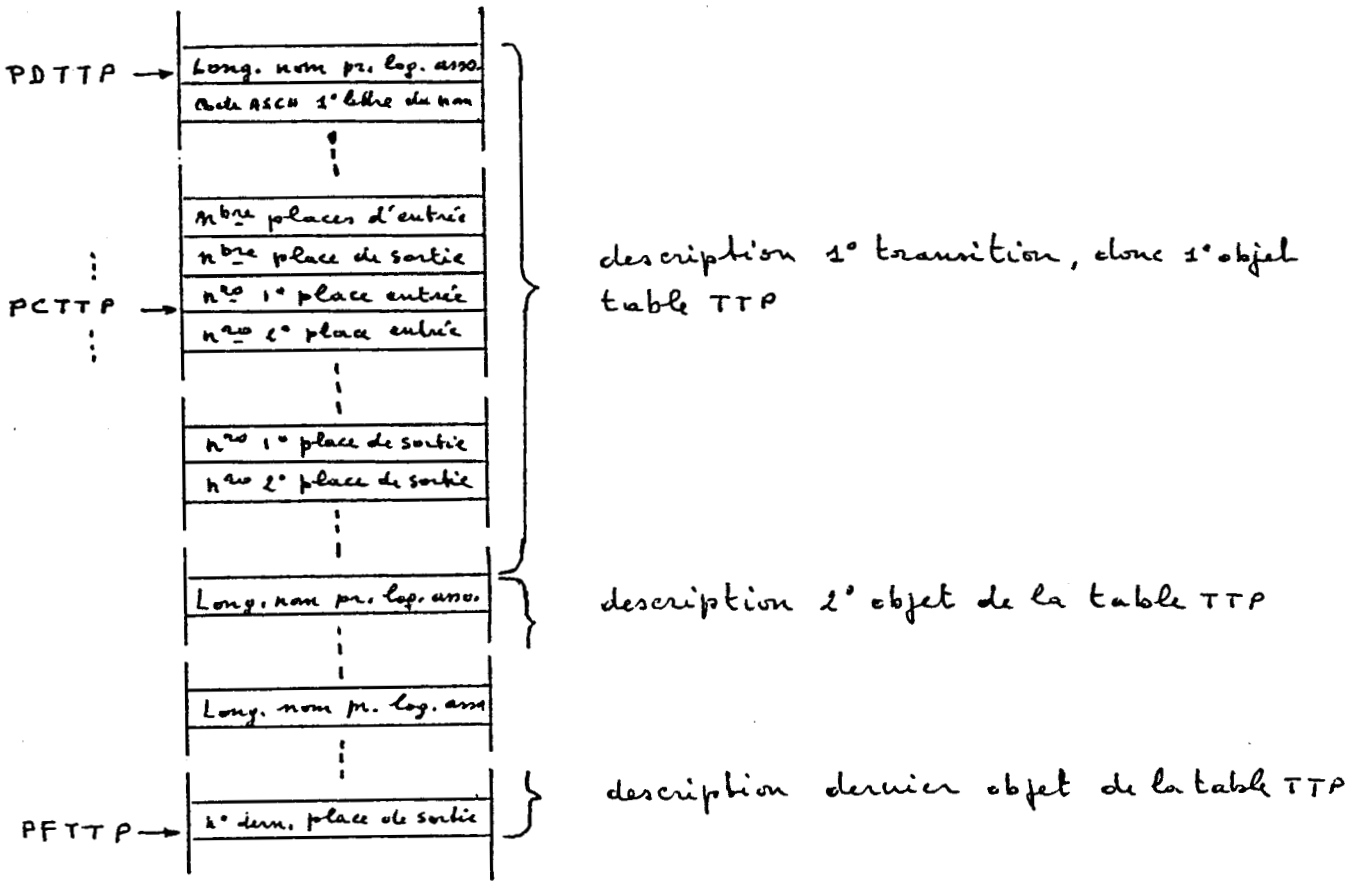


Fig. III-4

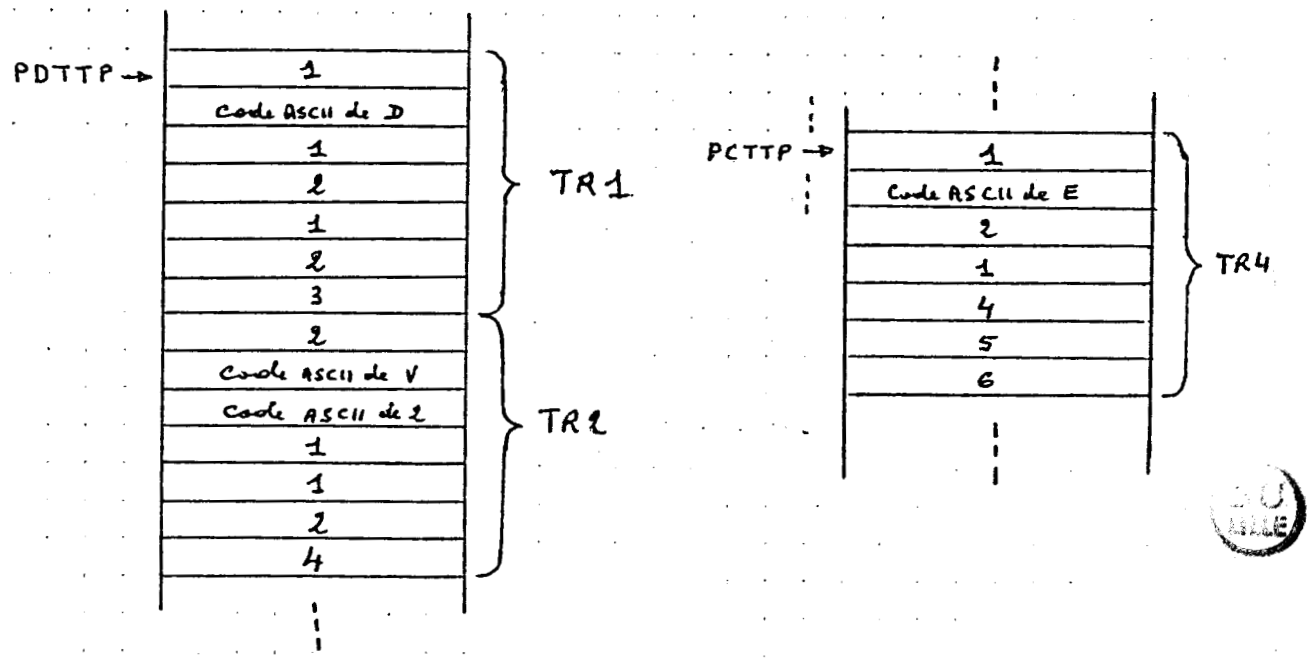


Fig. III-5

Cette configaration n'est pas suffisante pour permettre au programme échéancier de dresser la liste des tâches à effectuer. En effet il manque au moins les tâches associées aux places. De plus s'il est possible de recopier la zone Mo en Ram et d'y noter l'évolution du marquage ce qui donnerait les places marquées, il ne serait pas facile avec la seule aide supplémentaire de la table TTP de détecter les transitions validées.

Pour remédier à ce problème nous créons une table supplémentaire que nous appelons table places-transitions, soit TPT. Cette table donne la description de chaque place et spécifie les transitions en aval. Ici aussi le ième objet de la table coïncide avec la ième place du réseau.

Un objet de la table TPT est organisé de la manière suivante:

- Le premier octet indique la longueur du nom de la tâche associée à la place. Les octets suivant comportent les codes ASCII des lettres formant ce nom.
- Ensuite vient un octet indiquant le nombre de transitions en aval, suivi d'autant d'octets comprenant les numéros de ces transitions.

La figure III-6 représente la table TPT dans un cas général et la figure III-7 représente cette même table dans le cas du réseau de la figure III-1.

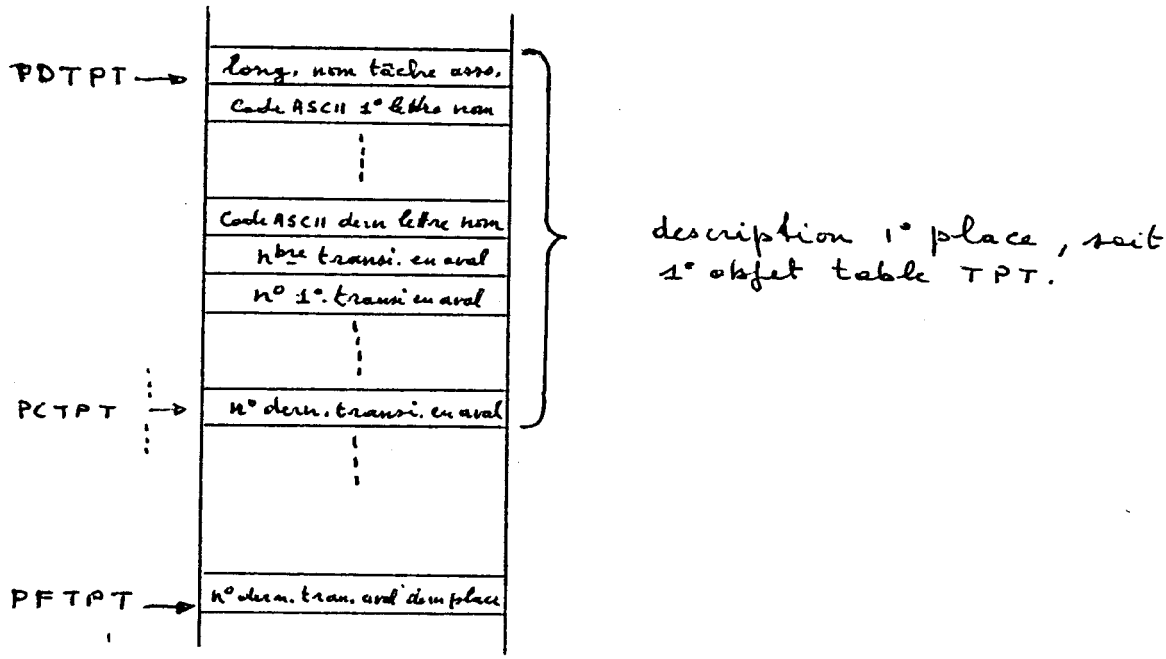


Fig. III-6

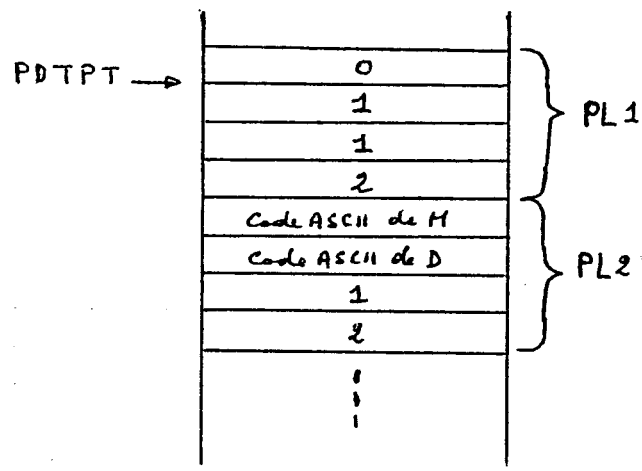


Fig. III-7

Comme nous le verrons au chapitre suivant lors de l'initialisation tous les processeurs se mettent en attente d'un travail a effectuer, le processeur hôte en plus se met à l'écoute du terminal, c'est à dire de l'opérateur. Lorsque ce dernier démarre une application il lance un programme qui recopie Mo en mémoire RAM et y constitue la table des places marquées M. Ensuite le programme échéancier est lancé, non pas à son point d'entrée normal mais en courcircuitant la phase d'actualisation de M. Il commence par établir les transitions validées puis dresse la liste des tâches à exécuter et enfin lance le programme ordonnanceur.

L'ensemble des tâches à exécuter le sont suivant la stratégie prévue au chapitre précédent. Lorsqu'un processeur exécutant une tâche vérifiant une proposition logique associée à une transition constate que celle-ci est vraie il lance à nouveau le programme échéancier mais en commençant cette fois par l'actualisation de M. Il termine de nouveau en retournant vers le programme ordonnanceur ce qui revient à prévenir les processeurs qu'il convient de s'attribuer une nouvelle tâche.

A ce sujet il peut y avoir un problème que nous allons analyser. La technique utilisée est de déposer dans un emplacement en mémoire commune un code convenu. Ce code est lu par chaque processeur à intervalle régulier à l'aide d'une interruption créée par une horloge interne. Si le code est reconnu le processeur va vérifier un autre emplacement en mémoire commune servant de compteur. Si ce dernier n'est pas à la valeur égale au nombre de processeurs moins une unité le compteur est incrémentè. Le dernier processeur efface le code et remet le compteur à zéro.

Le problème qui peut se poser est relatif au système d'arbitrage d'acces au bus commun. Si ce système ne garantit

pas l'impossibilité d'une interférence avec un autre processeur entre, par exemple, la vérification du compteur et son incrémentation il peut y avoir erreur.

Ainsi dans le cas de six processeurs, le compteur lu par le cinquième est à quatre, si le dernier processeur accède à la lecture du compteur avant son incrémentation à cinq, aucun processeur n'effacera le code ni ne fera la remise à zéro.

Pour éviter cette situation une autre solution peut-être envisagée qui consiste à faire rentrer chaque processeur dans une boucle d'attente. Cela de façon à donner aux autres processeurs le temps de lire le code à leur tour. Au sortir de la boucle ils vont chacun effacer le code et effectuer la remise à zéro du compteur avant de se lancer à nouveau dans l'attribution des tâches. La difficulté est l'évaluation la plus exacte possible du temps d'attente.

Cette question étant réglée il y a encore lieu d'établir un programme qui permette de transposer le texte édité donnant la description du réseau de Pétri, en Mo, TTP et TPT.

Nous ne détaillons pas outre mesure ce programme qui est simple comme nous allons le voir:

L'analyse de gauche à droite de la première phrase permet de créer Mo. Au départ il faut bien entendu déterminer PDMo en fonction du système. Le nom de l'application permet de créer un fichier des applications développées (directory) et de retrouver PDMo. La lecture du nombre de places permet de déterminer exactement la grandeur de la zone Mo, donc d'obtenir PFMo et le nombre qui lui est associé (nombre de bits significatif du dernier mot). La zone Mo est initialisée à Zéro. La lecture des numéros des places qui suivent permet de mettre à 1 les bits correspondant, ce qui termine la création de Mo (à la rencontre du mot " ; ").

Le nombre de places lu dans la première phrase permet aussi de déterminer le nombre des phrases suivantes qui vont permettre de constituer la table TPT. Chacune de ces phrases permet d'établir un objet de TPT. PDTPT peut être pris égal à PFMo + 1.

Pour créer un objet de TPT chaque phrase est lue de gauche à droite, le nom de la tâche associée est répéré entre les deux premiers espaces. Ce nom est lu deux fois, la première fois on compte le nombre de lettres qui le compose et que l'on inscrit, la deuxième fois on transcrit le code ASCII de ces lettres. Puis sont lu les numéros des transitions en aval de la place. Ces numéros sont lu deux fois également: une fois pour les compter et une fois pour les transcrire. Lorsque l'on à terminé avec toutes les phrases de ce type on a créé TPT et l'on connait PFTPT.

Il n'y a plus qu'a créer la table TTP avec les phrases restantes. PDTTP peut être égale à PFTPT + 1. Chaque phrase correspond à un objet de la table et sera lue de gauche à droite. De nouveau on repère le nom de la proposition logique associée par les deux premiers espaces. Il y a deux lectures, la première pour compter le nombre de lettres et la seconde pour transcrire leur code ASCII. puis le même travail se répète pour les numèros des places d'entrée précédés de PE et enfin pour les numéros des places de sortie précédés de PS.

III. 3. PROGRAMME ECHEANCIER.

Nous en sommes à présent au point de pouvoir étudier le programme échéancier dont nous avons déjà parlé. Nous allons commencer par rappeler ce qu'il doit réaliser.

Il devra actualiser M, donc déterminer les nouvelles places marquées, et préciser les nouvelles transitions validées.

Cela lui permet de définir la nouvelle liste des tâches à exécuter. Il lui reste alors à prévenir les processeurs qu'ils doivent s'attibuer une nouvelle tâche.

Au moment ou il est lancé ce programme connait l'adresse de la tâche correspondant à la proposition logique qui vient d'être affirmée. Cette adresse doit lui permettre de retrouver le numéro d'ordre de la transition correspondante dans TTP. Nous verrons comment ci-après.

Une fois ce numéro obtenu il est facile d'atteindre l'objet correspondant de la table TTP et d'y lire les numéros des places d'entrée et de sortie de la transition devant être tirée. Avec ces numéros il est facile d'actualiser M. Cela donne le nouveau marquage du réseau.

En parcourant alors M depuis le début on trouve les numéros des places marquées et pour chacune d'elles on lit l'objet correspondant de la table TPT. Cette lecture donne outre le nom de la tâche associée à la place marquée, ce qui permet de dresser la première partie de la liste des tâches à exécuter, le numéro des transitions en aval. Ces transitions sont candidates à la validation.

A partir d'ici deux procédures peuvent être retenues. Dans la première on établit directement la liste des transitions validées. Pour cela on prend chaque numéro des transitions en aval on lit dans TTP les numéros des places d'entrée et on vérifie dans M si elles sont toutes marquées. Dans l'affirmative on place le numéro de la transition en question dans la liste des transitions validées. Naturellement avant d'aller lire dans la table TTP on a vérifié si la transition examinée n'est pas déjà dans la liste des transitions validées. Ce procédé est répété pour toutes les transitions en aval d'une place marquée et pour toutes celles-ci.

Une autre procédure consiste à créer une liste intermédiaire des transitions candidates à la validation. Chaque objet de cette liste est composé de trois octets. Le premier contient le numéro de la transition, le second le nombre de places d'entrée et le troisième le nombre de places d'entrée marquées. Alors pour chacune des transitions en aval lue dans TPT on vérifie si elle se trouve déjà dans la liste intermédiaire, dans l'affirmative on incrémente le troisième octet correspondant. Dans la négative on crée un nouvel objet de la liste intermédiaire. Lorsque l'on a parcouru tous les éléments de M il faut trier la liste intermédiaire, c'est à dire ne garder que le premier octet des objets dont les deuxièmes et troisièmes sont égaux. Le résultat de ce tri correspond à la liste des transitions validées.

Une fois que l'on a la liste des transitions validées une simple lecture dans TTP permet d'identifier les noms des propositions logiques associées et d'établir ainsi la deuxième partie de la liste des tâches à exécuter.

Bien entendu il faut encore traduire cette liste de noms en une liste d'adresses. Il nous faut donc encore une table de conversion des noms de tâches en leur adresse de début. De même il nous faut une table de conversion des adresses des tâches correspondant aux propositions logiques en numéros des transitions associées.

Ces tables de conversions sont établies aisément au moment de la création des descripteurs de tâches.

Il reste au programme échéancier à prévenir les processeurs qu'ils doivent s'attribuer une nouvelle tâche, mais cette question a déjà étée discutée précédemment nous n'y reviendrons pas.

Un problème subsiste dans la mesure ou la procédure décrite fonctionne correctement dans le seul cas ou une proposition logique unique devient vraie à un instant donné. Or plusieurs propositions peuvent devenir vraies simultanément. Néanmoins les temps d'exécution des tâches correspondantes n'étant pas identiques, les résultats arrivent à des instants décalés et sur des processeurs différents.

Une solution peut être la suivante. Lors du constat d'une proposition vraie le processeur vérifie si le programme échéancier est en cours ou non au moyen d'un code en mémoire commune. Dans l'affirmative il émet éventuellement un message d'erreur. Sinon il vérifie une liste en mémoire commune. Cette liste donne les numéros des propositions logiques venant d'être reconnues vraies ainsi que le numéro du processeur ayant trouvé le résultat. Il inscrit son résultat et son numéro de processeur en fin de liste et entre dans une boucle d'attente. Au sortir de la boucle il revérifie la liste, s'il est toujours le dernier il place le code disant que le programme échéancier est en cours (après en avoir revérifier son absence) et lance ce dernier. Ce faisant il doit aussi réinitialiser la liste des propositions venant d'être reconnues vraies. Si au sortir de la boucle d'attente il constate qu'il n'est pas le dernier il attend à nouveau jusqu'au moment ou il est prévenu de devoir s'attribuer une nouvelle tâche.

Nous allons à présent voir le détail de la procédure correspondant au programme échéancier.

La première action que doit effectuer le programme échéancier est l'actualisation de M.

Pour cela le processeur qui vient de détecter qu'une proposition logique est vraie à mémorisé l'adresse de début de la

tâche associée. Il balaye alors la table de conversion adresses de tâches en numéros de propositions logiques. Il trouve ainsi le n° de la transition concernée. Ayant ce numéro il a donc aussi celui de l'objet de la table TTP qui décrit cette transition.

Le programme échéancier initialise alors le pointeur courant P de la table TTP à la valeur PDTTP (début de table) et parcours cette table rapidement jusqu'a l'objet concerné. Dès qu'il pointe cet objet il lit le nombre de places d'entrée qu'il mémorise, il fait de même pour le nombre de places de sortie. Il peut alors lire le numéro de chaque place d'entrée et mettre le bit correspondant de M à 0. De même il peut lire le numéro de chaque place de sortie et mettre le bit correspondant de M à 1. M est alors actualisée.

Il faut maintenant dresser la liste des tâches associées aux places marquées et détecter les transitions validées.

A cette fin M actualisée est balayée. Au départ le pointeur courant de M est placé à la valeur P = PDM, I donnant la position du bit lu est mis à 0 et un compteur associé I' est mis à 1. Pour chaque bit lu le compteur I' est incrémenté. Pour chaque bit rencontré à 1 on commence par sauter dans la table TPT à l'objet dont le numéro correspond à I'. Là on y lit le nom de la tâche associée à la place marquée et on le place dans la liste des tâches à exécuter. Puis on lit le nombre de transitions en aval que l'on mémorise en I". On lit ensuite le numéro de chacune des transitions en aval. Pour chaque lecture de numéro I" est décrémenté, lorsqu'il est égal à 0 c'est que l'on a terminé pour la place marquée considérée et l'on continue le balayage de M. Mais pour chaque lecture de numéro de transition aval on doit aussi faire les actions ci-après.

A l'aide du numéro on vérifie dans une liste temporaire s'il correspond à une transition déjà validée. Dans l'affirmative on passe à la lecture du numéro suivant. Dans la négative on se pointe sur l'objet de numéro correspondant de la table TTP on y lit les numéros des places d'entrée qui lui correspondent. Pour chaque place ainsi trouvée on vérifie si le bit d'ordre correspondant dans M est à 1. Si l'on trouve un zéro on a fini et on passe à la lecture de la transition aval suivante après avoir vérifier I". Si on trouve la dernière place d'entrée à 1 on met le nom de la transition dans la liste des noms de tâches à exécuter et son numéro dans la liste temporaire citée plus haut. On sait que l'on est à la dernière place d'entrée parce que l'on a lu leur nombre I" en commençant (il est reprit dans l'objet de TTP). Pour chaque place testée on décrémente I", dès que I" est égal à 1 c'est que l'on aborde la dernière place pour la transition considérée.

Il est évident également qu'avant de continuer le balayage de M on vérifie si son pointeur courant n'est pas égal à PFM. Dans l'affirmative c'est que l'on a terminé de dresser la liste des noms des tâches à exécuter.

A l'aide de la table de conversion "noms de tâches vers adresses de début de descripteurs correspondant" on dresse la liste finale en regroupant en tête les desripteurs des tâches associées aux places marquées qui ne doivent être exécutée qu'une seule fois.

N.B. Nous avons pris la première des deux procédures proposées pour trouver les transitions validées.

III. 4. LE PROBLEME DES TRANSITOIRES.

Ce problème correspond au fait qu'un nouveau marquage peut valider des transitions dont les propositions logiques sont directement vraies, sans nouvelles valeurs pour les données d'entrée. Cela entraine une nouvelle évolution du marquage pendant laquelle il faut assurer le maintient des valeurs de sortie. Pour tenir compte de cette éventualité il faudrait évaluer les propositions logiques nouvellement activées avant de relancer l'ordonnanceur tel que nous l'avons envisagé jusqu' ici (avec accès aux nouvelles valeurs d'entrée). Cette procédure ralentirait considérablement le système surtout s'il devait se succéder plusieurs marquages instables. Nous n'avons donc pas solutionné ce problème.

III. 5. CONCLUSIONS.

Nous avons vu comment introduire en machine la description d'une application et comment la transposer sous une forme exploitable lors de l'exécution. Nous avons mis au point le programme échéancier qui dresse les listes de tâches activées attendues par le programme ordonnanceur.

Au chapitre suivant nous allons intégrer l'ensemble de ce que nous avons vu et présenter une station locale multiprocesseurs répondant au cahier des charges ci-après:

Cette station est constituée sur le plan matériel de cartes essentiellement standards. Ces cartes comprennent une famille de processeurs dont un processeur hôte, auquel est rattaché un terminal pour les communications avec l'opérateur.

Elles comprennent également un arbitre de bus, une mémoire commune et un processeur pour l'acquisition des données. Sur le plan fonctionnel cette station doit pouvoir fonctionner même si un des processeurs tombe en panne. Le processeur hôte doit pouvoir apporter l'aide au développement de nouvelles applications, à leur lancement et à leur arret.

C H A P I T R E IV

P R O J E T D' U N E

S T A T I O N L O C A L E

M U L T I P R O C E S S E U R S.

IV. 1. INTRODUCTION.

Nous en sommes à présent au point de pouvoir présenter le projet d'une station locale multiprocesseurs en donnant d'une part le développement de la partie matérielle et d'autre part en complétant et en faisant la synthèse du logiciel correspondant. Nous commençons néanmoins par décrire certains projets similaires.

IV. 2. DESCRIPTION DE STATIONS SIMILAIRES.

a) Projet MARISIS (28)

Bien qu'assez éloigné de la notion de station telle que nous la concevons nous dirons un mot du projet français MARISIS. Il s'agit en fait d'un hypercalculateur qui fait intervenir deux types de parallèlisme dont nous avons déjà parlé au cours du premier chapitre: une machine SIMD et une machine MIMD.

Le projet MARISIS s'articule en trois volets. Le premier consiste en une machine de puissance intermédiaire, pouvant dépasser 100 mégaflops dans le modèle haut de gamme et qui porte le nom de ISIS.

ISIS est une machine adapté au traitement des vecteurs, de structure SIMD pouvant comporter de 8 à 64 processeurs élémentaires.

Le deuxième volet concerne l'étude d'une "machine à réseau d'interconnexion pour l'analyse numérique" à laquelle on a attribué le nom de MARIANNE.

Le troisième volet réalise la synthèse des deux systèmes précédents, consistant à intégrer un certain nombre de machines de base ISIS dans un système multiprocesseurs de type MARIANNE.

Le prototype d'ISIS et les premières maquettes de MARIANNE doivent sortir en 1986, le prototype de MARISIS est prévu pour 1988.

Notre projet se rapproche plus du volet ISIS mais étant à vocation différente ne peut se confiner dans la structure classique SIMD.

b) Projet SCEPTRE (29)

Ce projet français également, est un multiprocesseurs a couplage étroit dont l'exécutif a été développé ces dernières années. La liste des primitives disponibles donne une idée de ses possibilités:

- ACTION D'UNE TACHE SUR UNE AUTRE
  - . DEMARRER (tâche)
  - . SIGNALER (évènement, tâche)
  - . EFFACER (évènement)
  - . ARRETER (tâche)
  - . prédicat: ACTIVE (tâche)

- ACTION D'UNE TACHE SUR ELLE-MEME
  - . ATTENDRE (sémaphore)
  - . TERMINER

- GESTION DES REGIONS CRITIQUES
  - . ENTRER (région)
  - . SORTIR (région)

- GESTION DES RESSOURCES
  - . OBTENIR (ressource)
  - . ATTRIBUER (ressource)
  - . LIBERER (ressource)

- GESTION DES EVENEMENTS
  - . ATTENDRE, EFFACER, SIGNALER (évènement)
  - . prédicat : ARRIVE (évènement)

- GESTION DES QUEUES
  - . ENVOYER (élément, queue)
  - . RETIRER (élément, queue)
  - . prédicat : VIDE (queue), PLEINE (queue).

c) Projet DRM (25)

DRM est un projet PHILIPS conçu pour être un système entiérement distribué. Dans ce système la charge est partagée par un ensemble de processeurs relativement indépendants. La communication entre eux se fait sous forme de données ou d'instructions par

liaison V24 ou par le bus VME.

Le système est basé sur deux concepts: le Hama (pour Hardware machine) qui prend en compte le matériel et distribue l'intelligence; le Soma (pour Software machine) constitué par plusieurs tâches séquentielles communiquant entre elles par des "boites aux lettres".

En cas de panne d'un Hama, les Soma sont répartit vers les Hama restant.

Le système développé par PHILIPS est implanté sur carte VME/68000 et utilise le langage C. La première version a été disponible en août 1984.

IV. 3. DESCRIPTION MATERIELLE.

Nous avons déjà signalé dans les chapitres précédents que nous allons élaborer une station autour de cartes standards qui de plus sont essentiellement ce que la firme ABSY peut mettre à notre disposition. Nous utilisons ainsi des processeurs tournant autour du microprocesseur 68000 et basés sur le bus VME.

IV. 3. 1. Le système de développement.

Il s'agit de l'Exormacs et est utilisé pour l'écriture en assembleur, l'assemblage et le lien (link) des programmes qui sont ensuite transférés sur la station proprement dite.

IV. 3. 2. Les processeurs.

Ce sont les cartes standard "FORCE" (SYS 68 K/CPU-1) (13) modifiées pour pouvoir travailler en multiprocesseurs sous le contrôle de l'arbitre décrit ci-après et par l'intermédiaire du bus VME (23). La modification essentielle consiste à déconnecter l'arbitre propre des cartes CPU qui fonctionne suivant la technique du chainage (Daisy Chain). Le microprocesseur est un 68000 travaillant à 8 MHz.

La ROM contient un moniteur avec entre autre la possibilité d'écrire de petits programmes en assembleur, de désassembler, de transférer des blocs mémoires, ainsi que de pouvoir lire et sauver des blocs mémoires sur bande magnétique (l'enregistreur est à commandes électriques). La RAM va de $ 0800 à $ 20000.

Point de vue interfaçage ces cartes possèdent 3 ACIA et 1 PI/T

Le schéma fonctionnel correspond à la figure IV-1 ci-après.

IV. 3. 3. Le bus VME (23)

Ce bus se décompose en quatre parties:

- "data transfert bus" (bus pour le transfert de données).
- "priority interrupt" (priorité d'interruption).
- "D.T.B arbitration" (arbitrage sur le BTD)
- "Utility".

La première partie est en fait un bus système microprocesseur classique prévu pour 16 bits de données, extensible à

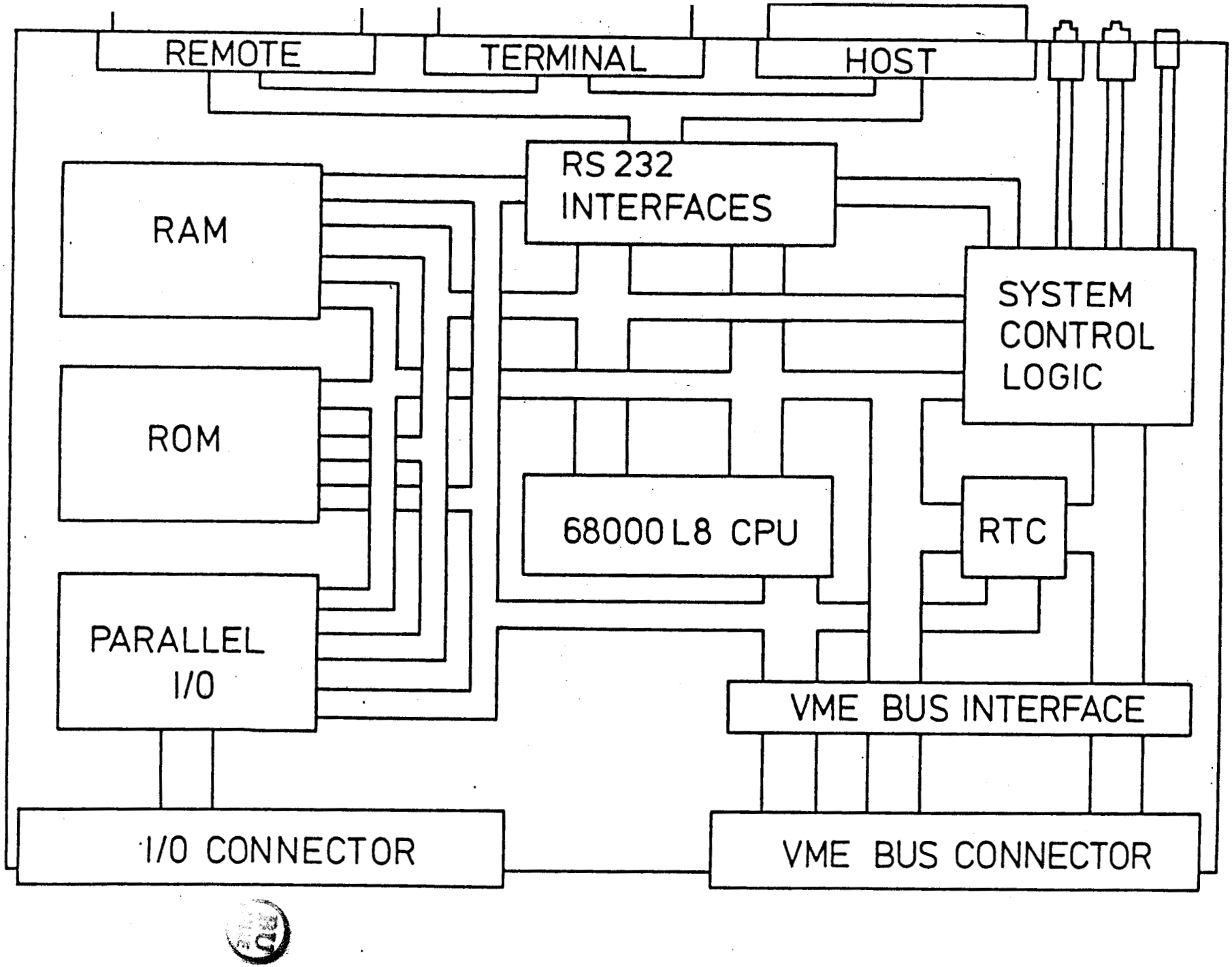

REMOTE
TERMINAL
HOST
RS 232
INTERFACES
RAM
ROM
PARALLEL
I/O
SYSTEM
CONTROL
LOGIC
68000L8 CPU
RTC
VME BUS INTERFACE
I/O CONNECTOR
VME BUS CONNECTOR

Fig. IV-1

32 bits, 24 bits d'adresses, également extensible à 32 bits et les lignes de contrôles.

La partie " utilities" correspond essentiellement aux lignes de test pour prévenir des défauts d'alimentation et de la vérification des initialisations.

La partie correspondant au "priority interrupt" permet la gestion des interruptions. Elle est constituée de sept lignes de demande d'interruption (interrupt Request). Celui ayant reçu la demande d'interruption et l'ayant acceptée envoie un signal de reconnaissance (Interrupt acknowledge) en le propageant par "Daisy-Chain" à chacun des demandeurs possibles. Lorsque le demandeur reçoit ce signal il dépose sur le "Data Transfert bus" un octet qui définit la routine d'interruption devant être effectuée.

Notons qu'il n'est pas possible à un processeur d'être demandeur d'interruption vis à vis d'un autre processeur.

La partie "D.T.B. arbitration" correspondant à l'arbitrage de l'acces au bus de transfert de données par les divers processeurs. Elle est composée de quatre lignes de demande d'acces au bus soit BR0 à BR3 (bus request), de quatre lignes allouant le bus soit BG0 à BG3 (bus grant) et d'une ligne indiquant que le bus est occupé soit BBusy. Notons qu'au dela de 4 processeurs il faut en principe utiliser la technique de "Daisy-Chain".

IV. 3. 4. L'Arbitre.

Dans l'esprit du bus VME il y a trois techniques d'arbitrage prévues:

a) hiérarchisation classique.

- Le niveau 4 est supérieur au 3 qui est lui même supérieur au niveau 2 et ainsi jusqu'au niveau 0. Cette hiérarchie par niveau étant complétée par "Daisy-Chain".
- Un demandeur de niveau supérieur oblige le détenteur du bus à le céder.

b) La technique dite de "Round Robin" : si le détenteur est de niveau "n" la priorité ultérieure est accordée au niveau "n-1"

c) Option unique: une seule ligne de demande de bus est utilisée (un seul niveau) et la priorité est fixée par "Daisy-Chain".

Selon l'esprit de la stratégie d'attribution d'une tâche que nous avons développé précédemment chaque processeur exécute une des tâches possible au même titre que son voisin. Nous ne devons donc pas hierarchiser les processeurs et devons donner le bus au premier qui le demande.

Nous avons alors réalisé une carte d'arbitrage semblable à celle développée à l'Ecole Royale Militaire à Bruxelles dans le service du Capitaine Timmermans.

Elle correspond au schéma bloc de la figure IV-2.

Le système est constitué par une horloge H qui attaque un compteur suivi d'un multiplexeur permettant de balayer successivement les demandes de bus éventuelles BR3 à BR0. Si la demande est présente au moment de la scrutation elle est mémorisée et réalise le signal d'acceptation correspondant (BG3 à BG0).

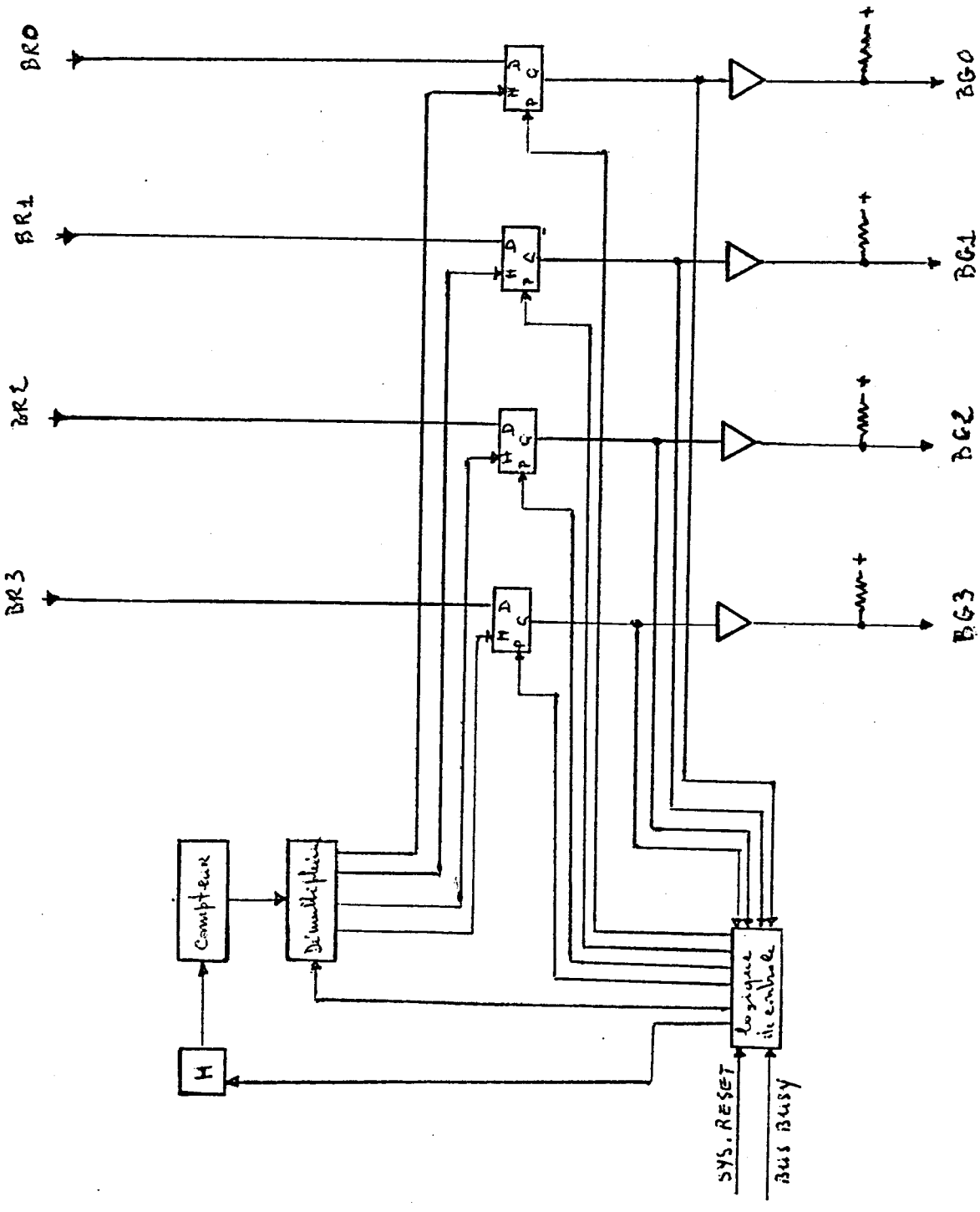


Fig. IV-2

La logique de contrôle inhibe la scrutation dès qu'une demande a été validée et tant que la ligne BBusy est active. De plus cette logique réinitialise également les bistables dès que la ligne BBusy est activée.

IV. 3. 5. La mémoire commune.

Cette mémoire n'a rien de particulier sinon qu'elle correspond à des cartes standard de 128 K bytes localisables à volonté dans l'espace adressable. Elle a été localisée à partir de l'adresse $ 100000. Les divers processeurs y accèdent via l'arbitre et peuvent ainsi communiquer entre eux.

IV. 3. 6. Le processeur de communication externe.

Ce processeur assure la communication entre la station multiprocesseur et d'autre stations du même genre ou un réseau.

Du coté de la station il communique par le bus VME en occupant une fenêtre mémoire de 128 bytes. Du coté stations extérieures ou réseau il communique au moyen d'USART et donc de liaisons séries.

Il est constitué autour d'un processeur 68121 et est à l'écoute de la station par l'intermédiaire de bytes sémaphores, prévient la station de l'arrivée de données par interruption et est averti par les USART de l'arrivée de données de l'extérieur. La communication avec l'extérieur (réseau ou autres stations) est controlé par un protocole ad-hoc.

Notons que les données en provenance de l'extérieur ou les résultats qui lui sont destinés occupent une zone mémoire commune au même titre que les informations locales.

IV. 3. 7. Schéma bloc fonctionnel de la station.

Ce schéma découle de ce que nous venons de dire et est représenté à la figure IV-3.

L'originalité de la station provient toutefois essentiellement de son logiciel qui est décrit ci-après.

IV. 4. DESCRIPTION DU LOGICIEL.

Le logiciel constitue certainement la partie essentielle du présent travail et a déjà été étudié en grande partie au long des chapitres II et III.

Il se répartit de la manière suivante:

a) Initialisation

b) Stratégie d'ordonnancement
   - Acquisition de données
   - Ecriture des résultats
   - Détection d'un évènement
   - Travail à effectuer ?
   - Processeur hôte ?
   - Demande Acces bus
   - Acces bus libéré
   - Attribution d'une tâche

c) Echéancier

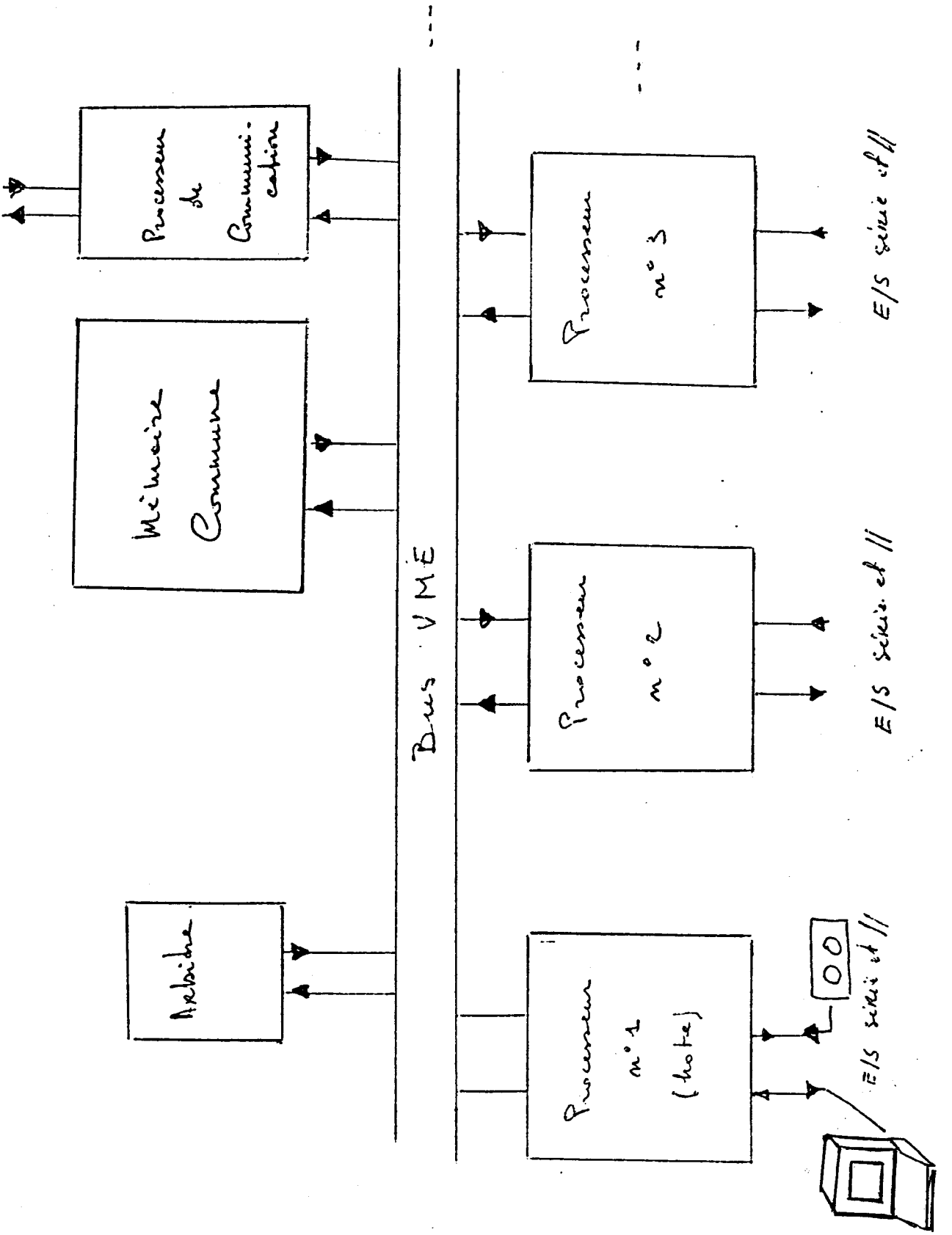

Processeur de Communication
Mémoire Commune
Arbitre
Bus VME
Processeur n°1 (hôte)
Processeur n°2
Processeur n°3
E/S série //
E/S série //
E/S série //

Fig. IV-3

d) Editeur-interpreteur

e) FORTH

f) Recherche des contraintes de précédence
   - cas général
   - cas des réceptivités.

g) Evaluation du nombre de processeurs nécessaires.

h) Calcul du temps d'exécution d'une tâche
   - temps moyen
   - dans le plus mauvais des cas.

i) Moniteur "Force" y compris la lecture et le chargement sur cassette.

Notons encore que le moniteur, l'initialisation, la stratégie d'ordonnancement et l'échéancier résident en PROM sur chaque processeur. Le FORTH est chargé sur chaque processeur a partir de la cassette via le processeur hôte et la mémoire commune. Les autres sont chargé en RAM du processeur hôte à partir de la cassette.

Parmis toux ces logiciels le seul important qui reste à étudier est l'initialisation.

IV. 4. 1. Programme d'initialisation.

Ce programme permet à la mise sous-tension ou lors d'une réinitialisation manuelle de préparer la station, c'est à dire l'ensemble des processeurs à exécuter un travail suivant la stratégie d'ordonnancement de tâches vue précédemment.

Tous les processeurs participent à égalité à ce travail commun, un seul étant distingué des autres; celui auquel

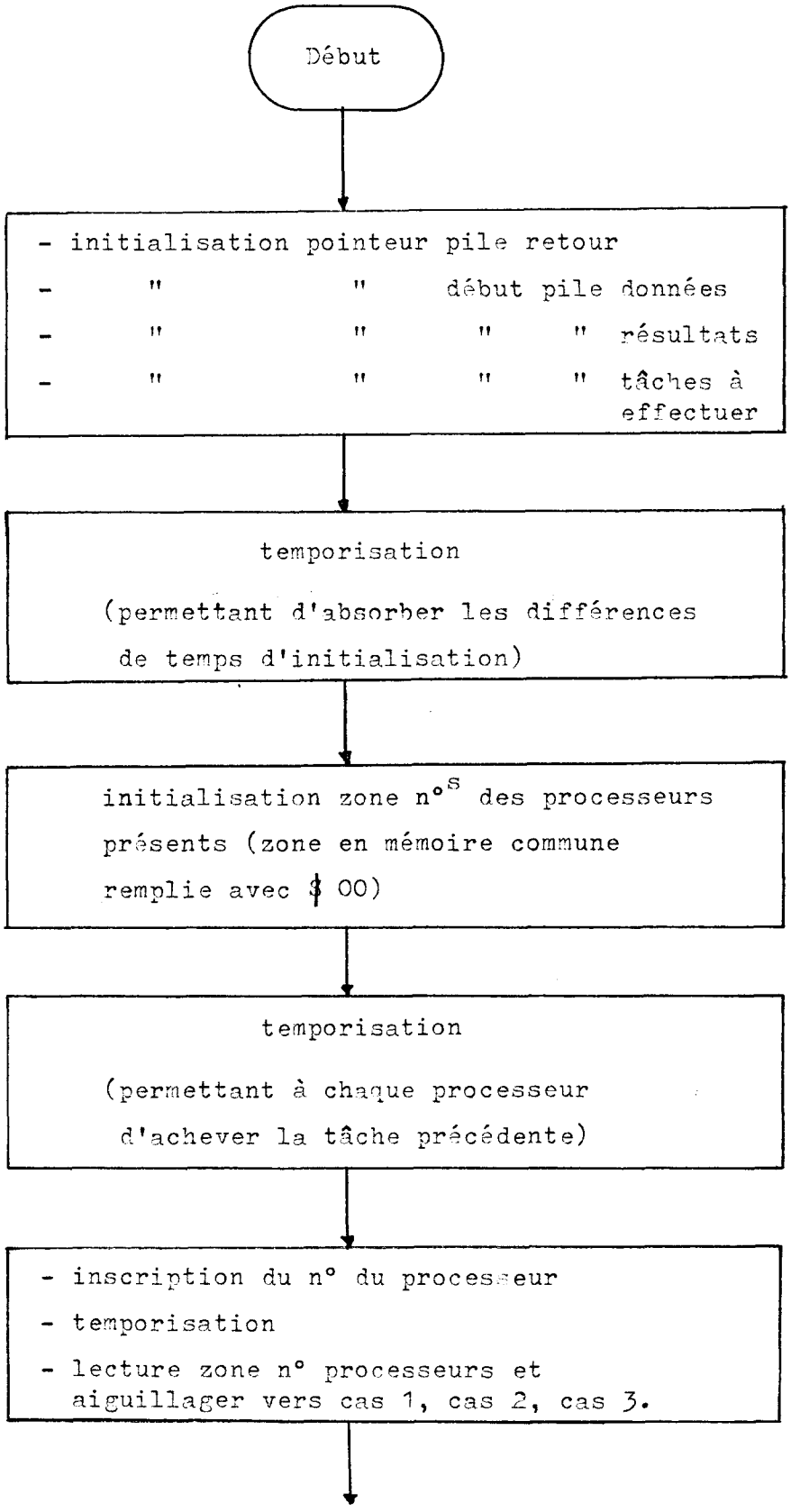
sera raccordé le terminal qui relie la station à l'opérateur et qui, comme nous l'avons déjà dit, porte le nom de processeur hôte. Ce dernier peut être retiré du pool de processeurs afin de développer ou de lancer un nouveau travail, de même ce processeur peut être réintégré au pool. Ces opérations peuvent être effectuées par l'opérateur à l'aide du terminal via l'ACIA. La stratégie d'ordonnancement montre que seul un ralentissement du travail intervient si l'on retire un processeur du pool, que ce soit l'hôte pour les raisons que nous venons d'évoquer ou tout autre processeur qui, par exemple, tomberait en panne.

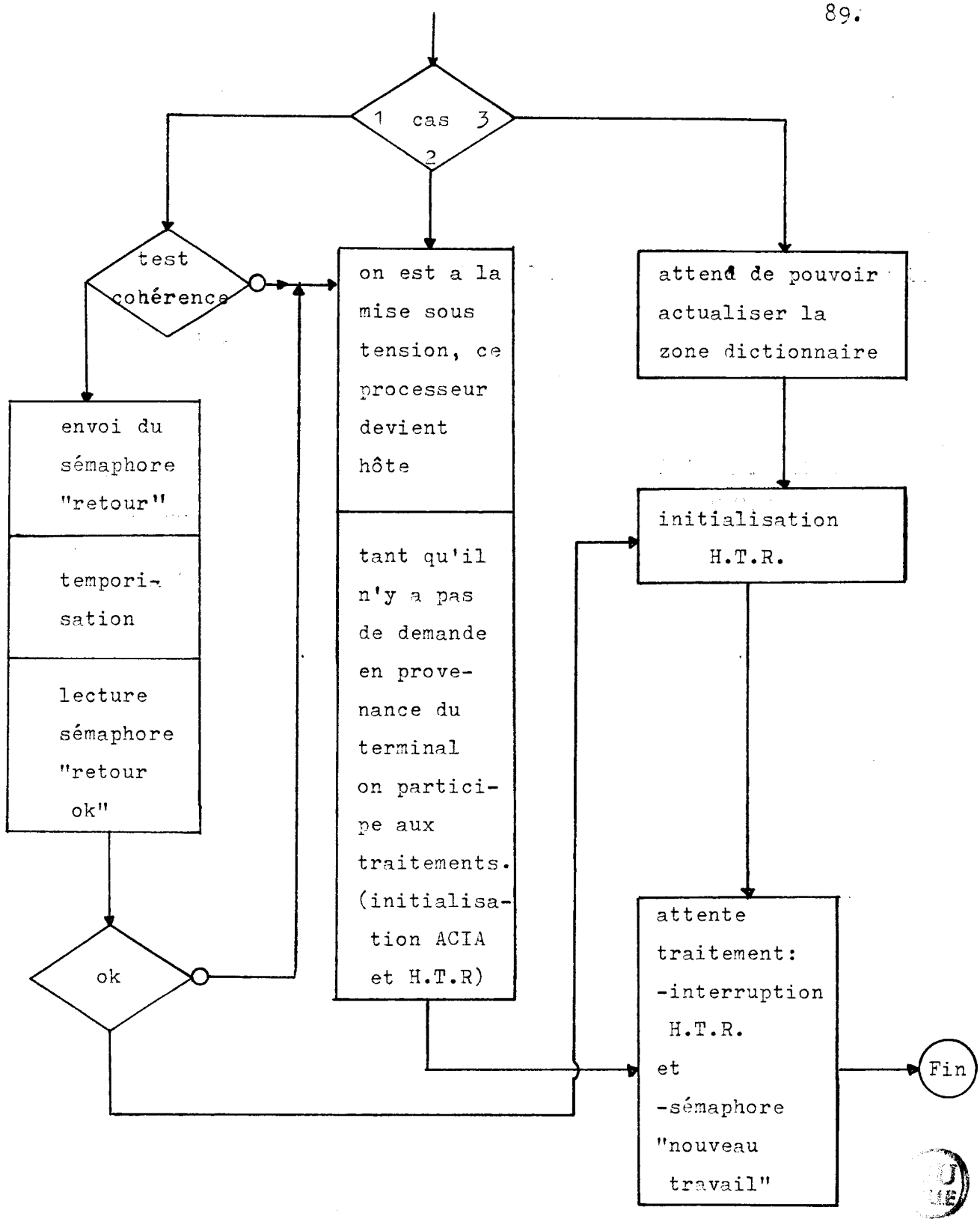
Une horloge temps réel (H.T.R.) est également initialisée et activée de manière à interrompre régulièrement chaque processeur afin de l'envoyer lire un sémaphore pour savoir si la tâche en cours est encore d'actualité. Cette tâche peut être en effet rendue caduque suite à l'intervention de l'échéancier ou au lancement d'un nouveau travail.

Lors de l'initialisation chaque processeur peut se trouver dans un des trois cas suivant:

- cas 1: il n'y a qu'un seul processeur (un seul n° de processeur inscrit dans la zone ad-hoc, celui du processeur initialisé).
- cas 2: il y a plusieurs processeurs initialisés et celui que l'on considère ici devient hôte (son n° étant le plus petit)
- cas 3: il y a plusieurs processeurs initialisés et celui que l'on considère ici n'est pas hôte. (son n° n'étant pas le plus petit).

Ce programme suit l'ordinogramme ci-après:


Début
- initialisation pointeur pile retour
- " " début pile données
- " " " " résultats
- " " " " tâches à effectuer
temporisation
(permettant d'absorber les différences de temps d'initialisation)
initialisation zone n°s des processeurs présents (zone en mémoire commune remplie avec $ 00)
temporisation
(permettant à chaque processeur d'achever la tâche précédente)
- inscription du n° du processeur
- temporisation
- lecture zone n° processeurs et aiguillager vers cas 1, cas 2, cas 3.


1 cas 3
2
test cohérence
on est a la mise sous tension, ce processeur devient hôte
attend de pouvoir actualiser la zone dictionnaire
envoi du sémaphore "retour"
temporisation
lecture sémaphore "retour ok"
initialisation H.T.R.
tant qu'il n'y a pas de demande en provenance du terminal on participe aux traitements. (initialisation ACIA et H.T.R)
ok
attente traitement: -interruption H.T.R. et -sémaphore "nouveau travail"
Fin

Notes: - 1 - Le test de cohérence correspond a vérifier l'existence d'une suite de nombres croissant régulièrement dans une zone de mémoire commune donnée.

- 2 - Dans le cas n°2 :
  - le processeur prépare le test de cohérence
  - le processeur prépare la zone des sémaphores.
  - le processeur place son jeton dans sa mémoire locale (pour se souvenir qu'il est processeur hôte)
  - initialise et active l'ACIA et l' H.T.R.
  - attente d'interruption par H.T.R. ou ACIA (s'il y a interruption par ACIA, H.T.R. est désactivée) (il y a une routine d'aiguillage des demandes d'interruptions qui différencie l'ACIA et H.T.R.)

IV. 5. MESURES ET RESULTATS.

IV. 5. 1. Temps d'exécution du programme d'attribution d'une tâche.

Diverses mesures effectuées en faisant précéder ce programme de la mise à 1 d'un bit du PI/T, en le faisant suivre de la mise à 0 de ce même bit et en bouclant, la tâche exécutée étant de longueur connue, ont permis d'évaluer le temps d'exécution à environs 100 μs.

Pour que le système multiprocesseurs soit efficace il faut donc travailler avec des tâches de longueur nettement supérieures à cette valeur.

IV. 5. 2. Gain en temps d'exécution d'un ensemble de tâches par un système multiprocesseurs.

a) La stratégie d'ordonnancement que nous avons présentée précédemment prévoit que lorsqu'un processeur arrive en fin de liste de tâches il n'attend pas les autres mais reboucle en début de liste avec un second jeu de valeurs. Cette technique rend difficile la mesure comparative du temps d'exécution de l'attribution et de l' exécution d'un ensemble de tâches par un ou plusieurs processeurs.

Il a dès lors fallu mettre en oeuvre l'ordinogramme suivant (figure IV-4) qui rend la stratégie quelque peu moins efficace mais donne des mesures stables: les mesures étant faites par visualisation à l'oscilloscope d'un bit du PI/T mis à 1 au début de cycle et remis à 0 en fin de cycle.

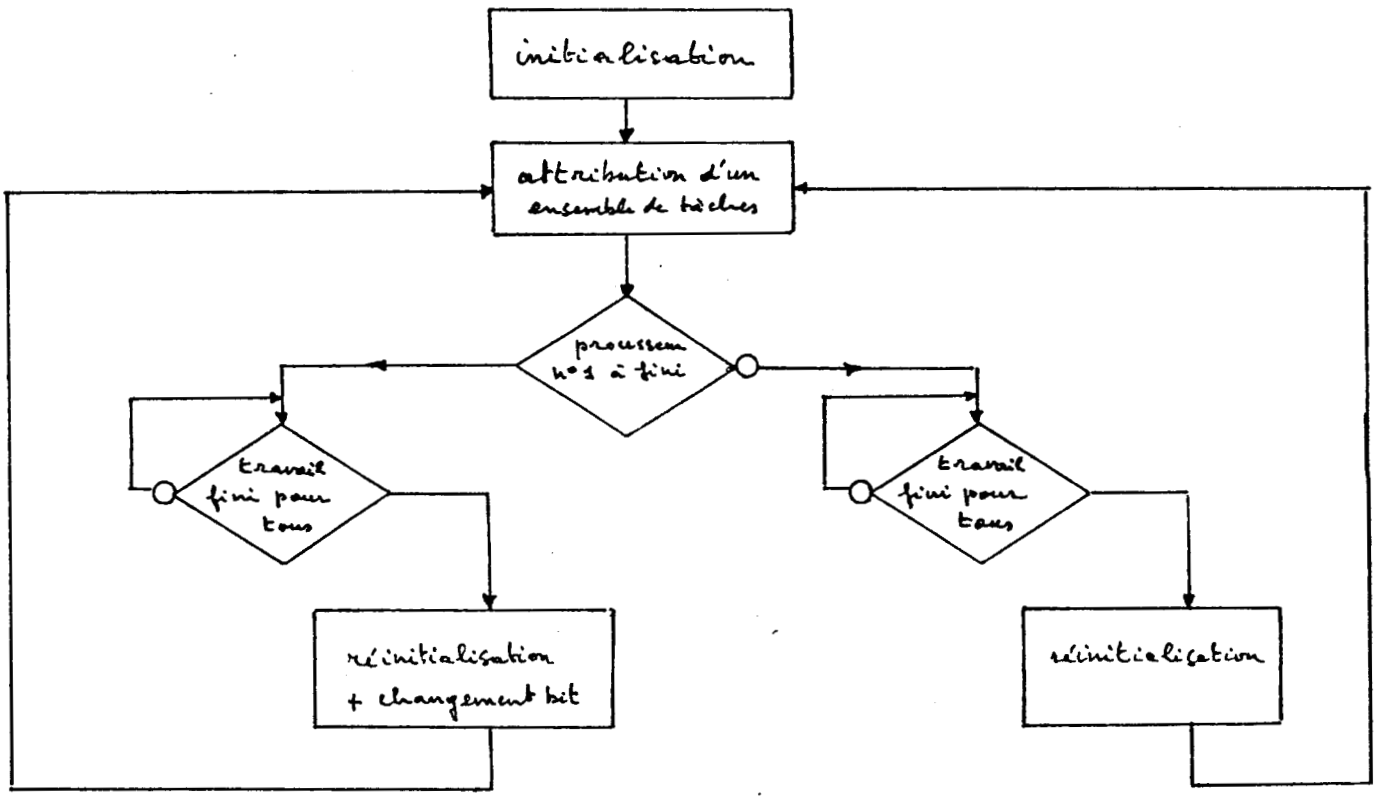


Fig. IV-4

b) Mots FORTH ayant servi pour créer l'ensemble de tâches utilisé pour la comparaison des temps d'exécution:

```
: mot1
    VAR11 Ø DO
            Ø 1 and drop
                                A Ø DO
                                    2 3 and drop
                            loop
                    loop ;

: mot2
    VAR21 Ø DO
            4 5 and drop
                                A Ø DO
                                    6 7 and drop
                            loop
                    loop ;

: mot3
    VAR31 Ø DO
            8 9 and drop
                                A Ø DO
                                    1Ø 11 and drop
                            loop
                    loop ;

: mot4
    VAR41 Ø DO
            12 13 and drop
                                A Ø DO
                                    14 15 and drop
                            loop
                    loop ;
```

avec A constante dont la valeur à été fixée à 99,
VA11, VAR21, VAR31 et VAR41 des variables permettant de régler le temps d'exécution pour chaque mot, leur valeur pour la mesure présentée ci-après à été fixée pour toute à ØØØF

Le tableau suivant résume les mesures faites:

ensemble de tâches exécutées	1 processeur	2 processeurs	3 processeurs
1 fois mot1 à mot4	320 msec 100%	170 msec 53%	140 msec 43,7%
2 fois mot1 à mot4	640 msec 100%	330 msec 52%	240 msec 37,5%
4 fois mot1 à mot4	1300 msec 100%	670 msec 52%	490 msec 37,5%

D'autres mesures avec d'autres mots Forth ont étées faites pour 1 et 2 processeurs donnant des résultats semblables (51 à 55%). Pour 3 processeurs seules les mesures présentées ont put être faites faute de pouvoir disposer plus longtemps d'une troisième carte.

Ces mesures semblent néanmoins satisfaisantes et, si l'on tient compte des contraintes auxquelles elles sont soumises qui réduisent l'efficacité du système, la technique proposée peut être considérée comme donnant des résultats fort proches de ce qui est théoriquement possible.

IV. 5. 3. Comparaison entre le temps d'exécution en FORTH et en Assembleur.

A la demande de la société ABSY nous avons créé en FORTH des mots pouvant tester des entrées (niveau du bit), réaliser des fonctions logiques simples sur ces entrées (par exemple ET/OU à n entrées, n étant quelconque) et appliquer le résultat sur une sortie.

La désignation d'une sortie ou d'une entrée devant se faire sous la forme I_{RCP} ou S_{RCP} avec R numéro de Rack, C numéro de la carte dans le Rack et P la position du bit sur la carte. Ces numéros devant être exprimés en décimal.

En réalité les numéros de Rack et de carte sont transformés en une adresse mémoire. Un masque permettant avec la position d'extraire ou de fournir la valeur du bit considéré.

Ce problème ayant été résolu un assembleur par la société ABSY on peut comparer les temps d'exécution. Le résultat des mesures a donné un rapport de l'ordre de 1,4 en faveur de l'assembleur.

Mots FORTH créés:

a) fonction ET à n entrées:

```
: ET ØØØØ ØØØ1 CALL ;        (appel d'une routine
                              assembleur réalisant la
                              fonction ET demandée)
```

b) initialisation:

```
: INIT ØØØØ ØØ1Ø CALL 1ØEØØØ1Ø CALL ;
```

c) création d'une entrée ou d'une sortie

1) HERE U. <CR> → xxxx

4 ALLOT <CR>

xxxx CONSTANT I_{RCB}
S_{RCB}

2) I_{RCB} / S_{RCB} } R C B ENTER <CR>

avec ENTER (I_{RCB} / S_{RCB} } R C B ...)

```
ØØ VARIABLE IA    (DECIMAL)
ØØ VARIABLE IB
: ENTER IA ! 2 * SWAP 256 * + IA @ 7 > +
        IB ! DUP IA @ SWAP ! 2 + IB @ SWAP ! ;
```

3) IN (I_{RCB} ...f)

```
ØØ VARIABLE IC              (HEX)
: IN 2 + @ 8 ØØØØØ . D!B ØØ44ØØ1Ø CALL 8ØØØØ2 . D@ .B
     IC @ PMASK + C @ AND Ø> ;
```

ou PMASK est créé comme suit

faire HEX <CR>

```
HERE U. → xxxx
Ø1 C,  <CR>
Ø2 C,   "
Ø4 C,   "
Ø8 C,   "
1Ø C,   "
2Ø C,   "
4Ø C,   "
8Ø C,  CR
```

xxxx CONSTANT PMASK <CR>

4) OUT (f S_{RCB} ...)

(HEX)

```
: OUT 2 + @ 8ØØØØØ . D!.B 8ØØØØ2 . D!B
ØØ2AØØ1Ø CALL ;
```

d) Mise à 1 et remise à 0 d'un bit du PI/T

```
: SETP 1Ø78 ØØ1Ø CALL ;
: RESP 1ØAC ØØ1Ø CALL ;
```

e) exemple de travail

```
: TRAV IØØØ IN IØØ1 IN ET SØ1Ø OUT ;
```

f) Bouclage pour mesure de temps

```
: DERN INIT BEGIN RESP TRAV SETP IØØØ IN = Ø UNTIL ;
```

IV. 5. 4. Gain en temps d'exécution du au "dépliage" d'un mot FORTH.

Le FORTH est un langage dont les mots sont écrit partiellement en FORTH et en assembleur pour le restant. Aussi un mot FORTH peut être défini comme une suite de CFA eux-mêmes défini en FORTH et ainsi de suite sur plusieurs niveaux.

Il est certain que ce "chaînage" fait perdre du temps lors de l'exécution de haut niveau.

Il semble donc intéressant de procéder a ce que nous nommons le "dépliage" des mots FORTH ou de certains en tous cas. Celui-ci ayant pour but de transcrire le mot FORTH considéré en une suite de CFA pointant uniquement sur des routines assembleur.

Une première technique permettant d'évaluer le temps gagné par chaque niveau de dépliage correspond aux mesures suivantes:

mots définis	temps d'une exécution	de dix exécutions
: p + ;	215 µsec	1,4 msec
: pl p ;	230 µsec	1,5 msec
: plu pl ;	240 µsec	1,64 msec
: plus plu ;	250 µsec	1,7 msec

Un problème lors des mesures est du au temps de rafraichissement des mémoires. Néanmoins on peut conclure que par niveau d'indirection la perte de temps est de l'ordre de 10 µsec, soit 5% pour un mot aussi simple que " + ".

Une seconde technique consiste à calculer les temps d'exécution de " : " et de " ; " à partir des instructions assembleur qui les composent.

	nombre de cycles horloge
: → move A2,-(A7)	10
move A1,A2	4
move (A2)+,A1	8
move (A1)+,A0	8
jmp (A0)	8

	nombre de cycles horloge
; → move (A7)+,A2	8
move (A6)+,A1	8
move (A1)+,A0	8
jmp (A0)	8

Soit au total 70 cycles à 8 MHZ ce qui donne approximativement 9 µsec.

Les deux techniques donnent des résultats sensiblement identiques, on peut compter sur environ 10 µsec.

IV. 5. CONCLUSIONS.

Nous avons décrit la station locale multiprocesseurs annoncée en montrant que sur le plan matériel, sauf en ce qui concerne l'arbitre, elle est construite à partir de cartes standards et du bus VME. Le bus VME intervient comme une limitation malgré tout, car étant donné la technique d'arbitrage, on est limité à quatre processeurs.

Le logiciel d'initialisation de la station a également été développé.

Les mesures faites montrent que l'ensemble matériel-logiciel ainsi choisi est satisfaisant: Le gain de temps de la stratégie d'ordonnancement tend vers le maximum pour traiter un travail sur plusieurs processeurs et l'utilisation du FORTH, tout en gardant les avantages qui lui sont propres et qui sont essentiellement ceux des langages de haut niveau permet, par la technique du "dépliage", de retrouver les vitesses d'exécution proche de l'assembleur.

C O N C L U S I O N S

G E N E R A L E S.

Nous avons pu mettre sur pied une station locale multiprocesseurs, incomplète peut-être, mais néanmoins suffisante que pour en montrer l'efficacité.

Naturellement il reste encore beaucoup à faire dans le domaine. Les études ou les réalisations sur ce sujet sont peu nombreuses.

En ce qui concerne notre étude il y a plusieurs points qui méritent d'être poussés plus avant encore.

Ainsi la première partie à développer est le problème du bus inter-processeurs. Nous avons été obligé vu les moyens mis à notre disposition, d'utiliser le bus VME. Nous avons vu que cela limite à quatre le nombre de processeurs pouvant travailler ensemble. Notons que diviser par deux le temps d'exécution d'un travail peut être déjà très précieux dans de nombreux cas, alors le diviser par quatre ou presque, n'est certes pas négligeable. Toutefois il nous semble qu'il serait heureux de pouvoir lever cette limitation. Cela pourrait sans doute se faire en adaptant la technique d'arbitrage d'acces au bus.

Un autre point à approfondir est l'interconnexion de plusieurs stations. Nous avons à peine eu l'occasion d'évoquer ce problème. Or un point important à étudier dans ce cadre est la datation des évènements.

Dans le cas du programme échéancier la table statique M pourrait probablement être remplacée avantageusement par une table dynamique contenant les numéros des places marquées.

Enfin le problème des évolutions transitoires, bien que difficile à régler de manière efficace, surtout dans le cas d'un système multiprocesseurs, devrait être envisagé et résolu.

A N N E X E II. 1.

Evaluation du nombre de processeurs souhaitables.

Ainsi, en rappelant que:

W = longueur réelle du séquencement.
Wo = longueur optimale du séquencement.
τ_i = temps d'exécution de la ième tâche.
m = nombre de processeurs utilisés.
n = nombre de tâches.

Alors, dans le cas d'un seul processeur on a évidemment:

$$W = Wo = \sum_{i=1}^{n} \tau_i$$

et l'on peut supposer, en première approximation tout au moins, que pour $m > 1$ l'on ait:

$$Wo1 \simeq \frac{\sum_{i=1}^{n} \tau_i}{m} \quad \text{(si m n'est pas trop élevé)}$$

Or, précédemment, nous avons vu que l'on pouvait obtenir une solution au problème du séquencement de n tâches sur m processeurs qui respecte la relation suivante:

$$\frac{W}{Wo} \leqslant 1 + (m-1) \frac{\max \tau_i}{\sum_{i=1}^{n} \tau_i}$$

avec max τ i = au plus grand de tous les τ_i d'où l'on peut déduire que:

$$\frac{W}{Wo} \leqslant 1 + (m-1) \quad \frac{\max \tau_i}{m.W'o1}$$

$$\text{soit } W \leqslant Wo + (m-1) \quad \frac{\max \tau_i}{m} \cdot \frac{Wo}{Wo1} \text{ OR } \frac{Wo}{Wo1} \geqslant 1$$

soit $Wo \geqslant Wo1$ c'est à dire $Wo = kWo1$ avec $k \geqslant 1$

$$\text{donc } W \leqslant k\, Wo1 + (m-1) \quad \frac{\max \tau_i}{m} \cdot k$$

Comme il faut avoir $W < d$, mais en cherchant à avoir $W \simeq d$ on pourra dire:

$$d \leqslant k \left(Wo1 + \frac{(m-1)}{m} \max \tau_i\right)$$

$$\text{ce qui donne : } m \geqslant \frac{k.\max \tau_i}{k\, W'o1 - d}$$

avec $W'o1 = Wo1 - \max \tau_i$ et $k \geqslant 1$ (valeur à déterminer expérimentalement), elle dépend de l'ordre de grandeur de m).

B I B L I O G R A P H I E .

1 - ABSY

"PPE-A, Peripheral Process Equipment for Acquisition & Automation"
ABSY, Bruxelles

2 - AHO A.V., J.E. HOPCROFT, J.D. ULLMAN

"The design and analysis of computer algorithms"
Addison Wesley 1974.

3 - ASHANY R.

"Current trends in Computer Architecture"
sept.13-17, 1982 Lausanne.

4 - BIDAINE E.R.

"Introduction aux Sémaphores"
Ecole Royale Militaire, Bruxelles 1981.

5 - BLANCHARD M.

"Comprendre, maîtriser et appliquer le Grafcet"
CEPADUES Editions, 1979.

6 - BRODIE L.

"Starting Forth"
PRENTICE -HALL, INC., ENGLEWOOD CLIFFS. 1981.

7 - CHU W. , HOLLOWAYS L.J., MIN-TSUNG LAN and KEAM EFA

"Task Allocation in Distributed Data Processing"
Computer November 1981.

16 - HOCKNEY R.W. & JESSHOPE C.R.

"Parallel Computers"
Architecture, Programming and Algorithms
Adam Hilger Ltd., Bristol 1981.

17 - INTEL

"Intel Multibus specification"
ORDER NUMBER: 9800683-04

18 - KAPOSI A.A.

"An engineer's guide to algorithmic structures"
Computer Aided Design Volume 9, nb 1, Jan. 1977.

19 - KIRRMANN H.

"Les Multiprocesseurs à couplage étroit"
Cours Postgrade en informatique technique,
Architecture des systèmes informatiques nouveaux
Ecole Polytechnique Fédérale de Lausanne, Avril 1982.

20 - MAHL R. et BROUSSARD J.C.

"Algorithme et Structures de données"
Laboratoire d'informatique - Université de Nice 1977.

21 - MOALLA M. & DAVID R.

"Extension du GRAFCET pour le représentation de
systèmes temps réel complexes"
RAIRO automatique/ Systems Analysis and Control
vol. 15, n°2, 1981.

22 - MOTORALA

"MC 68000 16-BIT Microprocesseur User's Manual"

23 - MOTOROLA

"VME bus Specification Manual"

8 - COFFMAN E.G.

"Computer and Job-Shop Scheduling"
J. Willey 1976.

9 - COURTOIS P.

"Le Concept FORTH, langage et système"
Editest 1983.

10 - DORNIC H.

"Un système temps réel distribué multiprocesseurs sur carte VME/68000"
Minis et Micro n° 210.

11 - ECOLE DE L'I.R.I.A.

"Le traitement parallele" Support du cours du 29 Mai au 2 Juin 1978 , volume 1 et 2.

12 - ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

"Traité d'électricité" Volume V: "Analyse et synthèse des systèmes logiques".

13 - FORCE

"SYS68K/CPU-1 User's Manual"
FORCE Jan. 1983.

14 - FORCE

"SYS68K Monitor Firmware User's Manual"
FORCE Jan. 1983.

15 - FOUREAU A.

"Séminaire d'analyse et de programmation structurées"
S.A. COMSYS , Bruxelles 01.82

24 - NATIONAL SEMICONDUCTOR CORP.

"MM57109 MOS/LSI Number-Oriented Microprocessor"
MARCH 1977.

25 - PHILIPS

"A Truly Distributed Real-Time Multiprocessing System for Microprocessors"
News Release (Philips)
Number: 9568E/ Sep/ 84-100

26 - PINAUD A.

"Programmer en FORTH"
Editions du P.S.I. 1983.

27 - QUINTANA J.M. PRO

"Analyse et mise en oeuvre d'un protocole de conversation pour un réseau multiprocesseurs à reconfiguration dynamique"
Thèse de Docteur-Ingénieur sept. 1979.
Université de technologie de Compiègne.

28 - REMY CL.

"Marisis Supercalculateur Français"
Micro-Systèmes Juillet-Aout 1984.

29 - SCEPTRE

"Proposition de standard de noyau d'exécutif temps Réel"
Rapport BNI n° 26/2 (1982)
Domaine de Voluceau Rocquencourt.

30 - SERIR A.

"Une station multiprocesseurs pour la conduite décentralisée de procédés industriels, Application à la commande continue et séquentielle"
Thèse de Docteur-Ingénieur, Institut Polytechnique de Grenoble, Avril 1983.

31 - SLOWINSKI R.

"L'ordonnancement des tâches préemptives sur les processeurs indépendants en présence de ressources supplémentaires"
RAIRO Informatique/ Computer Sciences
Vol. 15, n°2, 1981.

32 - THAYSE A.

"Application de la théorie des fonctions booléennes à la transformation de programmes".
Manuscript M9
Philips Research Laboratory, May 1981.

33 - THELLIERS S. et TOULOTTE J.M.

"Grafcet et Logique Industrielle Programmée"
Ed. EYROLLES 1980.

34 - THELLIERS S. et TOULOTTE J.M.

"Applications industrielles du Grafcet"
Ed. EYROLLES

35 - TRAN VAN KHAI

"Réseaux de processeurs temps réel. Modélisation analytique"
Revue Technique THOMSON - CSF Vol. 13, n°2, juin 1981.

36 - TSE - YUN FENG

"A Survey of Interconnection Networks"
Computer (IEEE) December 1981.

37 - VADIM E. KOTOV

"On Basic Parallel Language"
INFORMATION PROCESSING 80, S.H. Lavington (ed.)
NORTH-HOLLAND Publishing Company
IFIP, 1980.

38 - WIRTH N.

"Algorithms + Data Structures = Programs"
Prentice-Hall, Inc.

39 - YONG B.

"Définition et Simulation d'une unité de commande
pour processus simultanés"
Thèse d'Université, Université de Lille 1, Février 1982.