SYNDEX un environnement de programmation pour multi-processeur de traitement du signal. Mecanismes de communication

Nadia Ghezal, Spiros Matiatos, Pascal Piovesan, Yves Sorel, Michel Sorine

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1236

Programme 5
Automatique, Productique,
Traitement du Signal et des Données

SYNDEX
UN ENVIRONNEMENT DE PROGRAMMATION POUR MULTI-PROCESSEUR DE TRAITEMENT DU SIGNAL
MECANISMES DE COMMUNICATION

Nadia GHEZAL
Spiros MATIATOS
Pascal PIOVESAN
Yves SOREL
Michel SORINE

Juin 1990

RR_1236

SYNDEX
UN ENVIRONNEMENT DE PROGRAMMATION POUR MULTI-PROCESSEUR DE TRAITEMENT DU SIGNAL

MECANISMES DE COMMUNICATION

Nadia GHEZAL - Spiros MATIATOS - Pascal PIOVESAN - Yves SOREL - Michel SORINE

sorel@seti.inria.fr

INRIA Domaine de Voluceau Rocquencourt B.P.105 - 78153 Le Chesnay Cedex

Résumé

SYNDEX (acronyme pour EXécutif Distribué SYNchrone) est un environnement graphique interactif de développement d'applications de traitement du signal pour machines multi-processeur.

Nous présentons en introduction, les fonctions assurées par SYNDEX adaptées à la mise en oeuvre sur machines multiprocesseurs de programmes synchrones écrits dans le langage SIGNAL, quelques méthodes d'optimisation associées, la génération et le réglage du support d'exécution. Enfin nous décrivons rapidement le prototype existant de SYNDEX.

La partie principale du papier est destinée à définir les mécanismes de communication utilisés pour la génération automatique du support d'exécution temps réel des applications. Un noyau d'exécutif adapté aux multiprocesseurs visés est décrit.

Enfin nous donnons quelques mesures de performance des exécutifs générés dans le cas de machines multitransputer, ainsi que quelques exemples d'applications.

* Etude financée dans le cadre de la convention avec le CNET
n° 2.88.A009.000.00.MC.01.1

SYNDEX
PROGRAMMING ENVIRONNEMENT FOR SIGNAL PROCESSING MULTI-PROCESSOR

COMMUNICATION MECHANISMS

Nadia GHEZAL - Spiros MATIATOS - Pascal PIOVESAN - Yves SOREL - Michel SORINE

sorel@seti.inria.fr

INRIA Domaine de Voluceau Rocquencourt B.P.105 - 78153 Le Chesnay Cedex

Abstract

SYNDEX (acronym for SYNchronous Distributed EXecutive) is an interactive graphical development environment for signal processing applications on multi-processor machines.

First, we introduce the features of SYNDEX, oriented towards the implementation of synchronous programs written with the language SIGNAL on a multiprocessor, some associated optimization methods and the generation and tuning of real time executives. Then, we describe briefly the existing prototype of SYNDEX.

The main section of the paper is devoted to the communication mechanisms used for the automatic generation of the executive. An executive kernel fitted to the aimed multiprocessors is described.

Finally, we give some performance measures for the generated executive on a multitranputer and some application examples.

SOMMAIRE

I Principes généraux

1 - Les fonctions assurées par SYNDEX

L'environnement SYNDEX est conçu pour permettre à un programmeur d'application de traitement du signal, s'exprimant dans le langage SIGNAL[1], de bénéficier des services suivants :

- Dimensionnement de la machine cible, éventuellement multiprocesseur pour un programme d'application donné.
- Génération et réglage du support d'exécution de ce programme sur la machine obtenue après dimensionnement (ou choix à priori).

Les informations utilisées pour arriver à ces dimensionnements et réglages sont :

. Le programme SIGNAL de l'application (sous la forme d'un graphe d'exécution flot de données conditionné).

. Une estimation à priori des durées d'exécution des "grains" séquentiels (ou activités de calcul) composant l'application et des durées des communications entre ces grains. Ces durées sont liées aux performances des processeurs et moyens de communication utilisables.

A partir de ces informations, "l'optimiseur" de SYNDEX dimensionne la machine (calcul du nombre de processeurs si la machine est libre, capacité de la mémoire application, routes et leur buffers) puis il détermine un placement et un ordonnancement du programme d'application sur les processeurs et la mémoire application et des communications interprocesseurs sur les routes. SYNDEX permet de générer un simulateur fonctionnel de la machine cible programmée, où le programme d'application de départ a été réparti et le programme système (support d'exécution) a été généré à partir d'un noyau d'exécutif que nous avons cherché à réduire au maximum. Les seuls services retenus sont :

1) la gestion (création, allocation du temps d'utilisation) des ressources suivantes :
   - processeurs
   - mémoire des processeurs
   - moyens de communication internes à la machine : bus, liaisons biprivées (liens, route interprocesseurs) et avec l'extérieur.
2) la gestion du temps :
   - temps physique (dates et durées) pour l'évaluation de performances
   - temps logique (dates) pour l'interprétation des actions synchrones de SIGNAL.

Ces services sont suffisants pour supporter l'exécution des programmes SIGNAL sur les multiprocesseurs envisagés. Les services classiques (gestion de fichiers ...) sont rendus par un calculateur hôte.

Les dimensionnements et réglages peuvent se faire par itération : les informations a priori utilisées (le programme et les durées d'exécution des grains de calculs) peuvent être rafinées par réécriture du programme (prise en compte de la répartition) par correction des mesures à l'aide du simulateur généré (prise en compte de l'over-head introduit par les mécanismes de communication).

La figure 1 illustre l'intégration des environnements SIGNAL et SYNDEX ainsi que les services de SYNDEX.

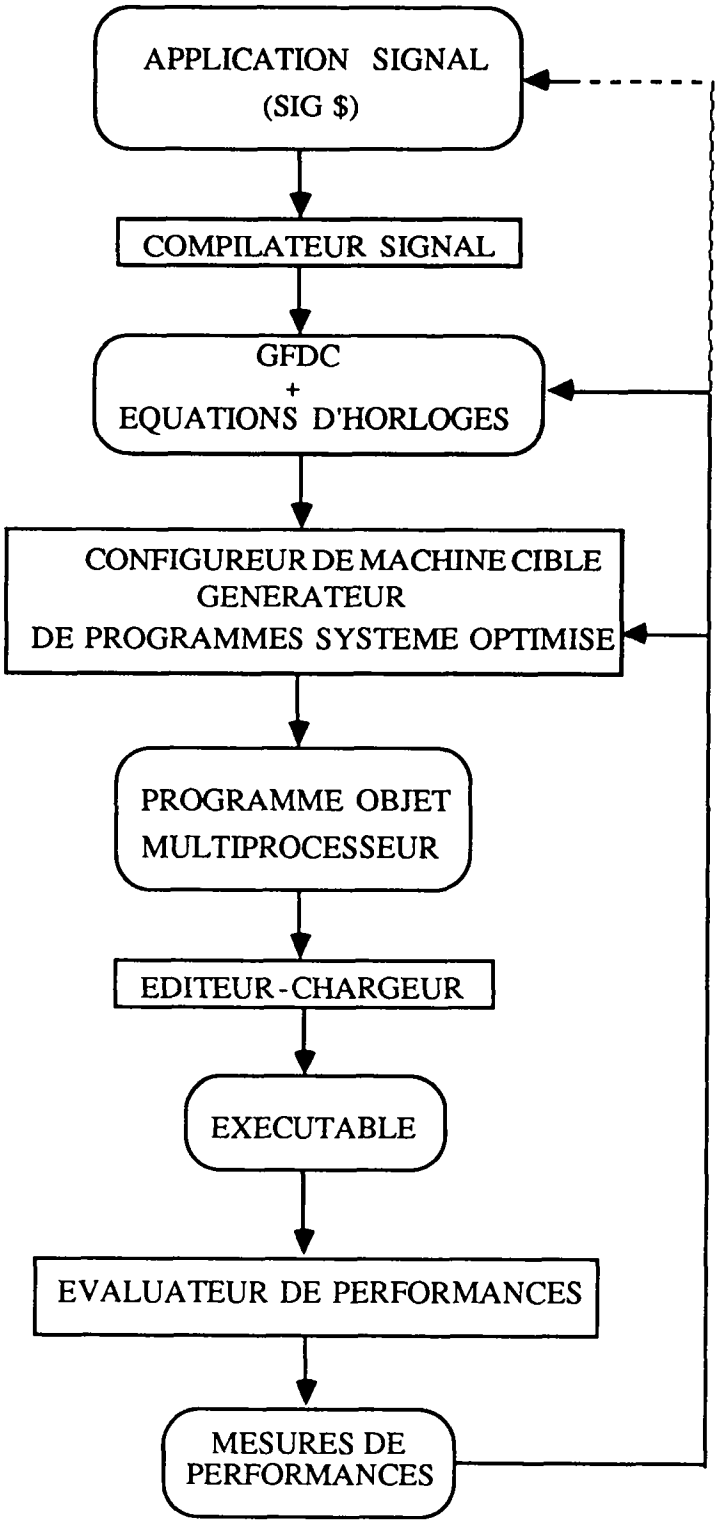

Fig. 1 Environnement SIGNAL SYNDEX

2 - Programmes synchrones et mise en oeuvre asynchrone

En traitement du signal ou en commande on écrit des applications modélisées par un processus dont le pseudo code SIGNAL est :

```
        PROG{ ?U !X}
        (|
(1)     | X' := G(X, U)
        | X := $X'
        |)
```

où U, X, X' sont respectivement les vecteurs d'entrée, d'état et d'état "suivant" d'un processus. Chaque composante de ces vecteurs est un signal représenté par un flot de valeurs et une horloge.

Remarques :

1) Les signaux ont en général des horloges différentes : un vecteur de signaux (U ou X par exemple) n'est pas un signal, les composantes ne sont pas présentes/absentes aux mêmes instants.

On notera $x(t) = \perp$ si le signal x est absent à l'instant t, $^{\top}$ s'il est présent et non valué (cas d'une horloge par exemple)

2) En général on peut vouloir masquer certains signaux et n'émettre que les autres ce qui revient à ajouter une équation de sortie $Y = H(U, X)$ et le profil de PROG devient { ?U !Y}.

Donnons maintenant un sens plus précis à (1) en utilisant le formalisme de SIGNAL, cf [1] et sa bibliographie et détaillons les étapes de sa mise en oeuvre "asynchrone".

A) Les horloges de signaux s'expriment en fonction d'autres horloges et de signaux booléens, composantes de U ou de X. Notons B(V) le vecteur des signaux booléens composantes du vecteur de signaux V alors (1) se réécrit de la façon suivante :

```
        PRÕG{ ?U !X}
        (|
        | X' := G̃(X, Ũ)
(2)     | X̃ := $X'
        | Ũ := (U, B(X))
        |)
```

où :

$\tilde{G}$ est une fonction de SIGNAL sans mémoire (construite uniquement avec des fonctions immédiates, des "when", des "default")

$X := \$X' \Leftrightarrow X_i := \$X'_i$, $i = 1, ..., N$, avec X_i composantes isochrones de X, c'est à dire X_i est un signal vectoriel et X_i et X_j ont a priori des horloges différentes si $i \neq j$.

$\tilde{U}$ est le vecteur obtenu en concaténant U et B(X).

B) La compilation du programme SIGNAL (2) conduit à :

B.1) Une expression des horloges des signaux produits X à partir, des signaux de l'entrée augmentée Ũ et d'un vecteur d'horloges libres, Φ.

Plus précisément notons pour un signal x :

$P(x) = {}^{\top}$ when $(x \neq \perp, ^{\top})$, horloge des instants de présence de valeurs du signal x

$T(x) = {}^{\top}$ when (x = vrai) pour x booléen (c'est à dire $\forall$ t , $x(t) \in \{\perp$, vrai, faux$\}$) et $= \perp$ pour x non booléen, horloge des instants "vrais" de x.

$S(x) = {}^{\top}$ when $(x \neq \perp)$, horloge des instants de présence de synchro ou de valeur dans x

On notera pour $X = (x1,...xn)$, $P(X)$ (resp. $T(X)$) le vecteur $(P(x_1)...P(x_n))$ (resp. $(T(x_1)...T(x_n))$)

Les horloges P(X) vérifient alors des relations du type :

(3) $$P(X_i) = h_i\,(P(\tilde{U}), T(\tilde{U}), \Phi), i = 1, ..., N.$$

où les h_i sont des expressions booléennes sur l'algèbre $\{{}^{\top}, \perp\}$ isomorphe à {vrai, faux}.

Remarquons que les h_i dans (3) ne sont pas uniques par exemple si $P(X_{io}) = h_{io}(.,.,.) = h'_{io}(.,.,.)$ alors la dernière égalité peut être considérée comme une contrainte sur le contexte du programme.

Choisissons donc une relation de type (3) par horloge et constituons le programme :

PROG-TIME.BASE{ ?$\tilde{U}$, Φ !H_1,..H_N}
(|
(4) | H_i := $h_i(P(\tilde{U}), T(\tilde{U}), \Phi), i = 1, ..., N$
|)

B.2) Une expression des signaux X qui peut s'écrire ainsi, en notant (voir [2]) :

$x_0 \otimes x_1 \equiv (x_0$ when $(S(x_1) = {}^{\top})$ default x_1

(remarquons que $S(x_0 \otimes x_1) = S(x_1)$, $P(x_0 \otimes x_1) = (P(x_0)+P(x_1)).S(x_1)$)

PROGi{ ?U, X, H_i !X_i'}, i = 1, ..., N
(|
(5) | $X'_i := G_i(X \otimes H_i, U \otimes H_i) \otimes H_i$
|)

La version compilée de (2) a donc la forme :

PROG-OBJ{ ?U, Φ !X}
(|
(6) | PROG-TIME.BASE{ ?$\tilde{U}$, Φ !H_1,..H_N}
| PROGi{ ?U, X, H_i !X'_i}, i = 1, ..., N
| X' := (X'_1, ..X'_N)
| X := $X'
| $\tilde{U}$:= (U, B(X))
|)

C Distribution du programme

C.1) La granulation (cf [3]) des composantes isochrones PROG i conduit à :

PROGi{ ?U, X, H_i !X'_i}, i = 1, ..., N
(|
(7) | PROGij{ ?U, X_i, H_i !X'_{ij}} j = 1, ... g_i
| $X'_i = (X'_{i1},...X'_{igi})$
|)

avec :

PROGij{ ?U, X_i, H_i !X'_{ij}}
(|
(8) | $X'_{ij} := G_{ij}(U \otimes H_i, X_i \otimes H_i) \otimes H_i$
|)

où

l'évaluation de X'_{ij} peut se séquentialiser à priori sans connaître l'ordre (partiel) des évaluations des composantes de X_i : G_{ij} est un "gain séquentiel".

Remarques :

1) En pratique on cherche la granulation la plus fine mais en général, rien ne permet de dire que le seul "gain séquentiel" ne sera pas PROGi lui-même.

2) Le graphe des PROGij, i = 1, ..., N, j = 1, ..., g_i est le graphe granulé, conditionné par les H_i.

C.2) Désynchronisations partielles des grains séquentiels.

Les instants de H_i sont les dates des événements "évaluation de X'_i" ou des multi-événements "évaluation des X'_{ij}, j = 1, ..., g_i". Dans ce dernier cas les événements élémentaires sont les évaluations des X'_{ij} qui se font aux instants d'horloges H_{ij}, i = 1, ..., N, j = 1, ..., g_i.

La désynchronisation partielle consiste à replacer H_i par {H_{ij} | j = 1,.. gi} dans (8).

Les limites de cette désynchronisation viennent :

1) du respect de l'ordre partiel induit par les dépendances inter "grains séquentiels",
par exemple si X_{ij} - > $X_{ij'}$ alors count(H_{ij}) $\geqslant$ count($H_{ij'}$)

2) de la profondeur du pipeline choisie ("chevauchement" partiel de plusieurs exécutions de G_i)

Par exemple, sans pipeline on a :

$\forall i, \forall h$, h horloge, (count(H_i)-1) $\otimes h \leqslant$ count(H_{ij}) $\otimes h \leqslant$ count (H_i) $\otimes h$, j = 1,..gi

(i.e. en cours d'un multi événement, un événement particulier peut être ou ne pas être intervenu).

Avec pipeline, count (H_{ij}) peut varier de plus que 1 avec j. Ces variantes de mise en oeuvre seront vues ailleurs.

Notons SEQ_{ij} une fonction $H_i \rightarrow H_{ij}$ d'ordonnancement a priori des grains séquentiels.

Les fonctions SEQ_{ij} sont à calculer en fonction de la machine cible et de ses performances.

D. Prise en compte de la taille mémoire et des durées d'exécution des grains séquentiels

A une paire (grain séquentiel, processeur) sont associées :

τ_{ij} durée de calcul (communications externes au grain non comprises),
m_{ij} taille mémoire requise pour atteindre la durée précédente.

A une paire (grain séquentiel, système de communication) sont associées :

cij durées des communications externes au grain

Les systèmes de communication pouvant être : mémoire-mémoire d'un même processeur, bus, route ...

Un problème usuel est alors de placer les grains sur des processeurs, caractérisés par (τ_{ij}, m_{ij}) $i = 1, ..., N$, $j = 1, ..., g_i$ puis de choisir les SEQ_{ij} pour séquencer totalement les grains placés sur un même processeur, puis de choisir les moyens de communication inter-grain.

Les durées non prises en compte (gestion des ressources de communication, ...) conduisent à une mesure des performances effectives en utilisant le simulateur fonctionnel. Pour cela on munit chaque processeur d'une base de temps locale, d'horloge $Hproc_k$, $k = 1, ..., P$ qui sert à compter le temps passé à calculer et à communiquer, et qui sert à évaluer les SEQ_{ij}.

Soit $P_k(i)$ le sous ensemble des indices j pris dans 1, ..., g_i, des grains placés sur le processeur k, $k = 1, ..., P$. On a $\underset{k}{U} P_k(i) = \{1, ..., g_i\}$ (tous les grains sont placés) et les SEQ_{ij} pour $j \in P_k(i)$ sont choisis tels que $\forall j \neq j' \in P_k(i)$ H_{ij} et $H_{ij'}$ n'ont pas d'instants communs et sont sous horloge de $Hproc_k$: les grains affectés à un processeur sont strictement séquencés. On donne un exemple de calcul des H_{ij} et du temps passé $time_k$.

C'est le processus "base de temps locale" qui est alors, en faisant l'hypothèse :

$$Hproc_k \geqslant H_i, \ \forall i \text{ tel que } P_k(i) \neq \emptyset$$

```
PROCR-TIME.BASE.k{ ?Hproc_k, H_i  !H_ij, (j∈P_k(i),i = 1,..,N), time_k}
(|
| un := 1 ⊗Hproc_k
| time_k := ztime + un
| ztime := $time_k init 0
| H_ij := Hproc_k when (count(H_ij') = count(H_i), ∀j' ∈ A(j))
|)
```
(9)

où :
A(j) est l'ensemble des indices des grains à exécuter avant le grain j.
L'overhead δT_{ik} sur le processeur k, par exemple, peut être estimé par :

$$\delta T_{ik} = T_{ik} - \left(\sum_{j \in P_k(i)} \tau_{ij} \right) \otimes H_i$$

avec : $T_{ik} = time_k \otimes H_i - \$(time_k \otimes H_i)$

E. Forme du programme partiellement placé et ordonnancé

```
PROG-EXEC{ ?U, Hproc_k, Φ  !X, time_k(k = 1, ..., P)}
(| PROG-TIME.BASE{ ?U, Φ  !H_i(i = 1, ..., N)}
| PROCR-TIME.BASE.k{ ?Hproc_k, H_i  !H_ij, (j∈P_k(i), i = 1,., N), time_k}
| U_i^k := (U ⊗Hi) cell Hproc_k
| X_i^k := (X ⊗Hi) cell Hproc_k
| PROGij{ ?U_i^k, X_i^k, H_ij  !X'_ij}
| X_ij^a := $X'_ij
| X̃ := (X_ij^a cell H_i, j = 1,..,g_i,i = 1,..,N)
| Ũ := (U,B(X))
|)
```
(10)

On a noté X_{ij}^a la version désynchronisée de X_{ij} qui est lui reconstitué (re-synchronisation) par X_{ij}^a cell H_i.

Le programme est complètement placé et ordonnancé lorsque PROG-TIME.BASE est lui aussi réparti sur les processeurs, ce qui donne alors les processus LOGICAL-TIME.BASE.k, $k = 1, ..., P$, les horloges libres Φ étant exprimées à l'aide des $Hproc_k$.

La valeur d'un placement ordonnancement peut s'évaluer, à i donné par exemple par :

$$T_i = \underset{k}{\text{Max}}\, T_{ik}$$

Lorsque N = 1 (programme monohorloge) T_1 est la mesure globale de performance. C'est le critère que nous avons considéré jusqu'à présent. Parmi les contraintes de placement prises en compte on considère la mémoire disponible sur les processeurs :

$$\sum_{i,j \in P_k(1)} m_{ij} \leqslant M_k\,,\ k = 1, ..., P$$

et les capacités de calcul et de communication de chaque processeur.

C'est le problème d'optimisation de niveau 1 qui sera vu ailleurs.

La figure 2 illustre la compilation d'un programme SIGNAL synchrone en un programme SIGNAL asynchrone réparti sur multiprocesseur.

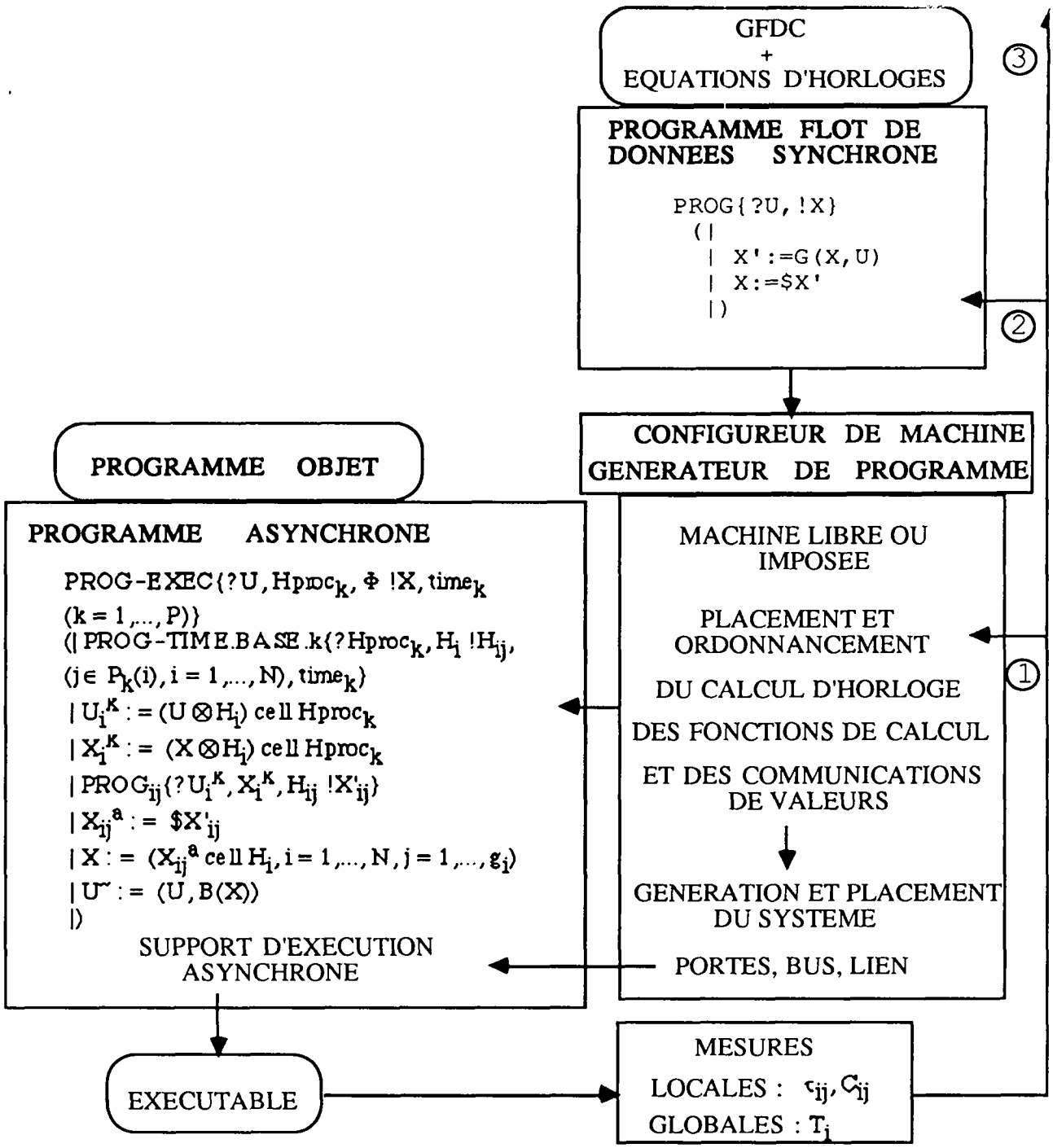

Fig. 2 Programmes synchrone et asynchrone

F. Protocoles de communication

Représentons par $\leftrightarrow$ une communication synchrone (par rendez-vous) entre P et Q : P $\leftrightarrow$ Q et P $\rightarrow$ Q une communication asynchrone. Divers schémas de communication sont possibles, par exemple dans le cas de deux horloges H_i et $H_{i'}$:

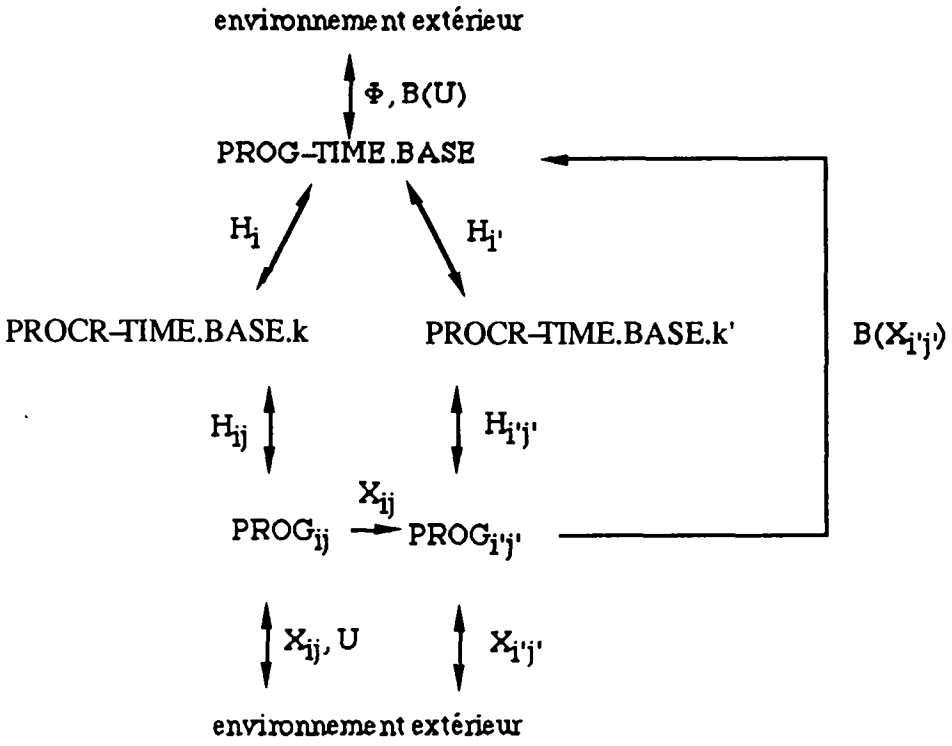

Fig. 3a

La figure 3a présente des communications synchrones pour les horloges et les communications avec l'extérieur, asynchrones pour les autres. Après "optimisation" des informations de contrôle sont envoyées avec des flots de valeurs :

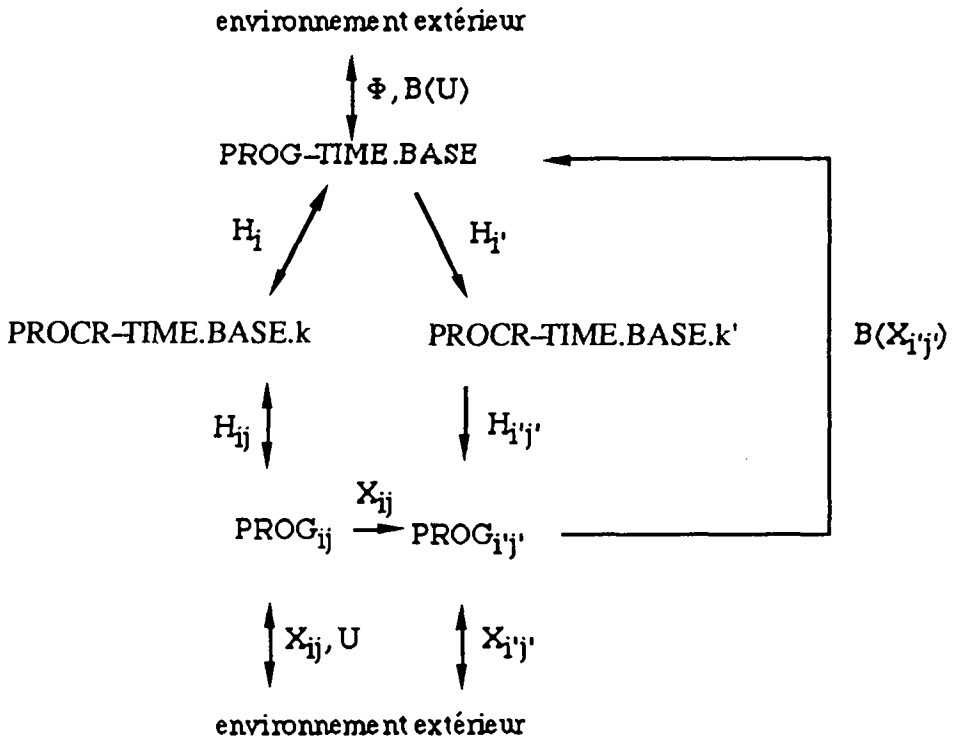

Fig. 3b

Le cycle $\{H_{i'}, H_{i'j'}, B(X_{i'j'})\}$ permet de communiquer $H_{i'}$ et $H_{i'j'}$ en asynchrone. Si de plus $H_i \leqslant H_{i'}$, le cycle $\{H_i, H_{ij}, X_{ij}, B(X_{i'j'})\}$ permet de communiquer H_i et H_{ij} en asynchrone.

Le choix de ces protocoles conduit aux problèmes d'optimisation de niveau 2. Cela sera vu ailleurs.

Nous retenons ici que les communications synchrones aussi bien qu'asynchrones sont utiles et nous indiquerons plus loin comment elles sont mises en place dans SYNDEX.
Nous récapitulons maintenant les problèmes d'optimisation soulevés.

3 - Dimensionnement de la machine cible, répartition des calculs et des communications, génération et réglage du support d'exécution

La figure 2 illustre aussi les optimisations possibles de la machine et des programmes qui se font à trois niveaux.

Niveau 1 : affectation des ressources et ordonnancement

Les ressources à affecter étant, comme on l'a déjà dit, les processeurs, la mémoire, les moyens de communication. On a considéré (cf [4]) deux types de problèmes :
. Affectation à ressources imposées
La machine étant donnée, il s'agit de minimiser le temps de traversée du graphe de l'application (valué par ailleurs) obtenu après affection, ceci sous contrainte de non saturation des ressources (capacités de calcul, de communication, de mémoire, limités).
. Affectation à ressources libres
Le type des ressources est imposé (les performances), leur quantité est libre. On cherche alors à réaliser le temps de traversée du graphe initial de l'application en minimisant les ressources mises en oeuvre, sous contrainte de leur non saturation.

Les algorithmes utilisés pour résoudre ces problèmes sont des heuristiques détaillées dans [4] et qui seront repris dans la description de l'optimiseur de SYNDEX. Disons simplement ici que les étapes principales de leur mise en oeuvre sont :

1. placement des activités de calcul sur les processeurs et ordonnancement des calculs et des communications
2. placement des communications sur les routes et allocation de la mémoire
3. génération des fonctions système.

Remarquons que la formulation déterministe des problèmes de dimensionnement / placement est justifiée par la nature statique du graphe flot de données issu de SIGNAL et la possibilité dans les applications envisagées de mesurer à priori les durées nécessaires. Actuellement ces algorithmes sont utilisables lorsque le graphe d'application est indépendant de l'instant logique auquel on le considère (programme SIGNAL monohorloge). Le cas ou ce graphe dépend du temps (conditionnement de certaines dépendances) est à l'étude.

Niveau 2 : choix des mécanismes de synchronisation

Il s'agit de mettre en forme et interpréter les équations d'horloge générées par la compilation d'un programme SIGNAL : les dépendances entre les horloges intervenant dans ces calculs conduisent à des flots de contrôle interprocesseur dont il d'agit de maîtriser le volume, par réécriture des équations, par association du contrôle aux données (acquittement par exemple). Enfin les processus produisant ces flots de contrôle doivent être placés sur la machine.

Niveau 3 : choix du code des processus, du type des processeurs

Nous n'envisageons ici que de présenter à l'utilisateur des informations lui permettant de faire ces choix. Par exemple la mise en évidence de l'activité / inactivité des processeurs et en particulier des chemins critiques de processeurs (toujours actifs) peut conduire à optimiser le code des

processus affectés à ces processeurs ou de prendre des processeurs plus rapides le long de ces chemins.

Le résultat du placement peut conduire à réécrire le programme SIGNAL en séquentialisant les actions devant être exécutées par un même processus et en générant les mécanismes de synchronisation "efficaces" (issus de l'optimisation de niveau 2) : on peut penser réduire ainsi les appels au système (ordonnanceur, ...).

4. Prototype de SYNDEX

Il existe un prototype de SYNDEX [5] qui offre les fonctionnalités précédentes à l'exception de la gestion du temps (en conséquence seuls les programmes SIGNAL monohorloge sont supportés dans cette version, de sorte que seules des optimisations de niveau 1 sont effectuées).

Ce prototype a été développé en Smalltalk-80 [6] sur SUN3/60 et SUN4/260 sous UNIX et PC386 sous MSDOS.

Il offre une interface graphique et textuelle interactive qui permet de décrire une machine cible et son programme d'application, puis génère automatiquement le programme système qu'il faut ajouter au programme d'application, soit pour effectuer une simulation de la machine cible sur une machine de simulation, soit pour exécuter le programme d'application sur la machine cible en temps réel. Le placement du programme d'application sur la machine cible se fait soit manuellement, soit assisté par "l'optimiseur" qui est un programmes d'optimisation paramétrable.

Dans cette version SYNDEX génère un simulateur codé en source OCCAM [7] et un exécutif pour une machine multi-TRANSPUTER [8].

Il existe une interface graphique et textuelle et un compilateur du langage SIGNAL qui est utilisé pour vérifier que le programme est correct et effectuer le calcul d'horloges. Avec SYNDEX on utilise ces programme corrects sous leur forme graphe flot de données conditionné.

II Implantation d'algorithmes sur une machine multiprocesseur

La construction du simulateur et la génération des exécutifs pour une machine cible s'effectuent en trois phases principales. On décrit les algorithmes associés à l'application de traitement du signal étudiée (sous la forme d'un graphe dit logiciel), on décrit ensuite la machine si elle est imposée (sous la forme d'un graphe dit matériel), enfin on la programme, cela revient à réaliser l'implantation des algorithmes sur la machine. Ces opérations sont décrites dans la suite à l'aide d'une syntaxe textuelle proche de celle de SYNDEX donnée dans [5].

1. Description des algorithmes

Elle s'effectue à partir d'un graphe d'activités élémentaires flot de données, éventuellement conditionné au sens du langage SIGNAL[1]. Ce graphe est construit par l'utilisateur ou bien généré automatiquement par le compilateur d'un langage de haut niveau adapté à la description d'algorithmes de traitement du signal comme le langage SIGNAL. A chaque *activité élémentaire de calcul* ou noeud du graphe (les G_{ij} vus précédemment) est associé un processus de calcul écrit dans le langage choisi pour la génération de code, par exemple en OCCAM. Ce processus représente les actions de calcul que le noeud devra exécuter. les entrées (resp. les sorties) de ce processus se font à partir (resp. dans) de buffers. Ceci permet d'instancier plusieurs fois un même processus de calcul afin de construire des activités élémentaires différentes partageant le même code.

Ces processus de calcul sont déclarés dans une bibliothèque de processus :

USE Nom-de-bibliothèque.Nom-de-graphe

Les communications entre activités élémentaires de calcul (arc du graphe logiciel) sont modélisées par des canaux équivalents aux canaux OCCAM voir fig. 4, ces canaux alimentent les buffers cités plus haut.

Les activités élémentaires de calcul se déclarent en donnant leur nom, le nom du processus de calcul associé appartenant à une bibliothèque et son interface :

(PROCESS Nom-activité-éléméntaire Nom-processus-appelé Interface).

PROCESS représente une déclaration générique, les activités peuvent être des fonctions immédiates FUNCTION, des retards DELAY, des opérateurs temporels de SIGNAL.

L'interface correspond aux noms des entrées ou des sorties typées, précédées respectivement de ? ou !. Un canal relie une sortie à une entrée. Un canal entrant (resp. sortant) est un canal paramètre d'une activité, sur lequel l'activité reçoit (resp. émet).

Pour construire un graphe logiciel on connecte des activités élémentaires de calcul entre elles

(CONNECT Nom-de-process/nom-de-sortie Nom-de-process/nom-d'entrée).

Un graphe d'activités est connexe et orienté voir fig. 4, tous les arcs sont connectés aux deux bouts, certains noeuds peuvent être terminaux (noeud d'entrée-sortie) dans la mesure où ils ne possèdent aucun canal entrant ou aucun canal sortant.

Dans la suite nous ne nous interessons qu'à la programmation d'un seul graphe sur multiprocesseur. Chaque graphe peut par exemple décrire les actions à exécuter à un instant logique d'un programme SIGNAL : exécution de $PROG_i$.

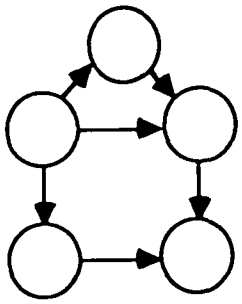

Fig. 4 Graphe d'activités élémentaires

2. Description de la machine

On dispose pour décrire une machine, d'objets, parmi lesquels on trouve des *composants* de base : *processeurs programmables* et *composants spécifiques* et des *moyens de connexions,* les *bus* et les *liens,* voir fig. 5. On trouve aussi des composants obtenus par assemblage (effectué par connexions) des objets précédents, la machine à décrire est le composant final.

Le nombre de composants de bus et de liens à utiliser et la façon de les relier sont choisis par l'utilisateur (machine imposée) ou bien calculés par un programme d'optimisation [4] (machine libre).

2.1. Déclaration des composants et des bus

déclaration des processeurs
(PROCESSOR Nom Liste-de-liens),

déclaration des composants spécifiques
(SPEC Nom Liste-de-liens),

déclaration des composants
(COMP Nom liste-de-liens Liste-de-bus),

déclaration des bus
(BUS Nom Type)

Les déclarations des composants de bibliothèque se font de la manière suivante :

USE Nom-de-bibliothèque.Nom-de-composant

On appelle *lien*, une paire de canaux ayant des sens de transmission "opposés". La liste de liens et de bus qui apparait dans la déclaration définit l'interface avec l'extérieur du composant.

La structure du processeur est la suivante : chacun de ses liens est relié à un *bus logiciel* par l'intermédiaire d'une *porte d'entrée-sortie.* Les bus logiciels et les portes d'entrée-sortie sont décrits plus loin. Les composants spécifiques ont une structure libre, ce sont des processus standards communicant avec l'extérieur par des liens.

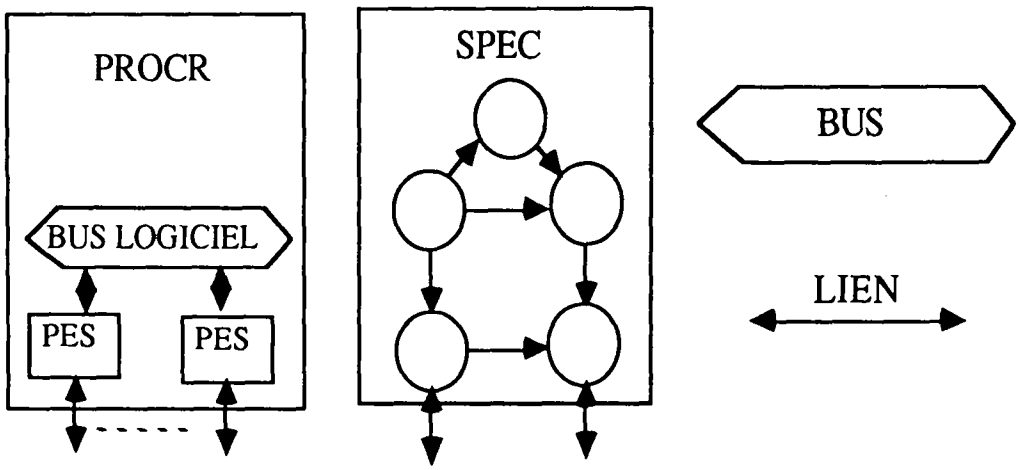

Fig. 5 Composant, bus et lien

2.2. Connexion et déconnexion de composants

Connexion et déconnexion de lien

(CONNECT Nom-de-composant1/nom-de-lien Nom-de-composant2/nom-de-lien)
(DISCONNECT Nom-de-composant1/nom-de-lien Nom-de-composant2/nom-de-lien)

Connexion et déconnexion de bus

(CONNECT Nom-de-bus Liste-de-noms-de-lien)
(DISCONNECT Nom-de-bus Liste-de-noms-de-lien)

La connexion d'un processeur à un bus s'effectue avec un seul lien.

La déconnexion de deux composants supprime toutes les connexions des types précédents voir fig. 6.

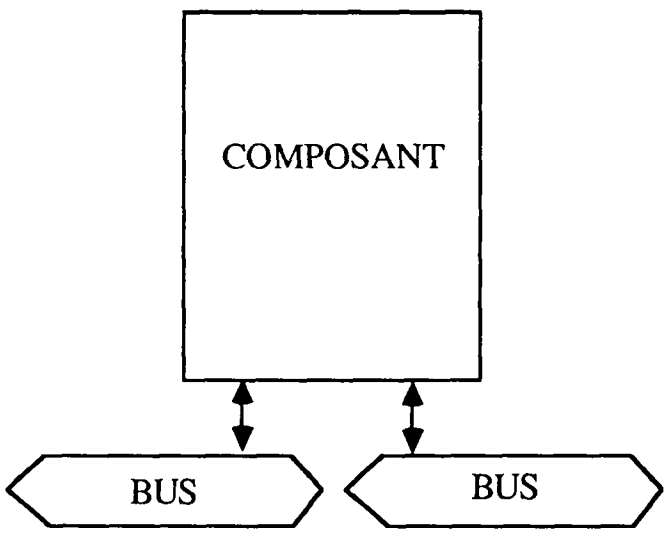

Fig. 6 Connexion d'un composant à deux bus

3. Répartition des activités de calcul et des canaux de communication : programmation de la machine

On effectue tout d'abord une partition d'un graphe d'activités, puis on affecte ou place chaque élément de la partition sur un processeur programmable, on place ensuite les canaux faisant

communiquer des éléments différents de la partition sur des bus et des processeurs, enfin on configure les bus. On a obtenu à ce moment une machine programmée, chaque processeur programmable est devenu un processeur programmé.

3.1. Répartition et affectation des activités

On partitionne le graphe d'activités en un ensemble de sous graphes voir fig. 7. Les canaux sont maintenant, soit canaux internes de sous-graphe, soit canaux externes (provenant du graphe initial ou reliant deux sous-graphes).

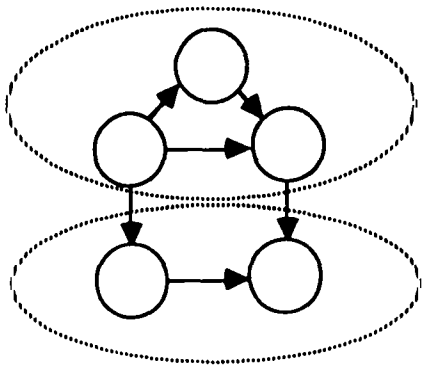

Fig. 7 Partition d'un graphe

On place sur chaque processeur programmable de la machine, un élément de la partition du graphe d'activités voir fig. 8. Ce sous graphe affecté sera connecté au bus logiciel interne du processeur par des *portes* lors de la programmation.

(PLACEPROCESS Nom-processeur Liste-de-process)

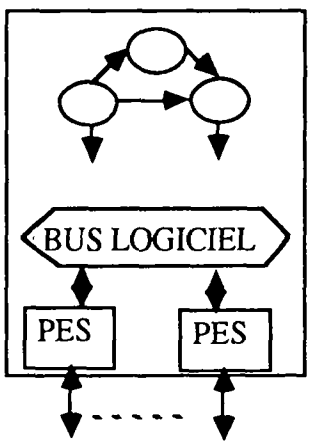

Fig. 8 Affectation d'un élément de partition à un processeur

3.2. Répartition des canaux externes et programmation des processeurs

On place sur des *routes* les canaux externes pour lesquels on connait les activités émettrices et réceptrices. Chaque route relie deux composants en passant éventuellement par plusieurs composants, bus et liens. Si une route passe par un seul bus ou lien c'est une *route directe.* Les canaux externes dont l'émetteur ou le récepteur ne sont pas connus ne sont pas affectés.

Les routes sont déclarées en donnant la succession des composants et des bus et liens qui la constituent.

(ROUTE Nom-de-route Liste-d'élément-de-route)

Un élément de route est formé d'un nom de processeur et d'un nom de lien, sauf pour le dernier processeur de la route pour qui son nom est suffisant.

On place chaque canal externe sur une route déclarée comme précédemment.

(PLACECOM Nom-de-canal-externe Nom-de-route)

Dans un processeur extrémité de route et pour une activité élémentaire émettrice (resp. réceptrice) située sur ce processeur, on crée une activité élémentaire nommée *porte d'émission* (PE) (resp. de réception (PR)) entre l'activité élémentaire déjà existante et le bus logiciel des processeurs participant à la communication (programmation des processeurs terminaux). PE est reliée à l'activité élémentaire par un lien et au bus logiciel par un autre lien.

Dans tous les processeurs les PES sont modifiées pour prendre en compte les informations de routage. Une PES peut être partagée par plusieurs canaux, son arbitrage est alors géré par une activité élémentaire nommée *Porte d'arbitrage de routes* (PART) placé sur un des processeurs (programmation des processeurs intermédiaires).

Pour les composants qui ne sont pas des processeurs, le placement ne crée aucune modification. Leur programmation éventuelle (dans le cas où ils sont constitués en partie de processeurs) est à la charge de l'utilisateur.

Lorsque les activités et les canaux externes ont été placées la *programmation* des processeurs se trouvant sur des routes est effectuée, voir fig. 9.

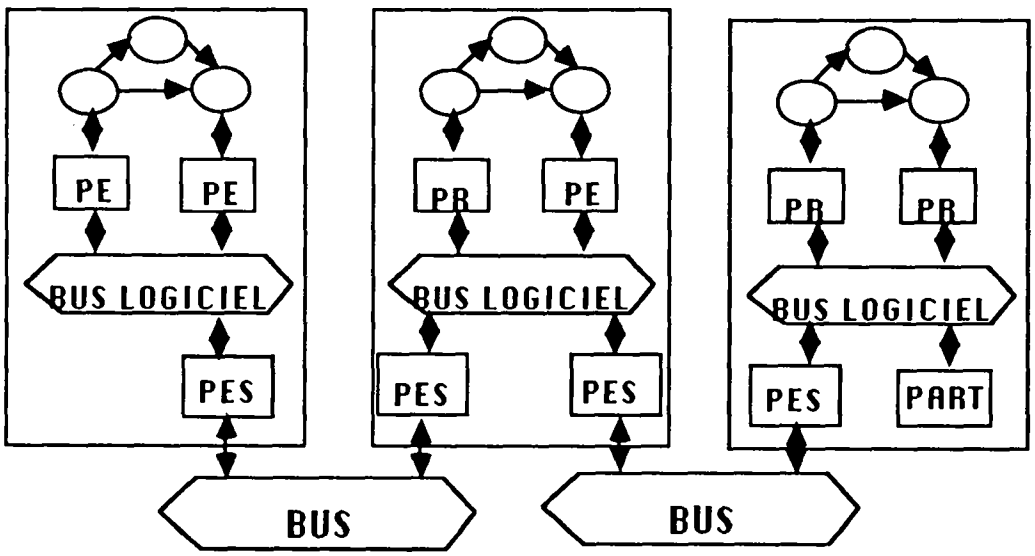

Fig. 9 Machine programmée

3.3. Configuration des bus

Dans le cas "data driven" les arbitres des bus logiciels gèrent les conflits d'accès aux portes de sortie des processeurs (voir plus loin) et les arbitres des bus physiques gèrent les conflits d'accès aux portes d'entrée.

- Déclaration des sémaphores.

Un bus logiciel peut contenir un sémaphore par porte d'entrée-sortie (utilisation maximale du parallélisme des entrées sorties) ou éventuellement un seul sémaphore (sorties séquentialisées par le processeur).

Un bus physique contient un seul sémaphore. Le processeur qui accède au bus peut choisir le ou les destinataires parmi tous les processeurs eux même connectés au bus.

- Configuration des sémaphores.

Chaque sémaphore se configure en choisissant les primitives INIT, GET et PUT.

III Description du support d'exécution

Un canal de type OCCAM au niveau du graphe d'activités de l'application, s'il devient externe lors de la partition, est transformé en une route implantée par une succession de portes, de bus et de liens. La communication entre ces portes s'effectue par passage de messages [6] depuis la porte d'émission jusqu'à la porte de réception en passant par des portes d'entrée-sortie intermédiaires. Celles ci reçoivent des messages, les stockent dans leurs buffers et les réémettent en tenant compte des informations de routage. Aux extrémités de la route on retrouve les activités élémentaires de calcul AEE et AER du graphe d'activités de départ, réécrites de manière à mettre en oeuvre un protocole de communication global ("bout en bout" au sens des couches ISO réseau et transport [7]. Ce dernier peut être synchrone ou asynchrone, de plus il peut faire ou non appel à un arbitre qui gère les routes concurrentes (i.e. qui partagent des bus ou des buffers) afin de prévenir les interbloquages [6].

La répartition des algorithmes sur la machine multiprocesseur conduit donc à ajouter un exécutif (couche système) aux algorithmes (couche application) donnés au départ. Cela est nécessaire afin de gérer convenablement les communications interprocesseurs. On décrit tout d'abord les mécanismes de communications à l'aide d'un ensemble de primitives indépendantes des choix d'implantation qui ont été faits pour la génération effective de l'exécutif. Ce dernier est construit automatiquement à partir d'un noyau d'exécutif représenté par des objets systèmes appelés portes (PE, PR, PES, BL, PART). Chacun de ces objets est lui même composé d'objets de base comme le routeur RT, l'arbitre ARB, la FIFO, les connecteurs de bus et d'application. Lors de la génération de code on effectue de la macro génération à partir des bibliothèques systèmes contenant ces objets système codés dans le langage choisi, par exemple en OCCAM si les processus de calcul sont décrits en OCCAM.

1. Protocoles et primitives de communication

1.1. Principes

Une communication entre deux processus utilisant comme support un canal, est une communication synchrone (c'est un rendez-vous OCCAM, émetteur et récepteur coopèrent de telle manière que la communication a lieu quand les deux processus sont prêts à communiquer). Lorsqu'on intercale entre ces deux processus un processus intermédiaire, celui-ci contient nécessairement au moins un buffer (éventuellement plusieurs qui seront partagés par plusieurs communications voir paragraphe III.1.2.2) pour permettre le transfert de p1 vers p2 à travers pi voir fig. 10. Ceci réalise le mécanisme d'une boîte aux lettres. Les communications entre p1 et pi, et entre pi et p2, restent, prises séparemment, des communications synchrones (rendez-vous OCCAM).

La communication entre p1 et p2 devient asynchrone puisque l'on a introduit une boîte aux lettres entre p1 et p2. Emetteur et récepteur peuvent réaliser la communication sans avoir besoin de coopérer: l'émetteur peut grâce à la boîte aux lettres envoyer les données de façon arbitraire sans nécessairement savoir si le récepteur est prêt à les recevoir sous réserve qu'il y ait de la place dans celle ci. On peut généraliser en introduisant autant de buffers que l'on veut entre P1 et P2.

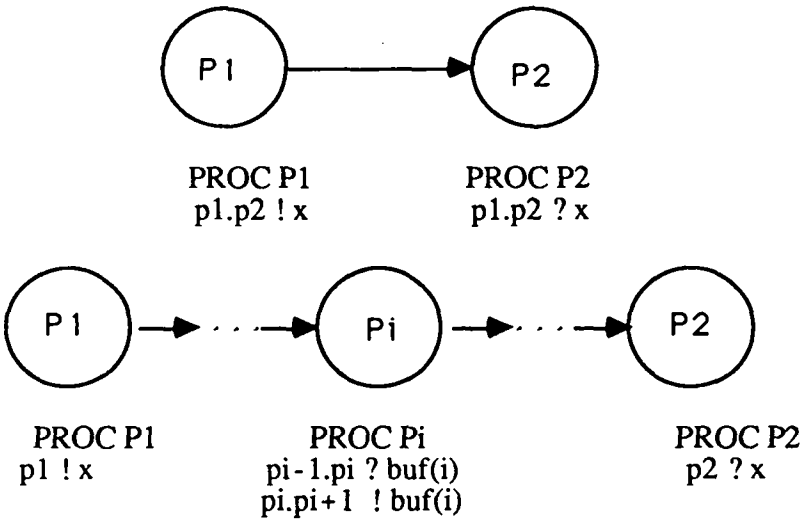

Fig. 10 Communications locales et globales

1.2. Protocoles

1.2.1 Protocole synchrone et asynchrone

Il est possible de rendre la communication de bout-en-bout p1 p2 synchrone en introduisant au niveau de p1 et p2 un protocole adapté, leur permettant de coopérer pour effectuer la communication. Ce protocole peut être réalisé actuellement de deux manières :

-protocole synchrone "data driven", l'émetteur envoie ses données mais ne redémarre que lorsque le récepteur lui a indiqué qu'il les a reçues,
-protocole synchrone "demand driven", l'émetteur doit attendre que le récepteur soit prêt à recevoir avant d'envoyer ses données, puis il peut redémarrer après avoir reçu un accusé de réception. Dans le premier cas, ceci nécessite l'envoi par le récepteur d'un message d'acquittement qui peut transporter des données en réponse. Dans le deuxième cas cet acquittement est facultatif.

Dans le cas d'un procole asynchrone "data driven", l'émetteur envoie ses données et peut redémarrer aussitôt.

1.2.2 Protocole de réservation de route

Lorsqu'on considère l'ensemble des canaux devenus externes après répartition du graphe d'activités, certains d'entre eux qui pouvaient communiquer en parallèle peuvent être affectés à des routes concurrentes. Deux routes sont concurrentes si elles ont au moins un buffer en commun. Pour prévenir les interbloquages [6], les buffers partagés par plusieurs routes, sont gérés globalement par un arbitre. C'est lui qui détermine, connaissant l'état des buffers, quelles communications peuvent s'exécuter. Pour les routes avec conflit potentiel, il est nécessaire de faire appel à un arbitre. Il faut donc envoyer des messages de réservation et de relâchement de ces buffers, vers cet arbitre. Ces messages doivent impérativement utiliser quant à eux des routes sans conflit.

Ceci constitue un second niveau de protocole qui s'ajoute au protocole de base vu plus haut (synchrone ou asynchrone). Il peut prendre plusieurs formes en fonction de la manière dont est faite la réservation de la route pour les données et pour l'acquittement.

Pour conclure on a quatre types de protocoles, synchrones ou asynchrones avec ou sans réservation de route.

1.3. Primitives de communication

Les primitives de communication de plus haut niveau (niveau application) SEND et RECEIVE, permettent de décrire les communications synchrones ou asynchrones avec ou sans

réservation de route, entre les activités élémentaires d'émission et celles de réception qui représentent le graphe de l'application. Ces primitives se décrivent à l'aide de primitives de plus bas niveau (niveau route) qui réalisent plusieurs communications asynchrones de bout en bout sur des routes. A chaque bout on dispose respectivement des primitives SENDRT et RECEIVERT relayées par une ou plusieurs primitives FORWARDRT. De même ces primitives se décrivent à l'aide de primitives de plus bas niveau (niveau bus) qui réalisent des communications asychrones point à point sur des bus (logiciel, physique ou lien) et qui sont SENDBS et RECEIVEBS. Enfin les primitives PUTMSG et GETMSG réalisent le transfert effectif par dépot et retrait de messages, des données et contrôles sur les bus.

1.3.1 Description des primitives SEND RECEIVE

Les protocoles synchrone ou asynchrone avec ou sans réservation de route vus plus haut s'écrivent avec les deux primitives SEND et RECEIVE en utilisant les primitives SENDRT et RECEIVERT de la manière suivante :

la porte d'émission PE effectue un SENDRT(données), puis dans le cas synchrone reste en attente sur un RECEIVERT(réponse). Les portes d'entrée - sortie intermédiaires PES propagent le message par des FORWARDRT jusqu'à la porte de réception PR en attente sur un RECEIVERT(données). Puis dans le cas synchrone, elle renvoie par un SENDRT(réponse) la réponse que propagent les portes intermédiaires vers la porte d'émission.

Si les routes sont concurrentes la réservation des routes est nécessaire. On doit ajouter uniquement au niveau des portes d'émission et de réception une couche de protocole au niveau réseau de l'ISO, écrite avec les primitives SENDRT et RECEIVERT.

Ce protocole peut prendre plusieurs formes selon que la route empruntée est réservée ou non et selon celui de l'émetteur (E) ou du récepteur (R) qui fait la requête de réservation (REQ) et la requête de relâchement (REL). Le tableau suivant Tab.1 donne quelques exemples de ces protocoles.

données		acquittement	
req	rel	req	rel
X	X	X	X
E	R	X	X
E	E	X	X
E	R	R	E
X	X	E	E

Tab.1 Protocoles de réservation de route

On donne successivement une brève explication pour chaque ligne du tableau :

- aucune réservation ceci est équivalent au protocole de base de la couche réseau.
- réservation d'une seule route faite par l'émetteur pour les données, relâchement par le récepteur,
- pas de réservation pour la route d'acquittement, réservation et relâchement faits par l'émetteur pour les données,
- utilisation de deux routes différentes pour données et acquittement, l'émetteur réserve la route de données, relâche la route d'acquittement et inversement pour le récepteur,
- pas de réservation pour la route de données, réservation et relâchement faits par l'émetteur pourl'acquittement.

On donne en exemple le code des primitives SEND et RECEIVE dans le troisième cas pour une communication synchrone :

```
SEND(données, réponse)

SEQ
  SENDRT(aee.arb, route-req, )
  RECEIVERT(arb.aee, route-grant, )
  SENDRT(aee.aer, data, données)
  RECEIVERT(aer.aee, acq, réponse)
  SENDRT(aee.arb, route-rel, )
```

```
RECEIVE(données, réponse)

SEQ
   RECEIVERT(aee.aer, data, données)
   SENDRT(aer.aee, acq, réponse)
```

Les trois paramètres des primitives SENRT et RECEIVERT sont :
- la route que le message devra emprunter :
  aee.aer et aer.aee désignent les routes de données et d'acquittement
  aee.arb et arb.aee désignent les routes pour les messages de contrôle
- le type du message
- le contenu du message (peut être vide)

1.3.2 Description des primitives SENDRT, FORWARDRT, RECEIVERT

Les primitives SENDRT, FORWARDRT et RECEIVERT servent à réaliser une communication de bout en bout sur une route.

La primitive SENDRT formatte le message à l'aide des informations route, type, contenu, puis envoie le message vers la première PES de la route à travers le premier bus nécessairement logiciel. La primitive FORWARDRT reçoit un message d'une PES à travers un bus et renvoie ce message vers la prochaine PES de la route à travers un autre bus. La primitive RECEIVERT reçoit un message de la dernière PES de la route à travers le dernier bus, nécessairement logiciel et extrait le contenu du message.

Ces primitives s'écrivent en utilisant les primitives SENDBS, RECEIVEBS de la manière suivante :

```
PAR

 SENDRT(route, type, contenu)
  SEQ
    FORMAT(route, type, contenu, message)
    SENDBS(bus1, message)

 FORWARDRT()
  SEQ
    RECEIVEBS(busi, message)
    SENDBS(busi+1, message)

 RECEIVERT(route, type, contenu)
   SEQ
     RECEIVEBS(busn, message)
     EXTRACT(message, route, type, contenu)
```

Pour chaque processeur d'une route il faut traverser nécessairement un bus logiciel puis un bus physique ou un lien . Si la route comprend N processeurs, elle traverse 2N-1 bus en exécutant 2N-2 fois la primitive FORWARDRT.

1.3.3 Description des primitives SENDBS et RECEIVEBS

Les deux primitives SENDBS et RECEIVEBS servent à traverser un bus logiciel, un bus physique ou un lien en réalisant une communication point à point.

Par exemple dans le cas d'un bus physique, la porte qui fait le SENDBS(bus, message) demande le bus à l'arbitre du bus par un PUTMSG(bus, bus-req) et attend l'octroi par un GETMSG(bus, bus-grant), puis envoie le message par un PUTMSG(bus, message), attend l'acquittement de ce message par un GETMSG(bus, bus-ack) et enfin relâche le bus par un PUTMSG(bus, bus-rel). La porte qui fait le RECEIVEBS(bus, message) attend le message par un GETMSG(bus, message) et envoie l'acquittement par PUTMSG(bus, bus-ack).

Ces primitives s'écrivent en utilisant les primitives PUTMSG et GETMSG de la manière suivante :

Dans le cas d'un bus physique :

```
SENDBS(bus, message)          RECEIVEBS(bus, message)

SEQ                           SEQ
  PUTMSG(bus, bus-req)          GETMSG(bus, message)
  GETMSG(bus, bus-grant)        PUTMSG(bus, bus-ack)
  PUTMSG(bus, message)
  GETMSG(bus, bus-ack)
  PUTMSG(bus, bus-rel)
```

Dans le cas d'un bus logiciel :

```
SENDBS(bus, message)          RECEIVEBS(bus, message)

SEQ                           SEQ
  PUTMSG(bus, bus-req)          GETMSG(bus, message)
  GETMSG(bus, bus-grant)
  PUTMSG(bus, message)
  PUTMSG(bus, bus-rel)
```

Dans le cas d'un lien :

```
SENDBS(bus, message)          RECEIVEBS(bus, message)

SEQ                           SEQ
  PUTMSG(bus, message)          GETMSG(bus, message)
```

1.3.3 Description des primitives PUTMSG et GETMSG

Les primitives PUTMSG et GETMSG effectuent respectivement l'envoie et la réception des messages sur les bus, d'un tableau précédé de sa taille qui représente le message.

La primitive PUTMSG envoie le message sur un lien par la primitive OCCAM d'écriture sur un canal éventuellement en traversant un bus physique ou logiciel et la primitive GETMSG reçoit le message sur un lien par la primitive OCCAM de lecture sur un canal.

Ces primitives s'écrivent de la manière suivante :

```
PUTMSG(bus, message)                    GETMSG(bus, message)

  bus.in ! SIZE message ::message         bus.out ? SIZE message ::message
```

1.4 Les services de communications

Nous récapitulons les services de communication de façon hiérarchisée en considérant l'ensemble des bus et des liens comme un seul réseau de communication. Il faut remarquer que ce réseau ne dispose pas d'un mécanisme naturel de diffusion, mais seulement de moyens de diffusion restreinte, les bus.

Les couches de protocole au sens ISO [7], en commençant par le niveau le plus bas sont les suivantes :

1- Liaison physique : elle se fait à l'aide de liens (entre composants et bus ou composant et composant) et s'utilise à travers les primitives PUTMSG et GETMSG. L'exécution de ces primitives fait appel aux actions !, ?, d'OCCAM.

2- Liaison de données : on dispose à ce niveau des moyens d'établissement et de libération de connexions "point à point" utilisant un seul bus physique ou lien. Ces moyens s'utilisent à travers les primitives SENDBS, RECEIVEBS invoquées par les PE, PES, PR.

3- Réseau : on dispose à ce niveau de moyens de communication asynchrones de "bout en bout", les routes, qui s'utilisent à travers les primitives SENDRT et RECEIVERT invoquées par les PE, PR et la primitive FORWARDRT invoquée par les PES.

4- Transport : on dispose à ce niveau de moyens d'établissement et de libération de routes, de fragmentation/réassemblage des messages et d'établissement de rendez-vous (qui tient lieu de contrôle de flux). Ces moyens s'utilisent à travers les primitives SEND, RECEIVE, invoquées par les PE et PR.

5- Session : on dispose à ce niveau de moyens de synchronisation plus évolués faisant éventuellement intervenir une date logique des messages (le réseau seul ne préserve pas l'ordre des messages) ou un conditionnement des communications. Ces moyens s'utilisent à travers des primitives SEND, RECEIVE à paramètres optionnels, invoquées par les PE, PR.Pour l'interprétation de programmes SIGNAL, on aura besoin des mécanismes suivants :

- la diffusion conditionnelle :SEND(liste de destinataires conditionnels, message),
- le mélange :RECEIVE(message1 DEFAULT message2, message),

La liste de destinataires conditionnels est une liste d'éléments du type (destinataire,WHEN c,$n) avec c message de type booléen destiné au réseau pour son contrôle, n entier, spécifiant le retard logique de transmission utilisé pour la configuration du réseau.Les messages sont constitués d'une partie top-message et d'une partie valeur-message. letop-message contient le nom d'une horloge de référence, une date relative à cette horloge, un nom de message et un booléen valant 1 si le top-message est suivi d'une valeur (contenu du message), 0 sinon (valeur absente).

Le type du message est le type de valeur-message.

Pour envoyer un message m, on évalue d'abord tous les booléens destinés au contrôle du réseau et on les envoie sous la forme de messages c. Ce sont eux qui apparaissent dans la liste l de destinataires conditionnels du message.

On constitue alors les listes suivantes :

-l0 : liste des destinataires pour lesquels valeur-c est absente ou fausse,

-l1 : liste des destinataires pour lesquels valeur-c est vraie.

On diffuse ensuite le message m aux éléments de l1 et le message m' aux éléments de l0, avec m' constitué de top-m dans lequel on a mis le booléen à faux (il se peut que m' = m si valeur-m est absente). les messages m ou m' suivent alors des routes conduisants aux éléments de l. Sur chaque route un FIFO de taille n, implantant le retard logique, est réservé aux messages de même nom.Pour mélanger les messages, on suppose qu'ils ont une horloge de référence commune. A chaque instant de cette horloge on reçoit alors top-m1 et top-m2. Si la valeur-message1 est présente, alors message: = message1, sinon, si valeur-message2 est présente alors message: = message2, sinon message est réduit à top-message avec le booléen à 0.

Le contrôle du réseau (évaluations des booléens et envois des messages c) se fait avec la primitive SIGNAL c.

6- Présentation : on présente les données externes fournies par les activités élémentaires de l'application, traduction des primitives en utilisant la syntaxe concrète du niveau inférieur.

7- Application : la partition du graphe des activités.

1.5. Structure des messages

Un message global envoyé par une PE associée à une activité élémentaire émettrice, vers une PR associée à une activité élémentaire réceptrice ou vers une PART associée à un arbitre de route se déplace de porte en porte. Il traverse éventuellement plusieurs bus logiciels, bus physiques et liens. Avant de traverser un bus logiciel (resp. physique), il reste en attente dans une PE ou PES (resp. PES). Cette porte déclenche l'envoi de messages locaux de contrôle pour traverser ce bus (envoi du message de demande de bus, attente du message d'octroi). Le message traverse le bus qui est libéré par un message d'acquittement local venant de la porte suivante. Le message global peut ensuite continuer vers la prochaine porte de sa route jusqu'à sa destination finale.

Les différents champs du message global sont les suivants :

TYPE	RTID	DEST	CONTENU

- le champ TYPE codé sur un octet désigne le type du message qui peut être

le type d'un message de contrôle de route :

RTREQ : demande de route
RTGRANT : octroi de route
RTREL : relâchement de route

le type d'un message de données :

DATA : données
ACK : acquittement

- le champ RTID codé sur deux octets représente un entier (par convention les entiers pairs sont affectés aux routes directes et les entiers impairs aux routes inverses). Il désigne l'identificateur de "route structurelle" celle-ci étant définie par une suite de PES permettant de passer du processeur contenant l'activité émettrice au processeur contenant l'activité réceptrice en traversant 0, un ou plusieurs processeurs intermédiaires.

La route définie dans les paragraphes précédants est donc un triplet (porte émettrice, route structurelle, porte réceptrice). C'est ce champ qui va servir à acheminer les messages de données et de contrôle de route à l'aide des tables de routages contenues dans les bus logiciels (voir III.2.3.). Dans l'hypothèse ou le nombre de routes structurelles est bien inférieur au nombre de routes ceci permet de réduire notablement la taille des tables de routage.

- le champ DEST désigne l'adresse sur le bus logiciel de la porte réceptrice du message, ce champ étant codé sur un octet. Lorsque le message est arrivé dans le dernier processeur de la route structurelle, c'est ce champ qui détermine la dernière porte de la route (PE, PR ou PART)

- le CONTENU du message (de 0 à (256-4) octets) dépendant du type, par exemple un message de type DATA a comme contenu les données provenant de l'application, dans le cas d'un message de type contrôle, le contenu pourrait être une information destinée ou venant de l'arbitre de route.

Pour les messages locaux la structure se réduit au champ type qui peut prendre les valeurs suivantes:

message de contrôle de bus logiciel :

BLREQ : demande de bus logiciel
BLGRANT : octroi du bus logiciel
BLREL : relâchement du bus logiciel

message de contrôle de bus physique:

BUSREQ: demande de bus
BUSGRANT: octroi de bus
BUSREL: relâchement de bus
BUSACQ: acquittement local

2. Description du noyau d'exécutif

2.1. Portes

Les primitives vues au dessus sont exécutées par les portes qui contiennent les buffers.

2.1.1. Porte d'émission (PE)

L'objet porte d'émission est un ensemble de processus OCCAM jouant le rôle d'interface entre l'activité élémentaire émettrice et le bus logiciel. Elle est chargée d'exécuter la primitive SEND(données, réponse), pour cela elle reçoit les données de l'activité élémentaire émettrice selon le protocole spécifié dans le graphe de départ et renvoie la réponse vers l'activité élémentaire.

La porte d'émission cf. fig. 11 contient les deux processus suivants :

-un connecteur de sortie d'application qui réalise la communication avec l'activité élémentaire et assure le déroulement des protocoles aux niveaux réseaux, transport, session et présentation.
-le connecteur de bus logiciel : la fonction d'un connecteur de bus logiciel est d'assurer la transmission et la réception des messages sur un bus logiciel, ces messages provenant ou allant vers le connecteur de sortie d'application .

a - description du connecteur de sortie d'application:

Ce processus communique avec l'activité élémentaire émettrice par l'intermédiaire de deux canaux : un dans le sens activité élémentaire-connecteur de sortie pour récupérer les données à envoyer (le protocole sur ce canal au sens OCCAM du terme (OCCAM reference manuel) pouvant être simple ou séquentiel), un autre dans le sens connecteur de sortie - activité élémentaire pour renvoyer la réponse reçue vers l'application. Il communique aussi avec le connecteur de bus logiciel par l'intermédiaire de deux canaux (un dans chaque sens).

Lorsque le connecteur de sortie d'application reçoit des données de l'application (invocation du SEND par l'activité élémentaire émettrice), il exécute la primitive SEND puis il envoie la réponse à l'application. Comme on l'a vu précédemment l'exécution du SEND consiste en une séquence d'invocations de SENDRT et RECEIVERT qui lors de leur exécution dans le connecteur d'application invoquent à leur tour les primitives SENDBS et RECEIVEBS. Ces dernières seront exécutées par le connecteur de bus logiciel.

b - description du connecteur de bus logiciel cf. fig. 12:

Il est constitué de deux processus en parallèle in et out qui exécutent respectivement les primitives RECEIVEBS et SENDBS. Le processus in reçoit des messages du bus par un canal, filtre les messages de contrôle et renvoie les autres par un canal vers le connecteur d'application. Le processus out reçoit des messages du connecteur d'application et les renvoie par un canal vers le bus. La primitives SENDBS devant recevoir un message de type BLGRANT venant du bus logiciel, c'est le processus in qui reçoit ce message et le tranfert après filtrage au processus out par un canal interne reliant le processus in au processus out. Le GETMSG(bus, bus-grant) de la primitive SENDBS cf. paragraphe III.1.3.3, correspond alors à une lecture sur ce canal interne.

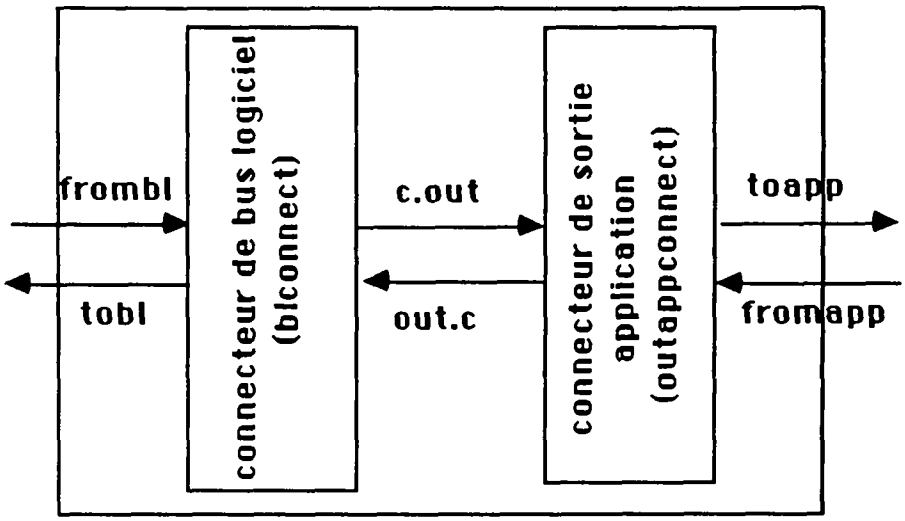

Fig 11 Porte d'émission

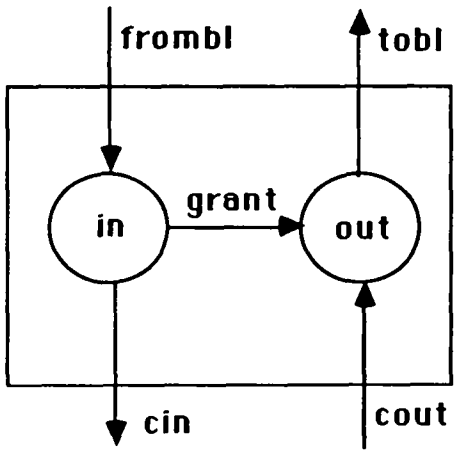

Fig. 12 Connecteur de bus logiciel

2.1.2. Porte de réception (PR)

L'objet porte de réception est un ensemble de processus OCCAM jouant le rôle d'interface entre l'activité élémentaire réceptrice et le bus logiciel. Elle est chargée d'exécuter la primitive RECEIVE(données, réponse). Pour cela elle reçoit les données venant de l'activité élémentaire émettrice lointaine et les transmet à l'activité élémentaire réceptrice, reçoit de cette activité élémentaire la réponse et l'envoie vers l'activité élémentaire émettrice lointaine.

La porte de réception cf. fig. 13 contient les deux processus suivants :

- un connecteur d'entrée d'application qui réalise la communication avec l'activité élémentaire réceptrice et assure le déroulement des protocoles aux niveaux réseau, transport, session et présentation
- un connecteur de bus logiciel (voir paragraphe III.2.1.1).

Le connecteur d'entrée d'application communique avec l'activité élémentaire réceptrice par l'intermédiaire de deux canaux : un dans le sens connecteur d'entrée-activité élémentaire pour envoyer vers l'application les données reçues (le protocole sur ce canal au sens OCCAM du terme pouvant être simple ou séquentiel), un autre dans le sens activité élémentaire-connecteur

d'entrée pour envoyer la réponse. Il communique aussi avec le connecteur de bus logiciel par l'intermédiaire de deux canaux (un dans chaque sens).

Le connecteur d'entrée d'application en attente permanente des données venant de l'activité élémentaire émettrice, exécute la primitive RECEIVE. Les données reçues sont envoyées vers l'application puis la réponse est envoyée vers l'activité élémentaire émettrice. Comme on l'a vu précédemment l'exécution du RECEIVE consiste en une séquence d'invocations de RECEIVERT, SENDRT qui lors de leur exécution dans le connecteur d'application invoquent à leur tour les primitives SENDBS et RECEIVEBS. Ces dernières seront exécutées par le connecteur de bus logiciel.

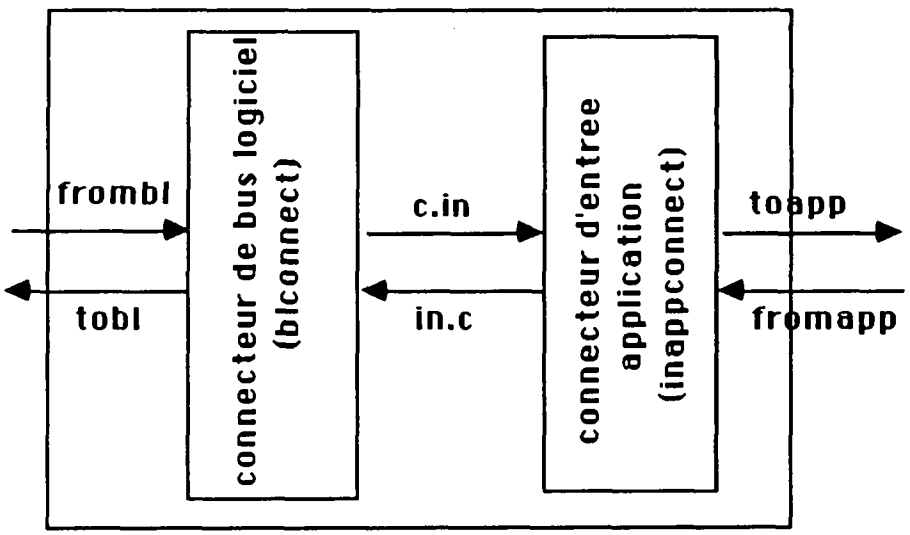

Fig. 13 Porte de réception

2.1.3. Portes d'arbitrage de route (PART)

L'objet porte d'arbitrage de route cf. fig. 14 est un ensemble de processus OCCAM qui interprète les messages de demande et relâchement de routes provenant des portes d'émission et de réception.

Elle contient les trois processus suivants :

- un connecteur de bus logiciel (voir paragraphe III.2.1.1),
- un FIFO permettant l'empilement des requêtes,
- un arbitre qui gère les conflits d'accès aux routes suivant une politique choisie à l'avance.

Actuellement l'arbitrage est centralisé (une seule PART utilisée pour régler chaque zone de conflit).

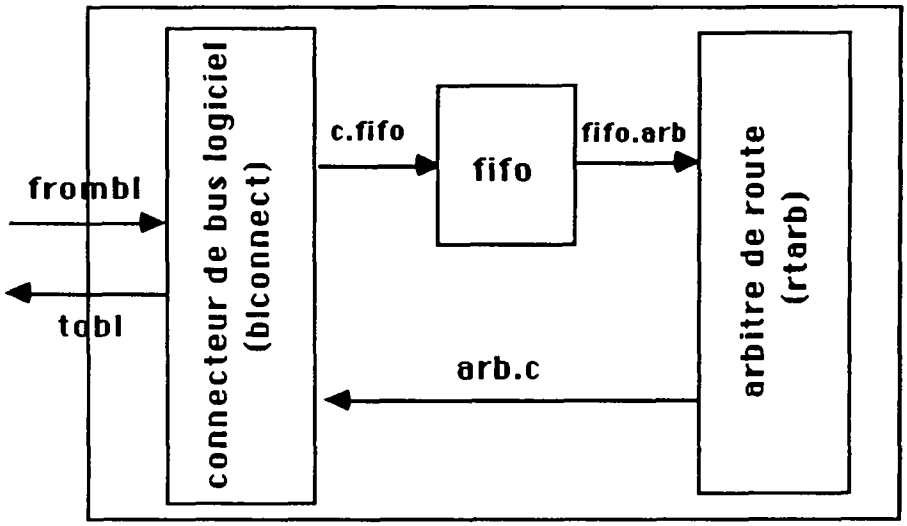

Fig. 14 Porte d'arbitrage de route

2.1.4. Porte d'entrée-sortie (PES)

L'objet porte d'entrée-sortie cf. fig. 15 est un ensemble de processus OCCAM jouant le rôle d'interface entre un bus logiciel et un bus physique ou un lien. Les messages provenant du bus physique (resp. logiciel) et adressés à cette porte, sont stockés et renvoyés vers la prochaine porte de la route (PR, si le message est arrivé à destination, PES sinon) à travers le bus logiciel (resp. physique) ou un lien. La PES est traversée par deux flots de messages qui se croisent.

La porte d'entrée-sortie contient les quatre processus suivants :

-le connecteur de bus : la fonction d'un connecteur de bus est d'assurer la transmission et la réception des messages sur un bus physique, ces messages provenant ou allant vers le buffer ,
-le connecteur de bus logiciel : la fonction d'un connecteur de bus logiciel est d'assurer la transmission et la réception des messages sur un bus logiciel, ces messages provenant ou allant vers le buffer (voir paragraphe III.2.1.1) ,
-deux fifo : ceux-ci représentent les buffers gérés en file de type fifo (premier entré, premier sorti) avec un fifo par sens de transmission.

Description du connecteur de bus cf. fig. 16 :

Il est constitué de deux processus en parallèle in et out qui exécutent respectivement les primitives RECEIVEBS et SENDBS. Le processus in reçoit des messages du bus par un canal, filtre les messages de contrôle et renvoie les autres par un canal vers le connecteur d'application. Le processus out reçoit des messages du connecteur d'application et les renvoie par un canal vers le bus. La primitives SENDBS devant recevoir un message de type BUSGRANT et un message de type BUSACK venant du bus physique, c'est le processus in qui reçoit ces messages et les tranfert après filtrage au processus out par des canaux internes reliant le processus in au processus out. Les GETMSG(bus, bus-grant) et GETMSG(bus, bus-ack) de la primitive SENDBS cf. paragraphe III.1.3.3, correspondent alors à des lectures sur ces canaux internes. La primitive RECEIVEBS devant envoyer un message de type BUSACK vers le bus physique, c'est le processus out qui envoie ce message après avoir reçu la demande d'envoi sur un canal interne reliant le processus in au processus out. Le PUTMSG(bus-ack) de la primitive RECEVEBS, correspond alors à une écriture sur ce canal interne.

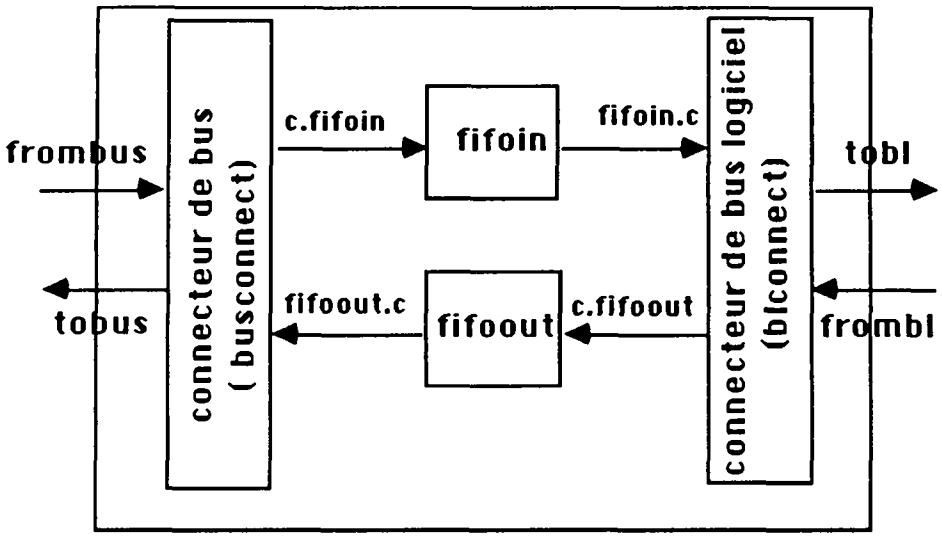

Fig. 15 Porte d'entrée-sortie

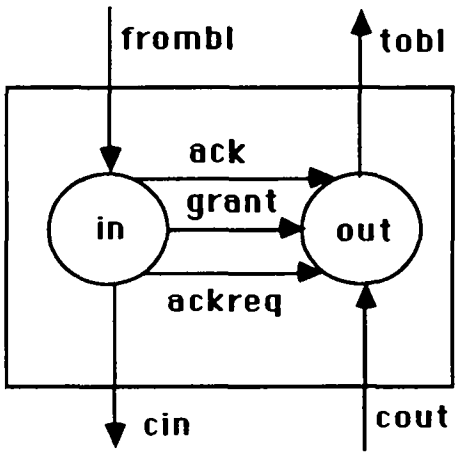

Fig. 16 Connecteur de bus physique

2.2. Activités élémentaires (ACT)

Si on génère du code OCCAM, une activité de calcul appelle un processus OCCAM, elle a la structure suivante :

```
PROC nom-d'activité (CHAN liste-de-canaux)
  WHILE TRUE
    SEQ
      PAR
        In
      Calculs
      PAR
        Out
```

In est un processus qui effectue la lecture de tous les canaux en entrée
Out est un processus qui effectue l'écriture sur tous les canaux en sortie
Calculs est le processus OCCAMappellé qui effectue des calculs.

Une structure de ce type respecte la règle classique d'activation flot de données, les activités sont les noeuds d'un graphe et leurs canaux sont les arcs. Les activités peuvent représenter des actions instantanées décrites en SIGNAL, par exemple ce sont les noeuds du graphe flot de données à représenter à chaque instant logique SIGNAL.

Il y a d'autres activités élémentaires comme les portes, les bus logiciels, ce sont des activités "système".

Un graphe d'activités élémentaires peut être représenté par un programme de type OCCAM où l'on met en parallèle plusieurs activités élémentaires. Ce programme a la structure suivante :

```
PAR i = 0 FOR n
 activité-élémentaire1
 activité-élémentaire2
 :
```

2.3. Bus (BUS)

Un bus est un moyen de connexion multipoint, on emploie un objet de type BUS dans les deux cas suivants :

- pour relier des composants entre eux : auquel cas nous parlerons de bus physique.

- à l'intérieur d'un processeur, pour relier des activités élémentaires à des portes d'entrée-sortie par l'intermédiaire de portes d'émission et de réception, nous parlerons alors de bus logiciel.

Le bus contient deux processus OCCAM voir fig. 17. et 18.

Le processus arbitre qui gère les conflits d'accès, un bus étant vu comme une ressource partagée. On peut choisir une politique quelconque d'ordonnancement des messages.

Le processus routeur qui assure le multiplexage des messages globaux et locaux venant des portes.

Dans le cas du bus logiciel si le message est un message global, les informations contenues dans la table de routage permettent les transferts de PE, PR vers PES et vice-versa ainsi que de PES vers PES (ceci correspond au routage interprocesseurs). Si c'est un message de contrôle pour le bus logiciel, il permet le transfert de PE, PR, PES vers l'arbitre et vice-versa.
La table de routage contient pour chaque route structurelle l'adresse de la PES vers laquelle le message doit être dirigé. Dans le cas où le message est arrivé dans le processeur destinataire, la valeur contenue dans cette table indique que l'adresse de la porte destinataire est à prendre dans le champ RTDEST du message.

Dans le cas du bus physique si le message est un message global, il est diffusé vers toutes les PES des processeurs connectés au bus. Si c'est un message de contrôle pour le bus physique, il permet le transfert des PES des processeurs connectés au bus vers l'arbitre et vice-versa.

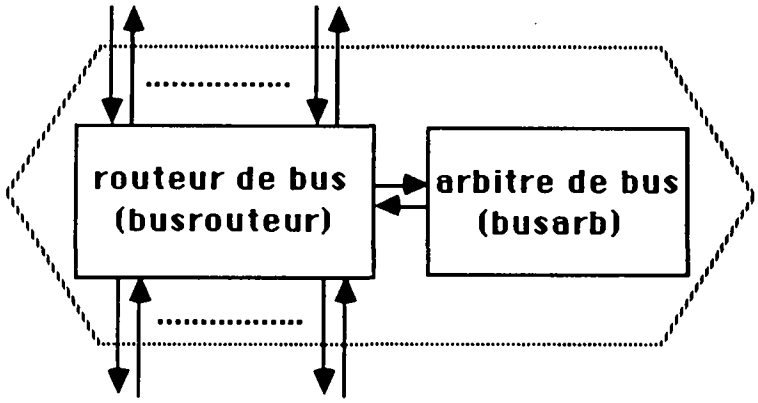

Fig. 17 Bus physique

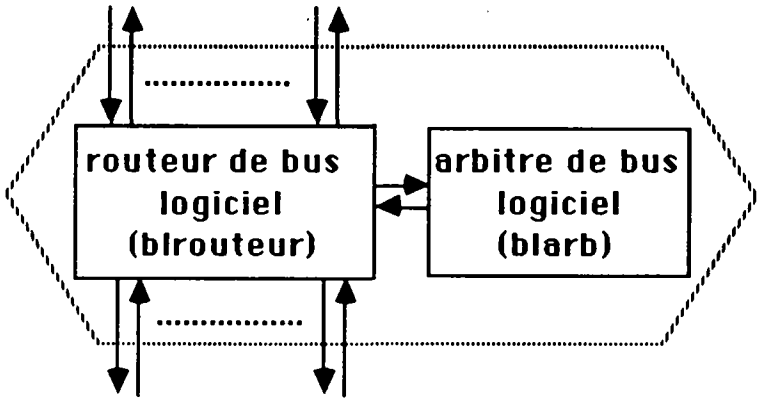

Fig. 18 Bus logiciel

2.4. Processeur programmable (PROCR)

L'objet processeur programmable est constitué d'un bus logiciel et d'autant de portes d'entrée - sortie que de liens reliant ce PROCR avec l'extérieur.

IV Génération de code et évaluation de performances

1. Génération de code

Nous avons choisi de générer du code OCCAM pour construire nos simulateurs, en effet ce langage se prête bien à la description d'applications parallèles flot de données, de plus il s'exécute directement sur un processeur le TRANSPUTER qui à été conçu sur mesure. Ce dernier se connecte simplement à d'autres processeurs du même type pour former une machine homogène multi-processeur et possède un exécutif temps réel cablé qui est utilisé directement à travers le langage OCCAM.

Un second choix a été fait, l'application de traitement du signal que l'on a programmée sur une machine (cf. chapitres précédents), est simulée sur une machine de simulation mono-transputer et s'exécute en temps réel sur des machines cibles multi-transputer pour l'instant.

2. Evaluation de performances

2.1. Méthode d'évaluation

Le matériel de développement INMOS est constitué d'un PC/AT muni d'une carte B004 comprenant un transputer T414 avec 2 Mo de mémoire et d'un rack ITEM muni d'une carte B003 comprenant quatre transputers T414 avec 256Ko de mémoire chacun.Tous les T414 fonctionnent à une fréquence de 15 MHz et les liens à une fréquence de 10 MHz.

Le système de développement TDS est chargé sur la B004. Il permet la compilation, le chargement et le contrôle des programmes s'exécutant sur la B003.

Les processus s'exécutent sur le transputer selon deux niveaux de priorité. En haute priorité les processus sont ordonnancés uniquement sur les opérations d'entrée-sortie, de plus en basse priorité, un processus ne peut s'exécuter pendant plus d'une tranche de temps. Les processus de haute priorité peuvent interrompre ceux de basse priorité.

Chaque transputer possède deux timers un pour chaque niveau de priorité, dans le cas du niveau de priorité le plus bas (resp. le plus haut) la période du timer est de 64µs (resp. 1µs). Pour avoir la résolution la plus fine on a utilisé le timer de haute priorité.

La méthode de mesure dans le cas d'un protocole synchrone sans réservation de route est la suivante:
dans l'activité élémentaire émétrice on lit la valeur du timer (tsart) avant l' envoi d'un message de données vers l'activité élémentaire réceptrice, puis on lit la valeur du timer (tfin) à réception du message de réponse. La différence (tfin-tstart) mesure donc le temps de transfert d'un message entre activités élémentaires.

2.2. Mesures

2.2.1. Entre deux transputers reliés par un lien

Nous présentons dans ce tableau cf. tab. 2 quelques mesures de temps pour la transmission d'un message de données de taille 1, 10, 20 et 30 octets (les temps sont donnés en microsecondes, il n'y a pas réservation de route) :

1 octet	10 octets	20 octets	30 octets
10	10	10	11
11	30	50	71
4324	5944	7750	9562
755	784	812	849
365	386	408	433

Tab. 2 Mesures des temps de transfert entre deux transputers

(c1) La première ligne du tableau correspond au cas de la transmission directe du message entre l'activité élémentaire émettrice et l'activité élémentaire réceptrice sur un TRANSPUTER (par un canal OCCAM).

(c2) La deuxième ligne du tableau correspond au cas de la transmission directe du message sur un lien entre deux TRANSPUTERS (canal OCCAM placé sur un lien).

(c3) La troisième ligne correspond au temps de trajet PE-BL-PES-lien-PES-BL-PR pour le message de données plus le temps de trajet PR-BL-PES-lien-PES-BL-PE pour le message d'acquittement entre deux TRANSPUTERS, le bus logiciel étant vu comme un bus physique (diffusion des messages).

(c4) Dans la quatrième ligne le bus logiciel est celui défini en 2.3 avec une politique d'arbitrage du bus programmable par l'utilisateur.

(c5) Enfin dans la cinquième ligne, une politique figée à priorité tournante est implémentée au niveau du processus blrouteur ce qui permet de s'affranchir des messages de contrôle du bus logiciel.

On a tracé un graphique donnant le temps de traversée en fonction des lignes du tableau ci-dessus cf fig. 19.

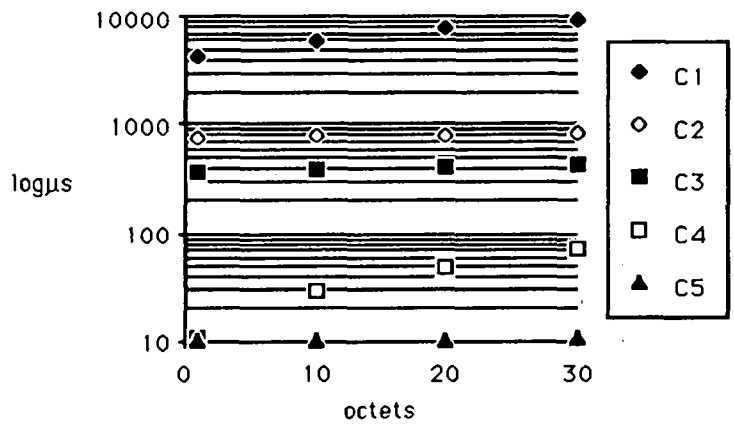

Fig. 19 Mesures entre deux transputers

2.2.2. Sur une route

Dans le dernier cas du paragraphe précédent, une autre série de mesure a été faite dans le cas où on traverse zéro, un et deux processeurs intermédiaires connectés par des liens. Pour n = 2, 3, 4 qui est le nombre total de transputers utilisés on a tracé un graphique donnant le temps de traversée en fonction de la longeur du message cf. fig. 20 et inversement le temps de traversée pour un message de 1, 10, 20, 30 octets en fonction du nombre de processeur cf. fig. 21.

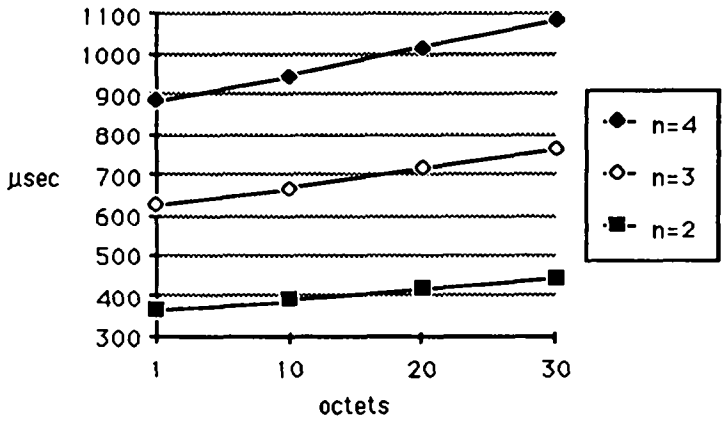

Fig. 20 Mesures sur une route en fonction de la longeur du message

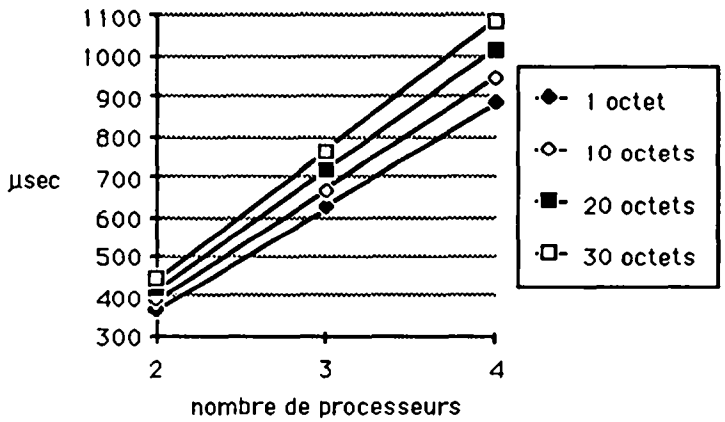

Fig. 21 Mesures sur une route en fonction du nombre de processeurs

2.3. Interprétation des mesures

2.3.1. Entre deux transputers reliés par un lien

Le tableau ci-dessous cf. tab. 3 présente pour chaque courbes de la fig. 19 le coût initial de communication qui correspond pour les courbes C1 à C3 à l'over-head système introduit par les objets systèmes BL, PE, PR, PES et la pente de chaque courbe. L'importance de cette pente dans le cas C1 nous a conduit à renoncer à la diffusion des messages sur le bus logiciel et à la remplacer par un mécanisme d'adressage direct. Dans le cas C2 le surcoût du aux messages de contrôle du bus logiciel s'évalue à 1µs par octet transmis. Dans le cas C3 on met en évidence l'over-head système qui est de l'ordre de 365-11 = 354µs, la différence de pente avec C4 correspond à l'over-head système en µs par octet, elle est minime parcequ'elle est introduite par des communications internes entre activités systèmes.

	coût initial(µs)	µs/octet
C1	4324	182.62
C2	755	3.24
C3	365	2.34
C4	11	2.07
C5	10	0.04

Tab. 3 Over-head système

2.3.2. Sur une route

Le tableau ci-dessous cf. tab. 4 présente, pour un (n = 3) et deux (n = 4) transputers traversés, le coût en µs/octet de la communication et le coût en µs/octet/T414 de la traversée d'un transputer (dans les mêmes conditions que la courbe C3).

	µs/octet	µs/octet/T414
n=4	7.03	2.21
n=3	4.83	2.17
n=2	2.34	-

Tab. 4 Coût de traversée en fonction du nombre de transputers

V Annexes

1. Un exemple de répartition manuelle : implantation d'un algorithme d'égalisation adaptative sur une machine à deux transputers

1.1. Spécification SIGNAL de l'algorithme

L'application traitée est l'identification d'un filtre transversal avec un égaliseur transversal adaptatif [11]. Un processus génère un signal pseudo aléatoire qui est envoyé d'une part sur le filtre à identifier et d'autre part sur le filtre adaptatif, on calcule une erreur en faisant la différence entre la sortie estimée par le filtre adaptatif et le signal sortant du filtre à identifier. Cette erreur sert à calculer à l'aide d'un algorithme de gradient stochastique les coefficients du filtre adaptatif. On visualise en temps réel sur un écran graphique l'erreur afin de vérifier la convergence.

Le programme SIGNAL de l'application est le suivant :

```
egaliseur {!ervisu}
=
(|gensig ?zi !zo sig
 |zi:=zo$1
 |wind ?sig !ow
 |winda ?sig !owa
 |filt ?ow !ofilt
 |filta ?owa zcoef !ofilta
 |err ?ofilt ofilta !er
 |gain ?er !erp
 |adap ?owa z-coef erp !coef
 |z-coef:=coef$1
 |visu ?er !ervisu
 |)
```

1.2. Utilisation de SYNDEX

Ce programme, décrit à l'aide de l'environnement SYNDEX, correspond au graphe logiciel orienté qui apparaît sur le bas de la figure 22. Il est dessiné en pointillé car il a compètement été placé sur la machine construite à l'aide de trois transputers. Cette machine correspond au graphe matériel non orienté qui apparaît sur le haut de la même figure.

Le graphe logiciel exprime des relations de précédence entre des actions à accomplir. Celles ci s'exécutent en parallèle, selon une règle flot de données.

Dans cet exemple le graphe n'est pas conditionné (pas de sous-échantillonnage et de mélange de signaux), une conséquence importante est que les conditionnements ne modifient pas la forme de ce graphe en coupant éventuellement des dépendances (tous les signaux ont la même horloge). Ainsi c'est toujours le même graphe que l'on traite lors du déroulement des instants logiques de SIGNAL.

Certains des noeuds du graphe sont des fonctions immédiates de SIGNAL, les noeuds de diffusion Di servent à matérialiser la diffusion qui est native en SIGNAL et qu'il faut traiter ici séparément. Enfin les noeuds retard Ri qui sont les $ de SIGNAL, représentent la ressource mémoire allouée aux processeurs. Tous les noeuds font appel à des processus de calcul, des processus de diffusion et des processus retard.

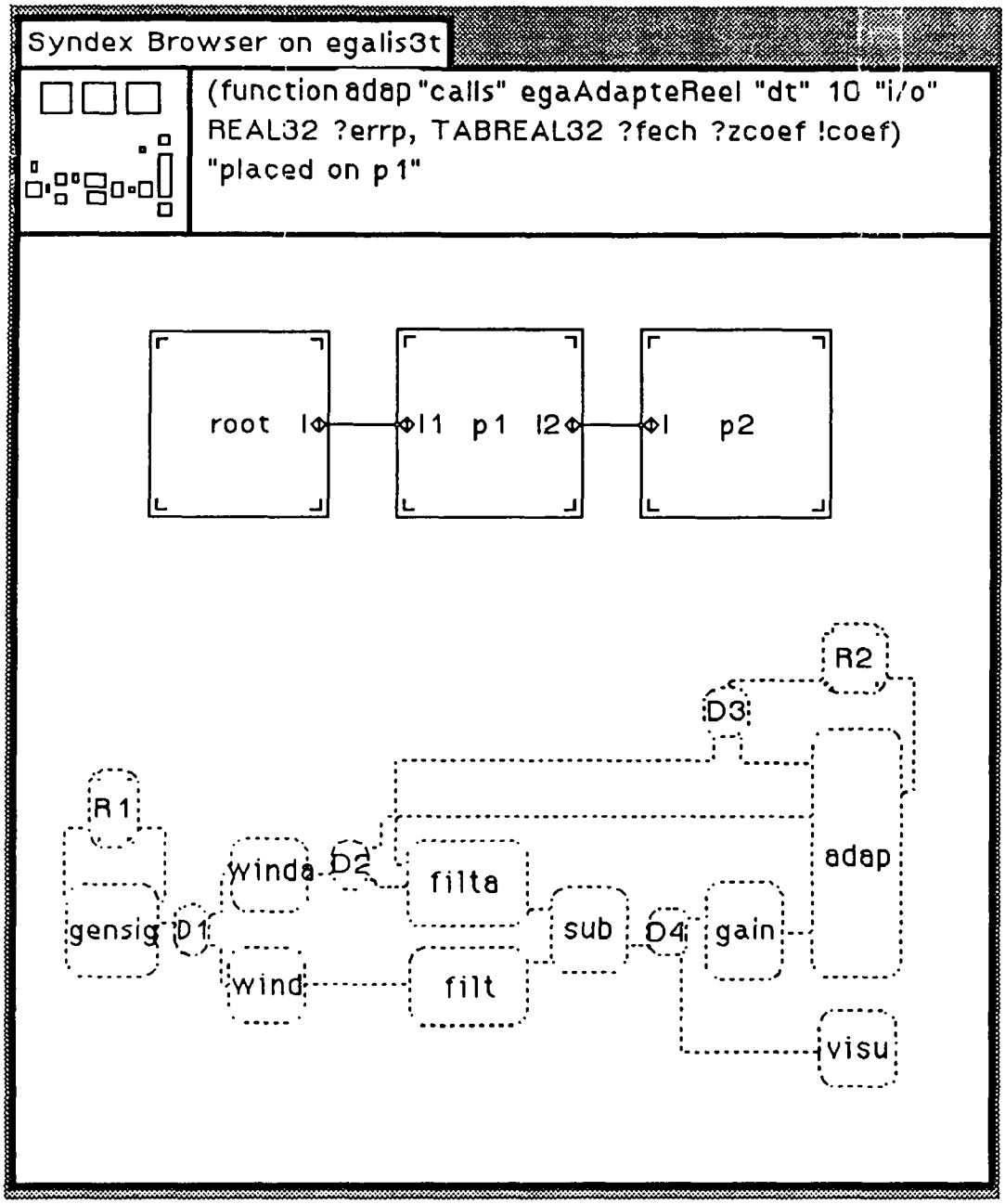

Fig. 22 SYNDEX Browser sur l'égaliseur adaptatif

La règle d'activation flot de données des processus correspond une séquence d'actions de la forme générale suivante :

Lecture de toutes les entrées
Appel d'un processus de calcul
Ecritures de toutes les sorties

On décrit brièvement la partie calculs de ces noeuds afin de détailler l'algorithme donné au début du paragraphe.

Gensig est un processus récursif qui génère un signal pseudo aléatoire par multiplication de nombres premiers entre eux combinés à la valeur trouvée précédemment.

Wind et winda sont des fenêtres glissantes, pour chaque échantillon d'entrée elles produisent un vecteur décalé de une à k unités dans le temps pour i=1 à k, on a :

$$X_{n-i-1}=X_{n-i}.$$

Filt et filta sont des filtres transversaux dont les coefficient sont respectivement fixes et variables (fournis par le processus d'adaptation) pour i=1 à k nombre de coefficients du filtre, on a :

$Y_n=\Sigma_i H_i X_{n-i}$.

On a choisi de ne pas utiliser une seule fenêtre diffusée sur les deux filtres fixe et adaptatif, afin de permettre des longueurs différentes pour ces filtres.

Adap est le processus d'adaptation des coefficients selon un algorithme de gradient stochastique pour i=1 à k on a : $H_i=H_{i-1}+erp.X_i$

Sub est le processus qui calcule l'erreur par différence entre le signal après filtrage adaptatif et le signal après filtrage, on a : e=Ya-Y

Gain est le processus qui calcule l'erreur pondérée erp, p étant une constante de gain de valeur inférieure à 1 on a : erp=e.p.

Les processus diffusion effectuent simplement une série d'affectations de l'entrée sur toutes les sorties

Les processus retard de une unité Ri sont de la forme $Y_n= X_{n-1}$, la valeur du signal de sortie est égale à sa valeur précédente. Ils fonctionnent différemment des autres processus, on doit dans leur cas, produire tout d'abord les sorties, puis consommer les entrées.

SYNDEX permet d'effectuer le placement des calculs et des communications ce qui conduit à la programmation de la machine. On peut soit effectuer ces opération de façon automatique en utilisant le programme d'optimisation automatique soit manuellement.

On se place dans le cas ou la machine est imposée. Le graphe logiciel correspondant au programme d'application décrit en SIGNAL a été placé manuellement sur les trois processeurs de la manière suivante :

sur Root le processus visu,

sur P1 les processus gensig, R1, D1, wind, filt .

sur P2 les processus winda, D2, filta, sub, D4, gain, adap, R2, D3,

Cette répartition conduit aux trois communications inter-processeurs suivantes :

D1-winda, filt-sub, visu-D4.

Ces trois communications externes ont été placées manuellement respectivement sur les deux routes suivantes :

P1-l2-P2 et root-l-P1-l2-P2.

Syndex permet de générer automatiquement le code OCCAM chargé sur la machine cible à trois transputers. En particulier il génére le code système nécessaire à la prise en compte des communications inter-processeurs, la gestion de la mémoire utiles pour éviter les interblocages ainsi que la mémoire application qui apparaît à travers les retards de SIGNAL.

1.3 Résultats obtenus en temps réel

La machine ainsi progammée fonctionne en temps réel, le signal d'erreur est visualisé sur un moniteur graphique connecté à un poste de travail PC contenant le transputer racine du réseau (processeur root) très simple dans cet exemple puisqu'ilest formé formé avec P1 et P2.

La figure 23 montre la convergence de l'algorithme de gradient pour 200 itérations, au bout de 100 itérations l'erreur résiduelle est très faible.

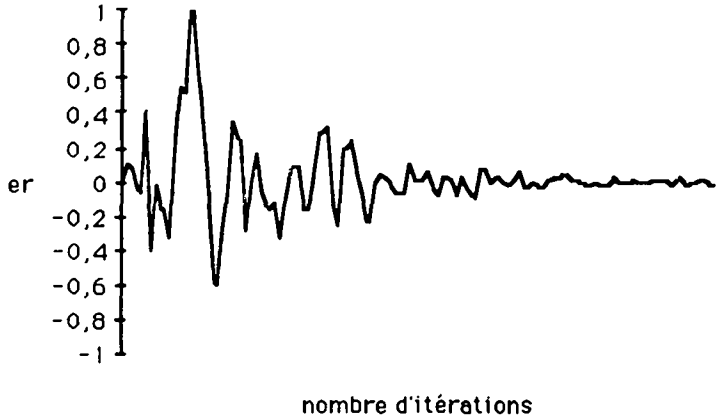

Fig. 23 Valeur de l'erreur en fonction du nombre d'itérations

2. Un exemple de placement automatique : implantation d'une FFT sur une machine à 8 transputers

Avant placement, le nombre de processeurs était imposé (il aurait pu être calculé par l'optimiseur), mais les processeurs n'étaient pas connectés. Le placement automatique s'est déroulé en trois étapes :

1) répartitions des activités de calcul sur les processeurs
2) répartition des communications inter-processeurs, qui donne lieu :
   - à la création des connexions physiques inter-processeurs [4]
   - à la création des routes nécessaires
3) calcul des tailles des tampons de communication associés à chaque connexion physique.

Les fenêtres "temporelles" de droite voir fig. 24, présentent le résultat du placement, avec une colonne par processeur et le temps de haut en bas. Comme pour les fenêtres "topologiques" de gauche, la grande fenêtre du bas affiche la partie zoomée définie par le rectangle en pointillé dans le diagramme total représenté shématiquement dans la petite fenêtre du haut. Les barres noires représentent les temps de communication inter-processeurs.

Le menu au centre présente la liste des noms de toutes les routes (menu obtenu par l'option "route" +shift du menu obtenu par l'option "hardware" du menu de la fenêtre texte). Les noms sont ceux générés par le placement automatique. La fenêtre texte affiche les informations relatives à la route sélectionnée (définition textuelle et communications inter-processeurs qui empruntent cette route).

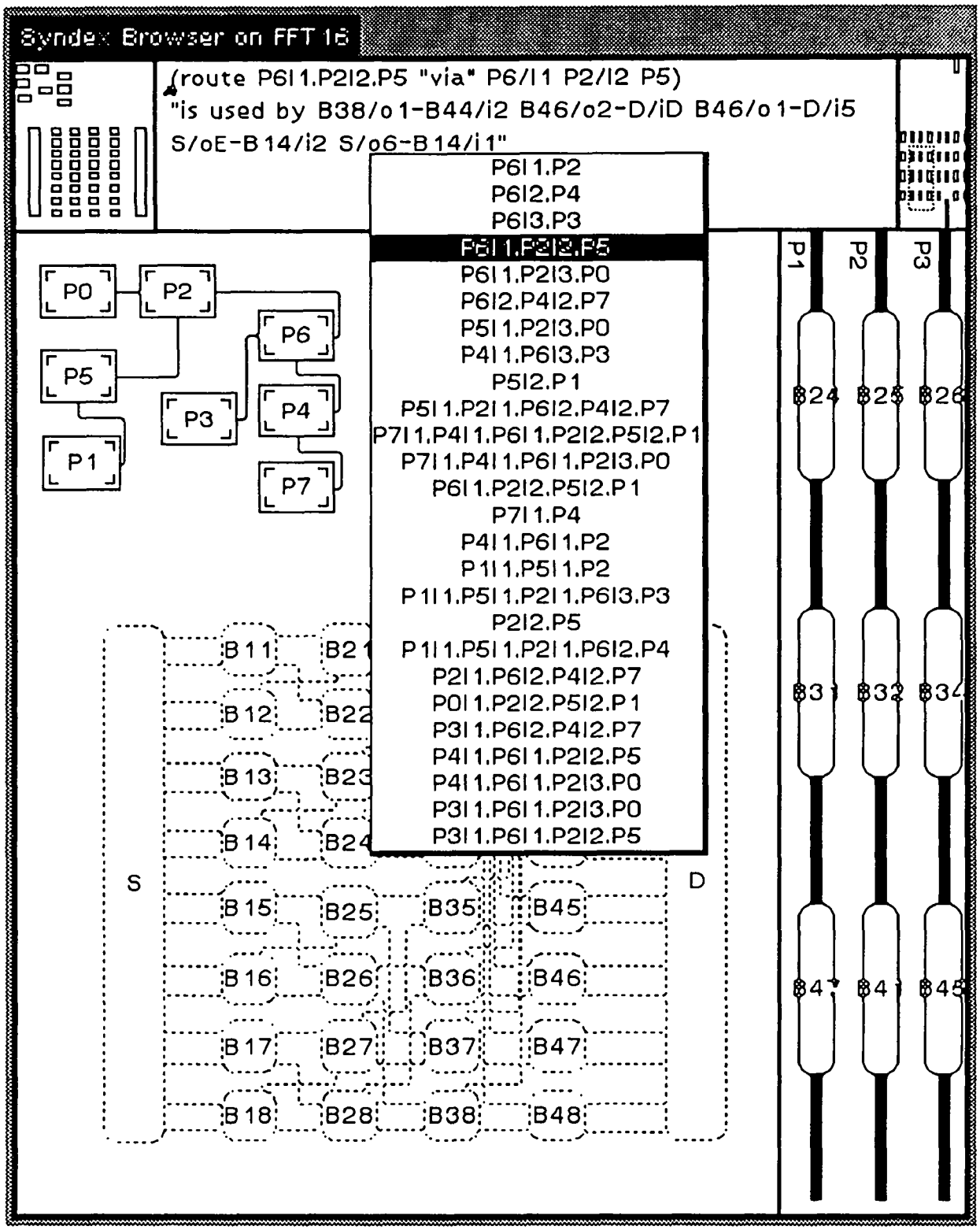

Fig. 24 SYNDEX Browser sur la FFT

3. Un exemple de machine complexe : le TNODE

La machine décrite voir fig. 25, représente une des quatre "workerboard" que peut contenir un Tnode [12]. Chaque "workerboard" comprend 8 processeurs Ti comportant chacun une interface

avec le "Control Bus System", un T800 dont les quatre liens sont reliés au "switch crossbar" et 1 à 4 Mo de mémoire.

Le "memory server" et le "disk server", reliés également au "Control Bus System" et au "switch crossbar" ne sont pas représentés.

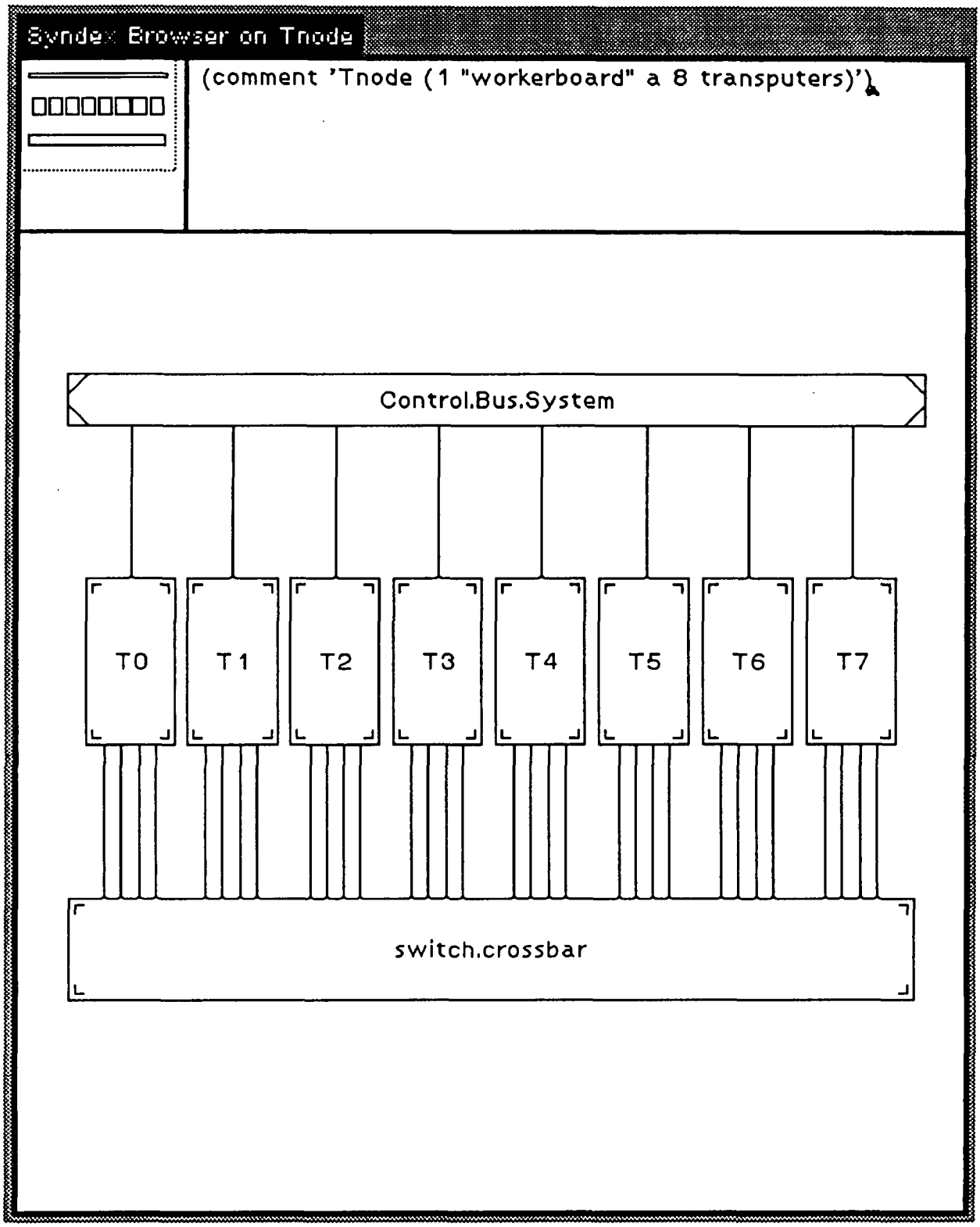

Fig. 25 SYNDEX Browser sur le TNODE

4. Code SYNDEX généré pour l'application égalisation adaptative

Ce code est la représentation textuelle de la machine "egalis3t".

C'est une sauvegarde "en clair" de toutes les informations nécessaires à la reconstitution de la représentation mémoire interne manipulée par le "browser Syndex", qui permet la représentation et l'édition graphiques de la machine, ainsi que la génération de code compilable (code OCCAM dans la version actuelle de SYNDEX).

La section "Graphic positions", ne présentant pas d'intérêt particulier, a été abrégée.

La syntaxe de cette représentation textuelle est décrite dans [manuel Syndex].

```
"machine egalis3t (20 March 1990 5:04:55 pm ) SYNDEX v.0"
  (comment 'egaliseur adaptatif reel place sur trois transputers')

"SOFTWARE SECTION"
  (iterations 100)
  (protocol COMPLEX "is" REAL32;REAL32)
  (protocol TABREAL32 "is" :INT<100:REAL32)
  (delay R1 "dt" 2 "i/o" INT "?i !o :init=" '0')
  (function gensig "calls" gensig "dt" 10 "i/o" INT ?zi !zo, REAL32 !o)
  (broadcast D1 "dt" 2 "i/o" REAL32 "?i" !o1 !o2)
  (function wind "calls" fenetre "dt" 10 "i/o" REAL32 ?i, TABREAL32 !o)
  (function winda "calls" fenetre "dt" 10 "i/o" REAL32 ?i, TABREAL32 !o)
  (function filt "calls" filtreReel "dt" 10 "i/o" TABREAL32 ?i, REAL32 !o
:coef='[0.0(REAL32), 0.0(REAL32), 0.0(REAL32), 0.0(REAL32), 1.0(REAL32),
0.0(REAL32), 0.0(REAL32), 0.0(REAL32), 0.0(REAL32)]')
  (function filta "calls" egaFiltreReel "dt" 10 "i/o" TABREAL32 ?zcoef
?fech, REAL32 !o)
  (broadcast D2 "dt" 10 "i/o" TABREAL32 "?i" !o1 !o2)
  (function sub "calls" moins "dt" 10 "i/o" REAL32 ?i1 ?i2 !o)
  (broadcast D4 "dt" 2 "i/o" REAL32 "?i" !o1 !o2)
  (function visu "calls" FLROUTREAL32 "dt" 10 "i/o" REAL32 ?i
:param='fs,ts, NbIterations, clock, "erreur"')
  (function gain "calls" multp "dt" 10 "i/o" REAL32 ?i !o
:g='0.05(REAL32)')
  (function adap "calls" egaAdapteReel "dt" 10 "i/o" REAL32 ?errp,
TABREAL32 ?fech ?zcoef !coef)
  (delay R2 "dt" 10 "i/o" TABREAL32 "?i !o :init=" '1::[0.0(REAL32)]')
  (broadcast D3 "dt" 10 "i/o" TABREAL32 "?i" !o1 !o2)
  (connect gensig/zo R1/i)
  (connect gensig/o D1/i)
  (connect D1/o2 winda/i)
  (connect D1/o1 wind/i)
  (connect wind/o filt/i)
  (connect winda/o D2/i)
  (connect filt/o sub/i2)
  (connect filta/o sub/i1)
  (connect D2/o2 adap/fech)
  (connect D2/o1 filta/fech)
  (connect sub/o D4/i)
  (connect gain/o adap/errp)
  (connect adap/coef R2/i)
  (connect D3/o2 adap/zcoef)
  (connect D3/o1 filta/zcoef)
  (connect D4/o2 visu/i)
  (connect D4/o1 gain/i)
  (connect R2/o D3/i)
  (connect R1/o gensig/zi)

"HARDWARE SECTION"
  (processor p1 "i/o" PPL l1=10 l2=10)
  (processor p2 "i/o" PPL l=10)
  (processor root "i/o" PPL l=10)
  (connect p1/l2 p2/l)
```

```
      (connect root/1 p1/11)
      (route root1.p112.p2 "via" root/1 p1/12 p2)
      (route p112.p2 "via" p1/12 p2)

    "PLACEMENT SECTION"
      (placeact "on" p1 "partition" R1 gensig D1 wind filt)
      (placeact "on" p2 "partition" winda D2 R2 D3 filta sub D4 gain adap)
      (placeact "on" root "partition" visu)
      (placecom "from D1/o2 to" winda/i "on" p112.p2)
      (placecom "from filt/o to" sub/i2 "on" p112.p2)
      (placecom "from D4/o2 to" visu/i "on" root1.p112.p2)

    "GRAPHIC POSITIONS SECTION"
      (position gensig "frame" 56 150 680 779 "zi" 1.18421 56 698 52 621 78
"zo" 2.18421 "o" 2.41515)
      (position D1 "frame" 165 203 702 758 "i" 1.33333 165 721 158 721 150 "o2"
2.14583 "o1" 2.8125)
      ...
      (position p1 "frame" 389 582 96 296 "11" 1.50119 389 196 351 196 326 "12"
2.5)
      (position p2 "frame" 641 836 96 296 "1" 1.50239 641 196 611 196 582)
      (position root "frame" 141 326 98 294 "1" 2.5)
    "That's all folks !!"
```

5. Code OCCAM généré pour l'application égalisation adaptative

Ce code a été généré pour une machine cible à trois transputers.

Pour une machine multi-transputers, le code généré pour chaque transputer doit être compilé séparément donc généré dans un fichier séparé. Les différents fichiers générés ont été regroupés et abrégés pour ne présenter que les parties dignes d'intérêt.

Ce code représente uniquement l'exécutif de la machine programmée, qui fait appel aux processus du noyau et du programme d'application.

Le code des processus du noyau (processus de communication PES, BL, PE et PR pour chaque protocole) n'apparaît pas, car il est compilé séparément dans les librairies système.

De même, le code des processus du programme d'application n'apparaît pas, car il est compilé dans la librairie d'application.

Le code généré automatiquement est commenté par de multiples références croisées et présente sous forme symbolique toutes les constantes (utilisées principalement pour le routage des communications).

```
    ------------------------------------------------------------
    -- machine egalis3t (20 March 1990 5:06:35 pm ) SYNDEX v.0
    -- egaliseur adaptatif reel place sur trois transputers
    ------------------------------------------------------------
    #INCLUDE "hostio.inc" -- contains SP protocol
    PROC egalis3t (CHAN OF SP fs, ts, []INT memory)
      #USE SYNDEX -- system libraries (PES,BL, PE and PR for primitive data
types)
      -- #USE INT -- primitive data type
      -- #USE REAL32 -- primitive data type
      #INCLUDE TABREAL32.inc -- (protocol TABREAL32 "is" :INT<100:REAL32)
      #USE TABREAL32.lib -- PE and PR library
      #USE egalis3t -- application library

      VAL NbIterations IS 100 (INT):

      -- destIDs in Procr p1 -- wind R1 D1 filt gensig --
      VAL p111 IS 0 (BYTE): -- pp1 >< root1
      VAL p112 IS 1 (BYTE): -- pp1 >< p21
      VAL D1o2 IS 2 (BYTE): -- PE >  2:2:D >windai/p2> 3:2:S
      VAL filto IS 3 (BYTE): -- PE >  2:3:D >subi2/p2> 3:3:S
```

```
      -- destIDs in Procr p2 -- adap sub winda D2 filta gain D3 D4 R2 --
      VAL p2l IS 0 (BYTE): -- ppl >< p112
      VAL D4o2 IS 1 (BYTE): -- PE >  1:1:D >visui/root> 0:1:S
      VAL windai IS 2 (BYTE): -- PR < D1o2/p1
      VAL subi2 IS 3 (BYTE): -- PR < filto/p1

      -- destIDs in Procr root -- visu --
      VAL rootl IS 0 (BYTE): -- ppl >< p111
      VAL visui IS 1 (BYTE): -- PR < D4o2/p2

      -- inter-processor routeIDs --
       VAL rootl.p112.p2.dir IS 0 (INT16): -- route root -> p2 = ( root.l
11.p1.12 1.p2 ) --
      VAL rootl.p112.p2.rev IS 1 (INT16): -- route root <- p2, reverse route --
      -- 0:1:S   visui -> D4o2
      -- 1:1:D   visui <- D4o2
      VAL p112.p2.dir IS 2 (INT16): -- route p1 -> p2 = ( p1.12 1.p2 ) --
      VAL p112.p2.rev IS 3 (INT16): -- route p1 <- p2, reverse route --
      -- 2:2:D   D1o2 -> windai
      -- 2:3:D   filto -> subi2
      -- 3:2:S   D1o2 <- windai
      -- 3:3:S   filto <- subi2


      -- processor p1 -- l1><rootl l2><p2l --
      -- in machine egalis3t ---- includes wind R1 D1 filt gensig --
      CHAN OF MESSAGE p111.rootl:
      CHAN OF MESSAGE p112.p2l:
      PROC p1 (CHAN OF MESSAGE il1, ol1, il2, ol2 )
        -- fifos --
        [10]BYTE p111.fifoIn, p111.fifoOut: -- ppl >< rootl
        [10]BYTE p112.fifoIn, p112.fifoOut: -- ppl >< p2l
        -- software bus BL -- connects -- p111 p112 D1o2 filto --
        [4]CHAN OF MESSAGE iBL, oBL:

        -- delay R1 -- i<INT<gensigzo o>INT>gensigzi --
        CHAN OF INT R1o.gensigzi:
        PROC R1 (CHAN OF INT Di,
            CHAN OF INT Do )
          INT d:
          SEQ
            CHAN OF INT INIT:
            PAR -- initial value
              INIT ! 0
              INIT ? d
            SEQ clock=0 FOR NbIterations
              SEQ
                Do ! d
                Di ? d
        : -- end of delay R1

        -- function gensig -- zi<INT<R1o zo>INT>R1i o>REAL32>D1i --
        CHAN OF INT gensigzo.R1i:
        CHAN OF REAL32 gensigo.D1i:
        PROC gensig (CHAN OF INT Dzi,
            CHAN OF INT Dzo,
            CHAN OF REAL32 Do )
          INT dzi:
          INT dzo:
          REAL32 do:
          SEQ clock=0 FOR NbIterations
            SEQ
              PAR -- input
                Dzi ? dzi
              gensig ( dzi, dzo, do )
```

```
        PAR -- output
          Dzo ! dzo
          Do ! do
  : -- end of function gensig

  -- ... (broadcast D1, function wind, function filt) ...
  PAR -------- body of Procr p1 --
    PES1 ( oBL[p111], iBL[p111], il1, ol1, p111.fifoIn, p111.fifoOut )
    PES1 ( oBL[p112], iBL[p112], il2, ol2, p112.fifoIn, p112.fifoOut )
    PEREAL32 ( D1o2.PE, PE.D1o2, oBL[D1o2], iBL[D1o2],
        D1o2, p112.p2.dir, windai )
    PEREAL32 ( filto.PE, PE.filto, oBL[filto], iBL[filto],
        filto, p112.p2.dir, subi2 )
    BLrr ( iBL, oBL, [p112,p111,p112,local] )
    R1 ( gensigzo.R1i, R1o.gensigzi )
    gensig ( R1o.gensigzi, gensigzo.R1i, gensigo.D1i )
    D1 ( gensigo.D1i, PE.D1o2, D1o2.PE, D1o1.windi )
    wind ( D1o1.windi, windo.filti )
    filt ( windo.filti, PE.filto, filto.PE )
: -- end of processor p1

-- processor p2 -- 1><p112 --
-- in machine egalis3t -- includes adap sub winda D2 filta gain D3 D4 R2
CHAN OF MESSAGE p21.p112:
PROC p2 (CHAN OF MESSAGE il, ol )
  -- fifos --
  [10]BYTE p21.fifoIn, p21.fifoOut: -- pp1 >< p112
  -- software bus BL -- connects -- p21 D4o2 windai subi2 --
  [4]CHAN OF MESSAGE iBL, oBL:

  -- function sub -- i1<REAL32<filtao i2<PRREAL32<filto o>REAL32>D4i --
  -- in processor p2 --
  CHAN OF REAL32 PR.subi2:
  CHAN OF SYNC subi2.PR:
  CHAN OF REAL32 subo.D4i:
  PROC sub (CHAN OF REAL32 Di1,
      CHAN OF REAL32 Di2, CHAN OF SYNC Si2,
      CHAN OF REAL32 Do )
    REAL32 di1:
    REAL32 di2:
    INT si2: -- synchro input buffer
    REAL32 do:
    SEQ clock=0 FOR NbIterations
      SEQ
        PAR -- input
          Di1 ? di1
          SEQ
            Di2 ? di2
            Si2 ! si2
        moins ( di1, di2, do )
        PAR -- output
          Do ! do
  : -- end of function sub

  -- ... (function winda, filta, gain et adap) ...
  -- ... (broadcast D2, D3 et D4, delay R2) ...
  PAR -------- body of Procr p2 --
    PES1 ( oBL[p21], iBL[p21], il, ol, p21.fifoIn, p21.fifoOut )
    PEREAL32 ( D4o2.PE, PE.D4o2, oBL[D4o2], iBL[D4o2],
        D4o2, root1.p112.p2.rev, visui )
    PRREAL32 ( windai.PR, PR.windai, oBL[windai], iBL[windai],
        windai, p112.p2.rev, D1o2 )
    PRREAL32 ( subi2.PR, PR.subi2, oBL[subi2], iBL[subi2],
        subi2, p112.p2.rev, filto )
    BLrr ( iBL, oBL, [local,p21,local,p21] )
    winda ( PR.windai, windai.PR, windao.D2i )
```

```
      D2 ( windao.D2i, D2o2.adapfech, D2o1.filtafech )
      R2 ( adapcoef.R2i, R2o.D3i )
      D3 ( R2o.D3i, D3o2.adapzcoef, D3o1.filtazcoef )
      filta ( D3o1.filtazcoef, D2o1.filtafech, filtao.subi1 )
      sub ( filtao.subi1, PR.subi2, subi2.PR, subo.D4i )
      D4 ( subo.D4i, PE.D4o2, D4o2.PE, D4o1.gaini )
      gain ( D4o1.gaini, gaino.adaperrp )
      adap ( gaino.adaperrp, D2o2.adapfech, D3o2.adapzcoef, adapcoef.R2i )
  : -- end of processor p2


  -- processor root -- 1><p111 --
  -- ...

  PAR ------ body of machine egalis3t--
    root (  fs,ts, p111.root1,root1.p111 )
    p1 (  root1.p111,p111.root1, p21.p112,p112.p21 )
    p2 (  p112.p21,p21.p112 )
: -- end of machine egalis3t
-- That's all folks !! --
```

6. Bibliographie

[1] Hybrid dynamical systems theory and the language SIGNAL. Rapport de recherche INRIA, 1988. Benveniste A., Le Goff B., Le Guernic P.

[2] SYNDEX : l'interface avec SIGNAL, à paraître

[3] Inférence de contrôle hiérarchique ; Application au temps-réel. Thèse de doctorat, Université de Rennes I, 1989. Le Goff B.

[4] Quelques méthodes de répartition et d'ordonnancement de processus de traitement du signal sur un réseau de transputers.Thèse de doctorat, Université de Paris-Sud Orsay, 1989. Ghezal N.

[5] SYNDEX un environnement de programmation pour multi-processeur de traitement du signal. Manuel de l'utilisateur, version V.0. Rapport technique INRIA , 1989. Lavarenne C., Sorel Y.

[6] Systèmes informatiques répartis. Dunod. Cornafion. 1988.

[7] Réseaux locaux, normes et protocoles. Hermes. Rolin P. 1988.

[8] Smalltalk-80, the langage and its implementation. Addison-Wesley, 1983. Goldberg A., Robson D.

[9] INMOS. OCCAM 2 Reference Manual. Prentice Hall, 1988.

[10] INMOS. Transputer Reference Manual. Prentice Hall, 1988.

[11] Utilisation du langage SIGNAL pour la spécification et la simulation d'algorithmes de traitement du signal. Onzième colloque sur le traitement du signal. Nice 1987. Dechelle F., Sorel Y.

[12] Supernode : transputer and software. Proceedings of third international conference on supercomputing.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249 - 6399