ARMOR : ARchitecture MOdulaire Reconfigurable pour le traitement d'un flot continu de données en temps réel.

D. DOURS, R. FACCA, A. FEKI, P. MAGNAUD

Laboratoire : IRIT-CERFIA - Université Paul Sabatier - Toulouse III - 118, route de Narbonne
31077 TOULOUSE Cédex

RESUME

Dans le cadre particulier des applications traitant un flot continu de données en temps réel, le projet ARMOR, a pour objectif de permettre à un utilisateur de définir la configuration optimale de la machine parallèle qui exécutera son application en temps réel et ce, sans avoir à tenir compte des contraintes temporelles et matérielles.

Pour ce faire, nous avons développé une méthodologie de conception fondée sur l'utilisation d'un modèle formel permettant d'expliciter le parallelisme intrinsèque d'une application en terme de réseau de modules communiquant par messages. Ce modèle peut ensuite être mis en correspondance biunivoque avec une architecture qui se caractérise par une structure récursive combinant le parallèlisme vrai et le pipe-lining. C'est une architecture modulaire reconfigurable, dans laquelle des modules matériels sont interconnectés selon une configuration qui se déduit de l'application.

SUMMARY

Whithin the particular environment of applications processing a continuous data flow in real time, the ARMOR project is aimed at making it possible for a user to define the optimal configuration of the parallel machine that will perform his application is real time and this, without having to take into account on time and hardware restraints.

To this end, we have developped a designing methodology based on the use of a formal model making it possible to state the intrinsic parralelism of an application in terms of a network of modules communicating throuhg messages. This model can then be put into a biunivocal correspondance with an architecture characterized by a recursive structure combining tru parallelism and pipe-lining. It is a reconfigurable modular architecture, in which hardware modules are interconnected according to a configuration which is deducced from the application.

I. INTRODUCTION

Les études d'architectures parallèles ont pour objectif d'exploiter les spécificités d'un domaine d'applications afin d'augmenter les performances ou de réduire le coût des machines qui les traitent.

Selon les domaines, les besoins de puissance sont plus ou moins importants et les spécificités sont plus ou moins bien identifiées. C'est pourquoi certains types de machines font l'objet d'études très poussées, et ce depuis fort longtemps. C'est le cas des super-ordinateurs qui sont des machines spécialisées dans le calcul numérique. D'autres sont plus récentes. Le domaine le plus prolifique étant certainement celui de la communication visuelle. En effet, une multitude de machines spécialisées ont vu le jour, que ce soit pour la vision par ordinateur ou pour la synthèse d'images. Mais ce ne sont pas les seules, car la tendance actuelle est de s'orienter vers la définition d'architectures parallèles dédiées à une classe d'applications. Certaines sont prévues pour gérer des bases de données, d'autres sont orientées vers le calcul symbolique.

Une classe importante est celle des applications relevant du traitement d'un flot continu de données en temps réel. Ce sont des applications très courantes dans le domaine du traitement du signal, que ce soit : la reconnaissance de la parole, le traitement d'images dynamiques, le traitement d'informations spaciales, le suivi médical..., mais aussi dans la commande et le contrôle : de robots, de satellites, de centrales nucléaires... et plus généralement dans toute application où un flot de données doit être traité en un temps inférieur à une borne donnée [G.R.T.R.88].

Cette classe d'applications a la propriété d'être modélisable par un réseau de modules dont le graphe de communication entre modules a les caractéristiques d'un réseau d'activité. [DOURS 86].

Nous avons développé une méthodologie de conception fondée sur l'utilisation d'un modèle formel permettant d'expliciter le parallélisme intrinsèque d'une application en terme de réseau de modules communiquant par message. Ce modèle peut ensuite être mis en correspondance bi-univoque avec une architecture parallèle qui se caractérise par une structure récursive combinant le parallélisme vrai et le pipe-lining. C'est une architecture modulaire reconfigurable, dans laquelle des modules matériels sont interconnectés selon une configuration qui se déduit de l'application.

II. MOTIVATIONS ET OBJECTIFS

L'augmentation des performances, notamment l'accélération des temps de calcul, peut être obtenue pour un coût raisonnable avec une architecture parallèle, si l'on parvient à un rapport étroit entre la structure de l'application et l'architecture de la machine qui la traite.

La voie la plus courante consiste à adapter l'application à un système existant. C'est généralement une machine parallèle à vocation générale de type SIMD ou MIMD. Une autre voie consiste à définir une architecture adaptée à l'application. Cette deuxième approche présente des avantages quant aux possibilités d'exploitation du parallélisme puisqu'on ajuste la structure de la machine à celle de l'application. En effet, il n'existe que deux méthodes pour effectuer des traitement parallèles sur une machine : le parallélisme vrai et le pipe-lining.

Le **parallélisme vrai** nécessite que les traitements soient indépendants les un des autres pour pouvoir s'exécuter simultanément sur des unités distinctes. Alors que pour le **pipe-lining** l'exécution complète d'un traitement passe par plusieurs étapes consécutives, chaque étape étant prise en charge par une unité distincte, l'ensemble de ces unités formant un **pipe-line**. Il faut remarquer que la segmentation d'un traitement et son exécution dans un pipe-line, présente d'autant plus d'intéret que le flot de données à traiter est long.

Ces deux types de parallélisme peuvent coexister et être mis en oeuvre à différents niveaux : opération, instruction, application. Le parallélisme des opérations et des instructions qui est interne à un processeur est qualifié de **"bas niveau"** alors que la coopération de plusieurs processeurs à l'exécution d'une application correspond à un parallélisme dit de **"haut niveau"**.

Le parallélisme de bas niveau est le plus facile à exploiter car les divers traitements correspondant aux opérations et aux instructions sont clairement définis, les ressources qu'ils mobilisent et leurs temps d'exécution sont connus avec exactitude. Tous les super-calculateurs exploitent au maximum ce type de parallélisme. Par contre le parallélisme de haut niveau est plus difficile à exploiter. En effet pour utiliser ce type de parallélisme on considère qu'une application est composée de tâches. Ce partitionnement fait apparaître des relations de dépendance entre tâches qui sont dues aux transferts des données. Une tâche reçoit ses données d'entrée de ses prédecesseurs, effectue des calculs sur ces données et envoi les résultats en sortie vers ses successeurs.

L'exécution de ces tâches sur divers processeurs fonctionnant de manière indépendante pose des problèmes de synchronisation entre les processeurs qui doivent communiquer.

Les communications entre processeurs peuvent avoir lieu par variable commune ou par message. On sait que l'utilisation d'une mémoire commune simplifie les communications et les synchronisations. Mais elle n'est acceptable que pour un nombre restreint de processeurs. De même l'interconnexion complète entre les processeurs devient vite trop onéreuse lorsque le nombre de processeurs augmente. C'est pourquoi dès qu'on cherche à connecter un nombre important de processeurs, on utilise des réseaux pour effectuer les communications. On a alors une architecture distribuée dont la topologie d'interconnexion la plus courante est l'hypercube.

Un autre problème est celui de la parallélisation de l'application. En effet, s'il existe d'assez bons **vectoriseurs** pour détecter le parallélisme de bas niveau, il n'existe pas actuellement de logiciel capable de déterminer le parallélisme de haut niveau exploitable par une machine cible donnée [FEAUTRIER 88]. Pourtant toutes les applications ne recèlent pas des calculs vectoriels. L'exploitation du parallélisme de haut niveau est donc le seul recours dans ce cas-là.

Puisque projeter une application sur une machine parallèle donnée, ne permet pas d'exploiter tout le parallélisme potentiel qu'elle contient, il semble donc qu'il soit préférable d'ajuster la structure de la machine à celle de l'application. Mais pour des raisons évidentes de coût, on a intérêt à définir une **architecture dédiée** à tout une classe d'applications. C'est pourquoi de nombreuses études portent sur les possibilités de reconfiguration d'une machine parallèle. Il s'agit généralement de machines **"réseau de processeurs"** à structure d'interconnexion programmable. L'architecture Super-Node [MUNTEAN 87] est un bon exemple.

Or, si ce type de machine permet d'optimiser le temps d'exécution, il ne garantit pas une exécution en temps réel. Lorsque le temps réel est une contrainte impérative, la recherche de l'adéquation entre l'application et la machine qui la traite doit être guidée par une méthodologie de conception qui tient compte des spécificités de l'application.

III. INTERACTION ENTRE ALGORITHMIQUE ET ARCHITECTURE

Il s'agit tout d'abord de dégager les caractéristiques de la classe d'applications envisagées. Il faut ensuite chercher à modéliser les traitements en tenant compte des propriétés particulières des applications visées. Il reste à définir une architecture adaptée au modèle formel.

3.1. Caractérisation des applications visées :

Lorsqu'on s'intéresse à la réalisation d'un système traitant un flot continu de données en temps réel, il faut faire en sorte que la vitesse de traitement du système soit suffisante pour absorber le flot de données sans perte d'information. Mais il est aussi parfois crucial de s'assurer que le système réagira à un événement dans un délai inférieur à une borne donnée. Il se pose alors le problème du temps de réponse qui est en fait celui d'associer à tout algorithme son temps d'exécution.

Dans le cas général, ce problème est indécidable, mais dans le cas particulier qui nous intéresse, des solutions peuvent être apportées. En effet, le point important ici, n'est pas de connaître le temps d'exécution, mais de s'assurer que dans tous les cas de figure, le système réagira en un temps inférieur à une borne donnée. Il est clair que lorsque l'algorithme comporte une récurrence dont l'arrêt est déterminé par une relation portant sur des variables récurrentes, le nombre exact d'itérations nécessaires pour satisfaire la condition d'arrêt n'est pas connu. Mais si l'on suppose le programme déterministe, toute suite de calculs est finie et en fonction des limites des données, on peut évaluer un minorant du nombre d'itérations suffisantes.

Nous avons vu qu'une application pouvait être partitionnée en tâches. Une telle décomposition peut être décrite formellement par un graphe multi-parti qui a les caractéristiques d'un réseau d'activité. C'est un graphe à une entrée et une sortie ne contenant ni boucle ni circuit, qui visualise un ordre partiel d'exécution, induit par une relation de précédence dans le temps.

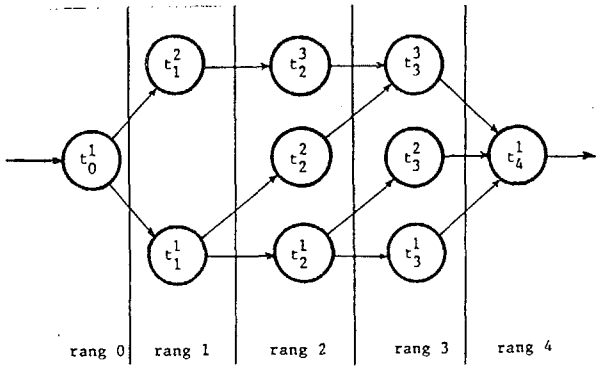

Fig.1 - Graphe de dépendance entre tâches.

Il est clair que chaque tâche peut à son tour être décomposée et représentée par un graphe de dépendance entre tâches. Cette conception progressive par affinements successifs permet d'obtenir une décomposition représentable par des réseaux d'activité imbriqués.

3.2. Modélisation des applications

Pour modéliser une application, nous allons transcrire la description fonctionnelle en terme de modules communicants. Cette transcription consiste à définir les modules de traitement associés à chaque tâche, à préciser les entrées et les sorties de chaque module et à définir les liaisons entre modules.

Un module sera considéré comme un opérateur complexe qui échange des données avec d'autres modules par l'intermédiaire de ses ports d'entrée et de ses ports de sortie.

Une liaison bipoint relie un port de sortie d'un module à un port d'entrée d'un autre module, si ce dernier utilise des données produites par le premier. Chaque liaison est munie d'une file d'attente infinie dans laquelle un module produit des résultats et l'autre prélève des données.

L'ensemble des liaisons définit un réseau de communication entre modules de façon explicite. Dans notre cas, ce réseau est un **graphe de dépendance de données** entre modules qui a les mêmes caractéristiques que le réseau d'activité des tâches fonctionnelles. Une liaison ne peut donc relier que deux modules de rangs adjacents.

Le réseau de modules ainsi décrit a un fonctionnement totalement asynchrone, car l'activation des modules est définie par la disponibilité des données dont ils dépendent. Chaque module peut à son tour être décomposé selon un graphe de dépendance et ce jusqu'au degré de finesse nécessaire. En effet, pour satisfaire aux contraintes imposées par le fonctionnement en temps réel, on peut être amené à décomposer un module. Pour effectuer cette décomposition, une série de transformations peuvent être appliquées à l'algorithme qui lui est associé pour en modifier la performance tout en conservant sa compétence. Un module qui n'est pas décomposé est appelé **module élémentaire.**

Après décomposition, on obtient une modélisation de l'application en terme de réseaux de modules communicants imbriqués. On peut montrer qu'une telle décomposition est toujours possible et on trouvera le détail des transformations dans [DOURS 86].

3.3. Architecture adaptée

Etant donné qu'il s'agit d'applications traitant un flot continu de données, il est clair que l'on a intérêt à définir une architecture modulaire ayant une structure récursive qui combine le parallélisme vrai et le pipelining qui soit isomorphe au modèle formel. De plus pour satisfaire à tout une classe d'applications, elle doit être reconfigurable ce qui nécessite un système de communication à structure adaptable.

A tout modèle formel, on fait donc correspondre un **module d'exécution,** qui peut être élémentaire ou de type réseau. Chaque module d'exécution est relié au système de communication du réseau par deux liens : un qui lui permet de recevoir des données, l'autre d'écouler ses résultats.

Un **module d'exécution élémentaire** est composé de deux processeurs, un pour effectuer le traitement des données correspondant à une tâche élémentaire de l'application, l'autre pour échanger les données avec le système de communication.

Un **module réseau** doit exécuter des traitements en parallèle. Ces traitements sont répartis sur les modules d'exécution qui le composent.

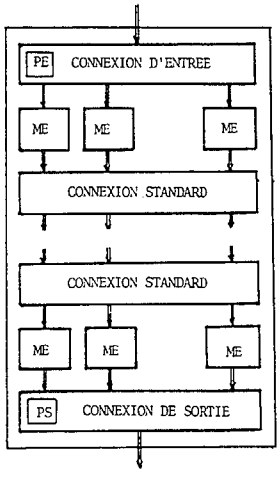

Fig 3 : Connexion standard

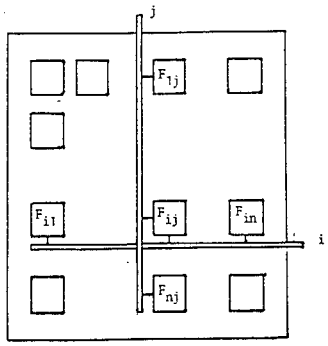

Fig 2 : Module réseau

Le **système de communication** doit permettre à tout module d'exécution de rang k de communiquer des résultats à n'importe quel module d'exécution de rang k+1. Il faut donc définir une **connexion standard** permettant les liaisons entre modules de rangs adjacents, dans laquelle chaque liaison dispose d'une file d'attente. Pour une application particulière, seules les liaisons logiques de l'application sont effectives. Un descriptif précise au processeur d'échange du module d'exécution les connexions permises.

Si n est le nombre maximum de modules d'exécution dans un rang, chaque module d'un rang donné doit pouvoir communiquer avec chacun des n modules du rang suivant. Il y a donc n^2 liaisons logiques possibles. Comme un module d'un rang k n'est relié à un opérateur de connexion que par un seul lien, il ne peut émettre que séquentiellement les résultats vers les différents modules de l'étage k+1 et ne recevoir que séquentiellement les données en provenance des différents modules de l'étage k-1. Les liaisons logiques étant multiplexées dans le temps, il n'est donc pas nécessaire qu'il y ait n^2 liens physiques comme dans un cross-bar. Si les FIFOs sont regroupées sous forme matricielle et si l'on note Fij (i,j=1,2,...,n) la FIFO de la liaison mettant en correspondance le module i du rang k avec le module j du rang k+1, il suffit que le module i soit relié à toutes les FIFOs de la ligne i par un bus commun pour qu'il puisse y disposer des résultats et que le module j de l'étage k+1 soit relié par un bus commun à toutes les FIFOs de la colonne j pour qu'il puisse y prélever des données.

Pour qu'un module réseau puisse être perçu par le système de communication comme un module élémentaire, nous avons défini une **connexion d'entrée** et une **connexion de sortie** de telle sorte que le module réseau ait le même comportement qu'un module élémentaire.

Ces connexions diffèrent de la connexion standard par le fait que les files d'attente ne sont pas directement alimentées ou n'alimentent pas directement un module. Ce sont des processeurs d'échange qui se chargent de collecter les données ou de distribuer les résultats depuis le milieu extérieur ou vers celui-ci.

Le **fonctionnement** est le suivant : une fois initialisé, chaque processeur de traitement exécute la tâche qui lui est allouée dans la phase de chargement. L'exécution de cette tâche nécessite des données que le processeur d'échange collecte dans la connexion standard amont et produit des résultats que le processeur d'échange distribue vers la connexion standard aval. Dans le cas d'un module réseau, c'est le processeur d'entrée qui approvisionne le module en données et le processeur de sortie qui écoule ses résultats.

Les transferts de données se font en parallèle avec l'exécution des traitements. Cette simultanéité, qui est possible grâce au système de communication mis en place, réduit au minimum les pertes de temps occasionnées par la gestion des communications. En effet, seules les demandes d'approvisionnement en données et de libération des résultats interfèrent avec l'exécution. Si les données issues de l'étage précédent sont prêtes au moment de la demande, l'exécution peut reprendre immédiatement.

Du fait que toutes les liaisons comportent une FIFO, les communications se font sans rendez-vous et tous les modules d'exécution sont asynchrones. Les synchronisations logiques imposées par l'application sont résolues automatiquement car l'activation d'un module est définie par la seule disponibilité des données dont il dépend.

Le module réseau de plus haut niveau doit assurer le lien avec l'environnement de l'application. Les données en provenance de l'extérieur ou les résultats devant être communiqués, doivent transiter par des périphériques qui seront reliés à la connexion d'entrée et à la connexion de sorite par des interfaces spécifiques. C'est le périphérique relié à la connexion d'entrée qui impose la cadence au système. Quant au périphérique de sortie, il doit être suffisamment rapide pour accepter le flot de résultats. Les processeurs d'entrée et de sortie de ce module doivent donc, en plus de leur fonction d'échange de données, gérer le dialogue avec un système hôte qui se chargera d'initialiser la machine en vérifiant que tous les processeurs fonctionnent correctement et en leur distribuant les traitements qu'ils auront à exécuter. Une fois initialisée, la machine devient autonome et peut fonctionner en permanence.

4. CONCEPTION D'UNE MACHINE

Pour définir la machine adaptée à l'application, l'utilisateur dispose d'un langage lui permettant de décrire l'application selon le modèle formel [MAGNAUD 86].

C'est un langage de type procédural, hiérarchisé en deux niveaux, l'un permettant de décrire les algorithmes séquentiels des modules élémentaires, l'autre permettant la description des réseaux de modules. Ce dernier niveau peut être vu comme un langage de description d'algorithmes parallèles mais, contrairement à d'autres langages de ce type [THOMESSE 77], [BOUSSINOT 81], [COMTE 82], [LE CERTEN 85], les structures de contrôle classiques permettant d'exprimer les algorithmes séquentiels n'existent pas au niveau de la description des réseaux. Elles ne serviraient à rien car elles ne pourraient pas

être exécutées sur l'architecture ARMOR. Ceci n'est pas restrictif car on montre que l'on peut toujours exprimer une décomposition modulaire sans ces structures [DOURS 86].

Lorsque l'application est décrite il faut trouver une **configuration** modulaire qui tienne compte des contraintes imposées par une exécution en temps réel. En effet les modules fonctionnels décrits par l'utilisateur ne sont pas forcément exécutables en temps réel. Car, s'il est normal que l'utilisateur fournisse une description rendant compte du parallélisme de situation induit par la fonctionnalité de l'application, on ne peut lui imposer de tenir compte des contraintes de fonctionnement en temps réel, qui dépendent des caractéristiques de la machine.

Une fois l'application compilée, deux cas peuvent se présenter. Ou bien tous les modules élémentaires décrits par l'utilisateur s'exécutent en temps réel, ou bien il existe un ou plusieurs modules ne satisfaisant pas à cette contrainte, on doit donc les décomposer.

Le but d'une telle **décomposition** est de transformer un module élémentaire en un réseau de modules, dans lequel tout module élémentaire s'exécutera en temps réel. L'objectif est de parvenir à une décomposition minimale, c'est-à-dire comportant le moins de modules possibles. Il s'agit en fait de **paralléliser un programme séquentiel** de telle sorte que le parallélisme obtenu soit juste suffisant, de constituer des modules et de construire le réseau associé.

La parallélisation consiste à déterminer les relations de dépendance entre blocs d'instructions à partir des ensembles de Berstein [BERSTEIN 66]. Un algorithme d'ordonnancement des blocs est ensuite mis en oeuvre. La constitution des modules consiste à appliquer des transformations de programmes [DOURS 86] et à déterminer le chemin des données. Il s'agit en fait de ne garder qu'un nombre minimal de chemins, de telle sorte qu'une variable produite dans un bloc puisse transiter vers les blocs qui vont l'utiliser. La construction du réseau de modules revient ensuite à déterminer les liaisons entre modules.

A l'issue de cette décomposition, l'application est décrite par un ensemble de réseaux de modules imbriqués dans lequel tout module élémentaire s'exécute en temps réel. Il faut maintenant projeter l'application sur l'architecture ARMOR.

La **projection** consiste à optimiser la description, c'est-à-dire minimiser le nombre de modules et le nombre d'imbrications de réseaux, tout en satisfaisant à la contrainte matérielle. En effet, pour des raisons techniques, l'architecture ARMOR ne peut exécuter que des modules réseaux ayant au plus huit modules par rang. Lorsque la description de l'application ne satisfait pas à cette contrainte, on montre qu'il suffit de créer une imbrication supplémentaire [DOURS 86].

Il faut remarquer que la méthode de décomposition modulaire minimise le nombre de modules créés. Il reste donc à optimiser la structure décrite par l'utilisateur.

Des transformations, qui conservent **l'équivalence opérationnelle** entre le réseau initial et le réseau transformé, permettent la minimisation du nombre de modules et du nombre d'imbrications, tout en tenant compte des contraintes temporelles et matérielles.

A l'issue de ces transformations, on peut construire un système multiprocesseurs particulier, dont la configuration sera optimale pour l'application envisagée.

5. CONCLUSION

Avec l'arrivée sur le marché des technologies VLSI, la conception de systèmes à haute performance a considérablement évolué. On est passé de l'implantation d'un algorithme sur une architecture donnée, à une stratégie de conception structurée dans laquelle l'architecture se déduit de l'algorithme afin de satisfaire aux contraintes que le système à réaliser impose, tout en tenant compte de la technologie existante. Ces mutuelles interactions entre l'algorithme, l'architecture et la technologie nécessitent des méthodes et des outils de conception nouveaux. De nombreuses recherches sont en cours sur le sujet.

Pour notre part, dans le cas particulier des applications traitant un flot continu de données en temps réel, nous réalisons un système de **décomposition automatique**, et un **configurateur d'architecture** [DOURS 87] est en cours d'étude. En ce qui concerne la recherche de la configu- ration optimale, l'explosion combinatoire des solutions possibles rend une automatisation intégrale difficilement réalisable. Il semble donc qu'il soit préférable de faire participer le concepteur aux choix d'une solution, le système se chargeant ensuite d'en vérifier la validité.

6. BIBLIOGRAPHIE

[Berstein-66] A.J.BERSTEIN : "Analysis of programs for parallel processing", IEEE Transactions on computeurs, vol.15, n°5, pp. 757-762 - Oct. 1966.

[Boussinot-81] F.BOUSSINOT : "Réseaux de processus avec mélange équitable: une approche du temps réel", Thèse d'état, Université de Paris VIII - 1981.

[Comte-82] D.COMTE et al : "Langage d'expression et de sunchronisation de tâches pour une architecture multi array-processors", Congrès AFCET, Architecture des machines et systèmes informatiques, Lille, pp. 119-129 - 1982.

[Dours-86] D.DOURS : "Conception d'un système multiprocesseur traitant un flot continu de données pour la réalisation d'une interface vocale intelligente", Thèse d'état, Université Paul Sabatier, Toulouse III - Fev. 1986.

[Dours-87] D.DOURS et al : "Configuration d'une Machine Parallèle pour le Traitement d'un flot continu de données en Temps Réel", ARCHITECTURES PARALLELES : Bigre+Globule, n°56, Nov. 1987.

[Facca-86] R.FACCA : "Architecture parallèle pour le traitement d'un flot continu de données: application à la parole", Thèse d'état, Université Paul Sabatier, Toulouse III - Fev. 1986.

[Feautrier-88] P.FEAUTRIER : "Parallélisation et Vectorisation Automatique: Etat de l'art et recherches récentes", Journées FIRTECH, Systèmes et Télématique, Architectures futures: Programmation parallèle et intégration VLSI, Paris, 9-10 Nov. 1988.

[G.R.T.R-88] Groupe de Réflexion Temps Réel du CNRS. Le Temps Réel". TSI - Vol.7 - N° 5 - 1988.

[Le Certen-85] P.LE CERTEN : "LC3, un langage de programmation parallèle", Actes du premier colloque C3, pp. 185-205 - 1985.

[Magnaud-86] P.MAGNAUD : "Définition du langage parallèle pour l'architecture ARMOR et étude du compilateur associé", Rapport de DEA, Université Paul Sabatier, Toulouse III, Juin 1986.

[Muntean-87] T.MUNTEAN : "SUPERNODE : Une Architecture Parallèle Reconfigurable de Transputers". ARCHITECTURES PARALLELES : Bigre + Globule N° 56 - Nov. 1987.

[Thomesse-77] J.P.THOMESSE : "A new set of software tools for designing and realising distributed systems in process control", IFAC, Real time programming, pp. 47-53 - 1977.