PDS : un générateur de système de développement pour machines parallèles

Jacques Eudes

PDS : Un générateur de systèmes de développement pour machines parallèles

J. Eudes

13 Décembre, 1990

Chapitre 1

Introduction

Pour exécuter une application sur une machine parallèle, la simple étape de traduction du code source, écrit dans un langage de haut niveau, vers un code binaire exécutable par compilation et édition de liens n'est pas suffisante. Pour que l'application puisse être exécutée sur plusieurs processeurs, il faut que celle-ci soit composée de plusieurs processus.

Deux cas peuvent se présenter :

1. L'application a été conçue en un ensemble de processus.
2. L'application est spécifiée dans un langage de haut niveau qui ne fait pas apparaître un ensemble de processus à placer sur le réseau de processeurs. Dans ce cas, il est nécessaire de transformer le programme initial en un programme **équivalent** composé de plusieurs processus.

Ensuite, il faut placer les processus de l'application sur les différents processeurs que nous avons à disposition dans la machine parallèle. Ce placement a pour but de permettre une exécution la plus rapide possible.

Ces deux étapes, extraction du parallélisme et placement des processus, sont des tâches absentes du circuit de développement nécessaire à une machine séquentielle. En revanche, elles sont très importantes au niveau d'une machine parallèle puisqu'elles conditionnent les performances globales de la machine pour l'application concernée. Il est donc nécessaire de concevoir des systèmes de développement spécifiques aux architectures parallèles.

Il existe deux classes de programmeurs sur machines parallèles. La première concerne des utilisateurs spécialisés dans le parallélisme. Ceux-ci conçoivent leurs applications en fonction d'une architecture de machine donnée ; ils

connaissent à fond toutes les possibilités et toutes les lacunes de celle-ci. En utilisant cette connaissance, leurs applications sont construites de manière à ce qu'elles s'exécutent le plus rapidement possible. Nous trouvons dans cette classe des utilisateurs recherchant de hautes performances pour leurs machines, comme par exemple dans le domaine du temps réel. La seconde classe regroupe des non-spécialistes voulant écrire une fois pour toutes leur application pour la faire s'exécuter ensuite sur plusieurs architectures dont ils ne veulent pas connaître les détails. Ces utilisateurs réclament des outils réalisant les tâches d'extraction et d'exploitation du parallélisme tels que nous les évoquions en début de chapitre.

Nous nous plaçons dans le cadre des utilisateurs non-spécialistes. Notre but est de concevoir un système de développement pour de tels usagers.

Nous présentons ici les fonctions devant être réalisées par un tel outil de développement. Elles permettront à l'utilisateur de penser les algorithmes de ses applications en se détachant complètement de la machine cible. Ainsi, l'usager n'a pas à se préoccuper des configurations possibles de la machine multiprocesseur dont il dispose, ce qui lui évite d'adapter ses algorithmes à une machine particulière.

La chaîne de développement doit fournir des outils permettant l'exécution de l'application sur une machine parallèle donnée. L'automatisation de cette partie du travail de développement allège le travail de l'utilisateur qui se décharge volontairement de la recherche de la performance à l'exécution. Par exemple, pour une application de type "*temps réel*" devant s'exécuter dans des délais maxima figés, le programmeur pourra évaluer différentes architectures matérielles sans surcharge de travail de sa part.

Il faut savoir qu'actuellement, l'utilisateur pense son algorithme en fonction de la configuration matérielle dont il dispose. Cette approche ne présente plus d'intéret pour lui puisque lors d'un changement de configuration (davantage de processeurs par exemple) il doit, dans bien des cas, réécrire son application. Ceci n'est plus tolérable à l'heure où les coûts de développement de logiciels deviennent particulièrement élevés par rapport aux coûts des développements matériels.

1.1 Caractéristiques des langages visés

Nous voulons compiler des langages décrivant des réseaux de processus communiquant entre eux de manière totalement synchrone. Ces langages ont la possibilité d'exprimer de manière explicite du parallélisme, les communications étant effectuées par échange de messages et basées sur le principe du "*Rendez-Vous*".

De nombreux modèles de programmation ont utilisé ce principe ; ADA et CSP en sont deux exemples. Le modèle CSP introduit par Hoare [Hoa78], permet une expression formelle du parallélisme et des synchronisations entre les processus décrits. Cette formalisation de l'écriture permet l'application de théorèmes prouvant la correction ou la non-correction du programme, ainsi que l'équivalence entre deux programmes.

Dans le cadre de notre environnement, nous travaillons sur un langage parallèle dérivé de CSP : OCCAM[1] [PH88], avec lequel sont effectuées nos applications pratiques et pour lequel nous disposons de nombreux compilateurs et outils associés.

Occam peut manipuler deux types de données :

- les variables telles qu'elles existent dans les langages séquentiels.
- les canaux qui sont des objets de communication.

Des processus primitifs permettent la manipulation de ces objets:

- Pour les variables :

  := pour modifier le contenu d'une variable avec une expression.

- Pour les canaux :

  ! pour émettre la valeur d'une expression sur un canal

  ? pour recevoir une valeur sur un canal et la stocker dans une variable

Pour que la communication puisse se réaliser entre deux processus, ils doivent s'exécuter en parallèle et partager un canal. En basant la communication sur le principe du Rendez-vous, la gestion d'un tampon de communication entre les deux processus est évitée.

[1] OCCAM est une marque déposée par INMOS Limited.

Des processus plus complexes sont élaborés sur la base des processus primitifs à l'aide des constructeurs ci-dessous :

SEQ permet l'exécution séquentielle de processus

PAR permet l'exécution parallèle de processus

ALT permet l'exécution de l'une des alternatives suivant un choix non déterministe

WHILE permet l'exécution itérative d'un processus

IF permet l'exécution conditionnelle d'un processus

En conclusion, nous éliminons la classe des langages basés sur le concept de mémoire commune en nous focalisant sur des langages à base de processus communiquant de manière synchrone par messages. Le mode de réalisation de cette communication importe peu ; le fait qu'elle soit par mémoire commune ou par liens physiques entre processeurs ne doit pas apparaître au niveau du langage. Nous pensons que le concept de la mémoire commune est une caractéristique matérielle de la machine utilisée qui ne doit pas entrer en ligne de compte dans la définition d'un langage. Les langages séquentiels classiques peuvent être considérés comme des cas particuliers de la classe des langages qui nous intéresse ici.

1.2 Caractéristiques des machines visées

Les machines qui nous intéressent sont des machines à architecture parallèle composées de nombreux processeurs sans mémoire commune. Ces processeurs peuvent communiquer entre eux grâce à des connections spécialisées à haut débit. Ces connections peuvent être établies de plusieurs manières :

1. totalement figée. C'est le cas le plus fréquemment rencontré sur les machines actuelles.

2. reconfigurable de manière semi-statique. Ceci permet avec une seule machine de choisir au démarrage entre plusieurs topologies possibles.

3. reconfigurable par programme. Pendant l'exécution de l'application, la topologie du réseau peut être modifiée.

Notre étude est essentiellement orientée vers une machine offrant cette dernière possibilité.

Le projet ESPRIT[2] P 1085 "Supernode" qui s'est achevé dans sa premièere phase en 1988, a été proposé dans le but de concevoir un supercalculateur à faible coût, exploitant les techniques du parallélisme massif. Dans ce projet, huit partenaires étaient présents dans le consortium franco-britannique, dont quatre proviennent d'organismes de recherche publics :

- L'université de Southampton
- L'université de Liverpool
- Le Royal Signals and Radar Establishment (RSRE)
- L'Institut National Polytechnique de Grenoble (INPG) auquel est rattaché le Laboratoire de Génie Informatique.

et quatre autres du monde industriel :

- THORN EMI,
- INMOS[3] constructeur du processeur transputer.
- TELMAT
- APSYS/APTOR

Le résultat de ce projet est une famille de machines dont nous présentons au chapitre 6 la configuration la plus simple. Le lecteur trouvera une description détaillée de l'architecture de l'ensemble de la famille Supernode dans la thèse (à paraître) de M. Philippe Waille et dans [Wai90].

Une machine Supernode est un ensemble de processeurs transputers reliés entre eux par des liaisons séries à haut débit via un réseau d'interconnections. La particularité de ce réseau est d'être reconfigurable dynamiquement. Cette reprogrammation du commutateur peut avoir lieu pendant l'exécution d'une application et peut être décrite dans un langage de programmation approprié.

Dans la section suivante, nous présentons les fonctions que doit réaliser la chaîne de développement.

[2]Europeen Strategic Program for Research in Information Processing Technology
[3]INMOS et transputer sont des marques déposées par INMOS Limited

1.3 Fonctions de la Chaîne de développement

Avec les systèmes de développement pour machines parallèles actuels, le programmeur doit se plier aux contraintes suivantes :

1. Exprimer lors de la conception si son application va "tourner" sur une machine mono-processeur ou bien sur une machine multi-processeurs.

2. Si c'est une machine multiprocesseurs, il doit décrire la topologie de celle-ci (nombre de processeurs et réseau de communication)

3. Dans le cas où la machine cible est multi-processeurs, il doit gérer lui-même la notion de localité entre les processseurs, démarche indispensable si l'on ne veut pas écrouler le système de communication interprocesseurs.

4. De même l'équilibrage de la charge de tous les processeurs est un problème dépendant de la topologie de la machine.

5. Les possibilités de routage des messages entre les processeurs de nombreuses machines parallèles sont réduites ou parfois absentes. C'est l'application qui effectue le routage à travers une topologie fixée. Les voies de communication par processeur sont toujours en nombre réduit par rapport aux voies logiques nécessaires pour l'application qui est alors obligée de multiplexer ces voies physiques.

6. Le problème de routage et de multiplexage des voies de communication devient difficilement contrôlable par le programmeur lorsqu'il exploite une machine à topologie reconfigurable.

Actuellement, l'utilisateur doit donc écrire de nombreuses lignes de code qui n'ont pas de lien avec son application. Nous qualifions ces lignes de *code système*.

Le code système concernant la gestion de la machine multi-processeurs devrait être généré de manière automatique et transparente à l'utilisateur. C'est le but de notre travail.

Détaillons les fonctions que se doit de réaliser une chaîne de développement entièrement automatique. Elles peuvent se décomposer en deux grosses parties indépendantes : l'extraction du parallélisme et l'allocation des processus extraits sur une configuration de processeurs donnée.

1.3.1 Extraction du parallélisme

Cette partie du travail peut être décomposée en deux phases :

- Une transformation du programme origine vers un programme massivement parallèle.
- Une extraction des informations nécessaires à l'allocation des processus sur les processeurs d'une configuration donnée.

La première phase consiste à partitionner un programme en unités atomiques : les processus. C'est l'extracteur qui détermine le grain des processus. Cette expression explicite du parallélisme peut s'effectuer soit sur un langage séquentiel, soit sur un langage structuré à base de réseaux de processus comme OCCAM, ou alors de manière implicite comme dans les langages déclaratifs comme PROLOG. Dans notre cas, nous nous intéressons davantage au langage de réseaux de processus.

Le résultat de cette phase est :

- Un réseau de processus d'un grain prédéfini (graphe).
- Le nombre d'occurences d'instruction de type communication, test conditionnel, partie séquentielle, partie itérative, etc...

Ces dernières informations doivent permettre à l'allocateur d'effectuer le placement du réseau de processus sur le multi-processeurs.

1.3.2 Placement des processus

Etant donné le nombre toujours limité de processeurs d'une machine parallèle et le nombre indéterminé de processus des applications, il est nécessaire de répartir ces processus entre les processeurs disponibles. Cette répartition est un point critique de la chaîne de développement puisque conditionnant la performance finale de l'application.

Le but principal de l'outil d'allocation est de plier *le plus efficacement* possible le graphe des processus produit par l'extracteur sur le graphe de processeurs de la machine cible. Si la topologie de la machine est modifiable, nous nous ramenons au cas d'un réseau statique en nous plaçant entre deux pas de

reconfiguration. Nous entendons par pliage efficace, un pliage qui **minimise le temps global d'exécution de l'application**.

L'absence d'un algorithme d'allocation de complexité polynomiale permettant une allocation optimale est malheureusement connue ; ce problème est *NP complet*.

En résumé, le travail à réaliser par cette partie de la chaîne de développement consiste en deux points :

1. Choix d'une heuristique de placement et son application sur le graphe de processus produit par l'extracteur.
2. Construction de la structure de routage inter-processus nécessaire aux communications.

Les résultats de l'allocateur sont :

1. Un graphe de processus plié sur le graphe de processeurs.
2. Une couche logicielle de routage et de multiplexage des voies de communications.

1.4 Plan de l'ouvrage

Le but de notre travail est de fournir une chaîne de développement pour des machines parallèles à topologie reconfigurable ou non.

Dans le chapitre 2, nous présentons différents systèmes de programmation existant dans le milieu industriel en faisant apparaître leurs avantages et leurs lacunes. Ces lacunes nous guideront sur la voie à suivre lors de la spécification de la structure du système de développement que nous avons conçu (Parallel programming Development System). La clef de voûte de PDS est la spécification d'une machine virtuelle (Parallel Virtual Machine ou PVM) qui nous permet de rester "éloigné" le plus longtemps possible du multiprocesseur cible. Sa présentation est réalisée dans le chapitre 3.

Les trois chapitres suivants sont consacrés aux trois modules présents dans notre système :

1. L'extracteur de parallélisme (traducteur vers la machine virtuelle)

2. L'allocateur

3. Le générateur de code pour le multiprocesseur.

Au dernier chapitre, consacré au générateur de code, nous détaillons la structure d'un générateur spécifique à un multiprocesseur issue du projet ESPRIT 1085 "Supernode".

Chapitre 2

Systèmes de développement pour multiprocesseurs

Après avoir présenté les caractéristiques principales des multiprocesseurs par rapport à d'autres architectures, nous décrivons dans ce chapitre un ensemble de systèmes de développement existants. Nous nous sommes attachés dans la première partie à faire ressortir les avantages et les inconvénients de ces systèmes en termes de productivité, d'aide et de confort du programmeur ; les principaux critères retenus sont :

- L'automatisation de tâches répétitives et sans intérêt pour l'application développée, mais qui constituent un support nécessaire à un ensemble d'applications.
- La présence d'outils d'aide à la programmation (bibliothèques de routage, etc...).
- Les performances du système.
- L'interface de programmation présente pour accéder au parallélisme.

En dernière partie, nous proposons un système de développement pour machines parallèles.

2.1 Architecture cible (classe de machines)

Les architectures de machines cibles pour les systèmes de développement que nous retenons sont des multiprocesseurs constitués d'un nombre important

de nœuds sans mémoire commune et connectés par des liens de communications suivant une topologie donnée ou reconfigurable par programme. Cette topologie peut être fixe ou variable. Un nœud de ce réseau est composé au moins d'un processeur de calcul, d'une mémoire locale et d'un circuit de communication permettant une communication inter-nœuds. La composition d'un nœud peut être plus ou moins riche suivant les besoins de l'application : co-processeur flottant, co-processeur vectoriel, etc ...

L'idéal serait que chacun des noeuds puisse communiquer avec tous ceux du réseau. Malheureusement, des contraintes matérielles nous limitent en nombre de connections inter-nœuds. Comme le degré (nombre de liens) est limité en nombre, des messages doivent être routés à travers plusieurs nœuds avant d'atteindre leur destination.

Voici une liste de points caractérisant les machines parallèles qui nous intéressent ici :

- Un nombre important de processeurs.
- Une exécution asynchrone.
- Des communications par échanges de messages.
- Un coût de communication modéré (haut débit de transfert des connections inter-processeurs).
- Facilité de connection entre deux processeurs.

2.2 Comparaison avec d'autres d'architectures

2.2.1 Les architectures à processeurs vectoriels

Un exemple en est le Cray X-MP [LMM85] qui utilise des circuits intégrés à haute vitesse et des processeurs vectoriels "pipeline" pour obtenir de bonnes performances. Le pipeline divise une opération en une séquence de N étapes élémentaires. Si un pipeline est à M étages (chaque étage correspondant à une étape de l'opération), il peut y avoir M opérandes dans le pipeline, une à chaque étage. Lorsque le pipeline est plein, les résultats de l'opération réalisée par ce pipe sortent au débit de l'exécution d'une étape élémentaire. Donc, l'introduction d'un pipeline de M étages devrait en principe améliorer les performances d'une opération d'un facteur M.

Malheureusement, ce facteur M ne peut pas être élevé puisque l'on arrive rarement à décomposer une opération en plus de 10 étages. D'autre part, si le résultat d'un pipeline est nécessaire à l'exécution d'un autre pipeline, ce dernier est bloqué tant que le premier pipeline n'a pas produit son résultat. Ceci peut se concevoir comme des bulles qui se propagent dans une série de pipelines et qui limitent donc les parties de code amenées vers ces processeurs vectoriels. Amdahl [Amd67] a montré que l'accélération maximale obtenue par rapport à une machine séquentielle est égale à :

$$\lim_{M \to \infty} S_{peedup} = \frac{N}{N(1-p)+p} \simeq \frac{1}{1-p}$$

où p est la fraction de code vectorisable.

C'est à dire que l'accélération est limitée par la fraction de code que l'on ne peut vectoriser : si l'on possède une machine aux pipelines de taille infinie et que 90% du code soit vectorisable, l'accélération obtenue ne serait que de dix par rapport à la vitesse d'une machine séquentielle sans pipeline.

Malgré la faiblesse du degré de parallélisme exploité par les processeurs vectoriels, ces machines font partie des plus puissantes du marché par l'utilisation d'une technologie de circuit électronique la plus rapide du moment.

Les différences notables par rapport aux multiprocesseurs sont :

- les coûts de fabrication modérés d'une machine multiprocesseur par rapport aux calculateurs vectoriels.
- Les multiprocesseurs utilisent le parallélisme plutôt que la vitesse de l'électronique pour augmenter les performances.
- Les multiprocesseurs peuvent exploiter plus facilement le parallélisme présent dans une application.
- Lors de la programmation de processeurs vectoriels, l'utilisateur doit exprimer directement dans son code des ordres de vectorisation pour que le compilateur génère du code vectorisé.

2.2.2 Les architectures à mémoire commune

Schématiquement, ces machines sont constituées de processeurs connectés à une mémoire commune par un ou plusieurs bus à haut débit . Ces architectures sont souvent utilisées pour accroître le temps de réponse en répartissant

des processus indépendants pour qu'ils s'exécutent de manière concurrente. D'une manière générale, ces architectures sont dédiées à des processus à gros grain de parallélisme.

Considérons le cas particulier du Sequent Balance 2100, où le système d'exploitation est une variante de UNIX Berkeley 4.2BSD avec des extensions au niveau de l'ordonnancement des processus unix. Cet ordonnancement étant centralisé, un processus peut migrer sur différents processeurs au fur et à mesure de l'évolution de l'état de charge du système.

Dans cet environnement, un programme parallèle est constitué d'un groupe de processus unix. Ces processus interagissent en utilisant une bibliothèque permettant une allocation de données en mémoire commune et une synchronisation inter-processus. La réalisation de la mémoire commune est effectuée par une "*projection*" d'une région physique de la mémoire centrale dans l'espace d'adressage virtuel de chaque processus. Une fois projetée, la mémoire commune peut être allouée à des variables du programme utilisateur. Les bibliothèques fournies au programmeur permettent à celui-ci d'allouer dynamiquement des données en mémoire commune, et de gérer le parallélisme à l'aide d'extensions du *fork()* et du *wait()* d'UNIX. Les processus peuvent être synchronisés de différentes manières. Le lecteur trouvera dans [KT88] une description plus fine de ces bibliothèques et une illustration avec l'exemple du problème du voyageur de commerce.

Les différences notables par rapport aux multiprocesseurs sont :

- Sur les systèmes à mémoire partagée, tous les processeurs sont équidistants. Les coûts de communications inter-processeurs ne dépendent pas de leur répartion géographique.

- Sur les multiprocesseurs, les processeurs sont séparés par des distances variables, et les performances sont d'autant plus élevées que la distance à parcourir par les messages est réduite. Comme nous le détaillerons dans le chapitre 5, des stratégies de mapping des processus sur les nœuds du multiprocesseur ont été développées : en effet, les processus communiquant beaucoup entre eux sont placés sur des nœuds proches.

2.3 Exemples de programmation de multiprocesseurs

Dans cette section, nous décrivons quelques systèmes de programmation de multiprocesseurs :

- Intel iPSC[1]
- Helios[2]
- TDS[3]
- Inmos Stand-Alone Toolset

Avant de présenter le système de développement de chaque machine, nous effectuons une brève description des différents composants de celle-ci.

2.3.1 Intel iPSC

L'Intel iPSC/1 et iPSC/2 sont directement issus des travaux de recherches effectués au California Institute of Technology (CalTech), avec pour résultat, la fabrication d'une machine multiprocesseur nommée Cosmic Cube [Sei85, SSS88]. Un accord a été passé entre Intel et CalTech qui a permis à Intel de reprendre les résultats de ce projet pour construire une machine plus performante : l'Intel iPSC. Dans un premier temps, nous effectuerons une description des composants de cette machine qui ont la particuliarité d'être ceux que l'on peut trouver sur des machines séquentielles. Nous détaillons d'abord la structure de la version 1, puis présentons de manière succinte la version 2, en mettant en avant les améliorations effectuées.

iPSC/1 - Organisation matérielle

La machine est organisée en hypercube de degré variant de cinq à sept. Chaque nœud de cet hypercube est constitué :

- d'un processeur 80286,

[1]iPSC est une marque déposée par Intel Corporation
[2]Helios est une marque déposée par Perihelion Software Limited
[3]TDS est une marque déposée par INMOS Limited

- d'un co-processeur 80287,
- de 512Ko de mémoire dynamique,
- de 64Ko de mémoire morte,
- et d'une partie communication.

La mémoire de 512Ko est partagée en double accès entre le processeur et la partie communication. Cette partie communication est constituée de huit transceivers Ethernet dont sept sont utilisés pour des connections internœuds. Le huitième est connecté sur un bus Ethernet partagé par tous les nœuds et par une machine hôte appelée aussi gestionnaire du cube (cube manager). Cet ordinateur hôte est une machine Unix utilisée à la fois pour le développement, pour le lancement des programmes, et pour recueillir des informations sur l'état de chaque nœud.

iPSC/1 - Organisation logicielle

Le modèle de programmation retenu dans l'iPSC est celui d'un hypercube dont tous les nœuds sont connectés directement à une machine maître : le gestionnaire de cube. Tout programme consiste en un ensemble de processus qui interagissent en se transmettant des messages. Généralement, ces programmes contiennent un processus maître exécuté sur la machine de développement et des processus esclaves exécutés sur les nœuds. Le comportement typique d'un processus maître est :

1. effectuer le chargement des nœuds avec le code des processus esclaves.
2. activer les processus esclaves.
3. émettre les données à traiter vers chaque nœud.
4. traiter les résultats produits par ces mêmes nœuds.

Le langage de développement choisi dans ce sytème est le langage C standard. Toutefois, le programmeur peut utiliser des primitives fournies dans deux bibliothèques pour exprimer du parallélisme. L'une d'entre elles est dédiée à la machine hôte, alors que l'autre est réservée aux nœuds de l'hypercube. Ces libraires contiennent des primitives que nous pouvons grouper en trois catégories :

1. Primitives de gestion des messages

   Lorsque deux processus nœuds ou un processus nœud et un processus du gestionnaire du cube veulent communiquer, ils doivent tout d'abord ouvrir un canal. Comme ces processus s'exécutent de manière asynchrone, ils peuvent ouvrir `copen()` ou fermer `cclose()` le canal à des moments différents. Ces primitives sont présentes dans les deux bibliothèques.

   Les routines pouvant utiliser un canal ouvert entre la machine hôte et un nœud sont `sendmsg()` et `recvmsg()` . Ces procédures permettent à un processus de la machine hôte de :

   (a) transmettre un message vers un autre processus de la machine hôte, ou vers un processus présent sur un nœud de l'hypercube, ou vers tous les nœuds de l'hypercube (diffusion).

   (b) recevoir un message émis par un processus de la machine hôte ou un nœud de l'hypercube.

   Les nœuds de l'hypercube communiquent de manière asynchrone, deux à deux par un lien bidirectionnel ou avec la machine hôte en utilisant le bus Ethernet commun. Deux types de communication sont possibles :

   - des communications non bloquantes : `send()` et `recv()`.
   - des communications bloquantes : `sendw()` et `recvw()`.

   L'un des paramètres de ces primitives désigne, dans l'espace d'adressage du processus courant, le tampon dans lequel est placé le message à émettre. Dans le cas de communications bloquantes, ce tampon est réutilisable dès l'exécution de la primitive `sendw()`. Mais dans le cas contraire, nous devons nous assurer grâce à la primitive `status()` que le message contenu dans ce tampon a bien été émis.

   Lorsque deux nœuds ne sont pas voisins géographiquement, des nœuds intermédiaires sont utilisés pour appliquer un algorithme de routage "store-and-forward". L'activation de cet algorithme est réalisée par un mécanisme d'interruption qui pénalise le traitement en cours sur le processeur.

   Enfin, une routine appelée `syslog()` permet de faire des traces qui sont enregistrées, pendant l'exécution du programme, dans un fichier de la machine hôte.

2. Primitives d'information

   Ces routines permettent au processus courant de connaître le degré de l'hypercube `size()`, ou son identité `mypid()`, ou encore l'identité du nœud où il est chargé `mynode()`.

3. Primitives de gestion des processus

   Comme nous l'avons annoncé ci-dessus, le parallélisme ne peut être exprimé qu'au niveau des processus de la machine hôte. Ces routines vont permettre à celle-ci de contrôler l'exécution des processus nœuds.

   `load()` : permet de charger un nœud avec le code d'un processus et de le lancer.

   `lwaitall()` : force le processus courant à attendre la fin de l'exécution d'un processus nœud.

   `lkill()` : tue un processus nœud.

Avantages et inconvénients

Avantage principal : le programmeur n'a pas à se soucier du routage de ses messages à travers l'hypercube et dispose d'autre part d'une possibilité de trace des processus nœuds vers le gestionnaire du cube (`syslog()`).

Les inconvénients sont les suivants :

- Il faut que le programmeur gère lui-même ses tampons de messages lors des communications.
- Seul un processus présent sur le gestionnaire du cube peut créer des processus sur les nœuds.
- Le placement des processus sur le multiprocesseur est fait manuellement.
- Le programme s'exécutant sur les nœuds et celui s'exécutant sur la machine hôte sont stockés dans des fichiers différents, et les primitives de communication n'ont pas le même nom bien qu'un canal physique identique soit utilisé.
- Le coût de création d'un processus et celui des communications empêchent l'utilisation d'un grain de parallélisme fin.
- La possibilité de debug est limitée avec la primitive `syslog()` : celle-ci modifie le comportement global du programme puisqu'il y a conflit pour accéder à la machine hôte.

iPSC/2 - Améliorations par rapport à l'iPSC/1

La constitution d'un nœud d'un hypercube iPSC/2 est assez différente de celle présentée précédemment. Chaque nœud est constitué :

- d'un processeur 80386,
- d'un co-processeur 80387,
- de 1 à 16 Mo de mémoire locale,
- d'un module de connection directe inter-nœuds appelé *"Direct-Connect^{TM} routing Module" (DCM)*,
- et suivant les besoins, de modules de calcul vectoriels ou scalaires.

La différence essentielle par rapport à la version iPSC/1 réside dans la partie connection internœud. Chaque *DCM* supervise 8 liens bidirectionnels ayant un débit maximum de 2,8 Moctets par seconde. Les communications réalisées sont toujours asynchrones, mais les performances obtenues sont bien meilleures. En effet, le routage des messages n'affecte plus le processeur (80x86), puisque le mécanisme introduit par le *DCM* évite un traitement par interruption tel celui réalisé dans l'iPSC/1.

De plus, iPSC/2 possède une version améliorée du noyau de système minimal chargé sur chaque nœud (*Node eXecutive/2*). Celui-ci permet de gérer jusqu'à vingt processus par nœud et d'utiliser la machine entre plusieurs usagers.

Au niveau du système de développement lui-même il n'y a pratiquement pas de changement. Le lecteur est invité pour de amples informations à consulter [Int88,Int89,BR89].

2.3.2 Helios

Helios[HEL] est un véritable système distribué multi-utilisateurs permettant d'exploiter un réseau de transputers de topologie quelconque. Ce système est composé :

- d'un serveur de fichiers.

- d'un ensemble de systèmes, appelés nucleus, répartis sur le réseau de transputers.

Dans Helios [Gar87,BCE*88,EKN88], les processeurs sont considérés comme des ressources pouvant être chargées avec le système minimal (nucleus) comprenant les parties suivantes :

1. un noyau (kernel) qui gère les ressources physiques du transputer. La gestion des messages et des voies de communications physiques est effectuée à ce niveau.

2. une bibliothèque système résidante qui implémente une interface appel de procédure pour réaliser un équivalent d'appel système semblable à celui des machines séquentielles.

3. un chargeur de programme qui gère l'ensemble du code chargé sur ce processeur (code utilisateur et code système).

4. un allocateur de processus qui gère la création de tâches, et leur destruction une fois achevées.

Une tâche est constituée d'un espace d'adressage privé dans lequel s'exécute ce que nous appelons un flux de contrôle. Celui-ci est constitué d'un ensemble de processus s'exécutant sur le même processeur.

Pour ce qui est de la communication entre les tâches, Helios fournit une ressource particulière nommée *port* et des primitives (`PutMsg` et `GetMsg`) permettant de réaliser un "rendez-vous" point à point. Le noyau de routage d'Helios autorise la perte de messages. Dans ce cas, l'émetteur reçoit un accusé de réception signifiant que le message a été perdu.

Nous ne nous intéressons ici qu'à la partie philosophie de programmation dans le système Helios ; pour obtenir davantage de précisions sur la structure du système consulter [BCE*88]. Nous ne détaillons pas ici le protocole Clients/Serveur qui n'est pas utile au niveau de la programmation parallèle.

Exploitation du parallélisme algorithmique

Deux formes d'exploitation du parallélisme sont réalisables dans Helios :

à l'intérieur des tâches : Le contrôle de ces processus est entièrement laissé au matériel (Scheduler intégré dans le silicum dans le cas du Transputer). Ainsi, la bibliothèque fournie avec Helios nous permet de réaliser des appels à une fonction "Unix-like" *fork* qui lance des exécutions parallèles d'une fonction passée en paramètre. Le lancement du processus transputer est effectué après avoir fait une copie de la pile et des arguments de la fonction.

Tout accès à ce parallélisme se fait donc au niveau de la compilation du code source de la tâche, ce qui implique que les processus d'une même tâche ne peuvent être répartis sur des processeurs différents. De ce fait, l'unité d'allocation de ressources est la tâche. Cette limitation n'empêche pas le placement de deux tâches sur le même processeur.

entre les tâches : La notion de tâche forcée permet au programmeur d'exprimer une exécution parallèle de tâches. Un langage de contrôle CDL [Gar87] (Component Distributed Language) est fourni au programmeur pour décrire un programme parallèle. CDL supporte deux niveaux de description d'un groupe de tâches :

Au plus bas niveau, les tâches composantes d'une tâche forcée sont explicitement décrites. Ainsi le programmeur définit les besoins en ressources de chaque tâche et spécifie les flux sur lesquels les tâches communiquent. Les flux communs à différentes tâches sont utilisés pour les communications inter-tâches. Tout ceci décrit entièrement la structure en terme de communication de l'ensemble des tâches composant une tâche forcée.

A un niveau plus élevé, la relation entre composantes d'une tâche forcée peut être spécifiée en utilisant un langage de commande. Celui-ci contient des constructeurs parallèles simples qui permettent d'exprimer que telle ou telle tâche s'exécute en parallèle et communique de telle ou telle façon par des extensions de tube (pipe) UNIX.

Les constructeurs introduits par CDL sont inspirés de ceux de CSP, ou d'OCCAM :

- un constructeur parallèle binaire, noté ^^ permet à deux tâches de s'exécuter en parallèle sans communiquer entre elles. Le flux d'entrée de ce constructeur est réparti de manière aléatoire entre les flux d'entrée des deux tâches. De même, le flux de sortie associé à ce constructeur est un mélange des flux de sortie associés aux deux tâches.

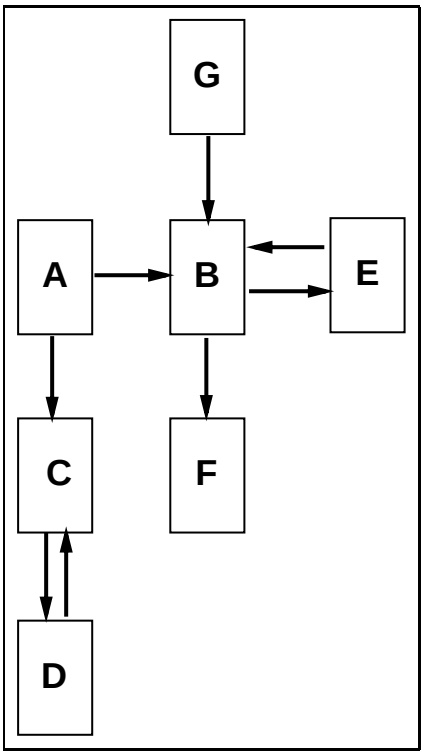


Figure 2.1: Un exemple de structure de tâche forcée

- un constructeur tube (pipe), noté | dont la sémantique est similaire à celle présente dans Unix définit une direction de communication de la première tâche vers la seconde, les deux tâches s'exécutant en parallèle.
- un constructeur tube inverse, noté |< , réalise dans le sens opposé la même opération que le constructeur tube en réalisant une communication du second processus vers le premier.
- un constructeur "subordinateur" (<>) définit des communications bidirectionnelles entre ces composantes. Un flux par direction est utilisé pour réaliser cette communication.
- un constructeur ferme (Farm), noté ||| définit lui aussi des communications bidirectionnelles. Il est utilisé avec des réplicateurs pour construire une structure de tâche connue sous le nom de ferme de processeurs : une tâche maître communiquant avec beaucoup d'autres tâches esclaves en multiplexant les mêmes flux de communications.

Le compilateur CDL fourni dans Helios gère automatiquement les allocations des flux de communications entre les tâches en fonction des constructeurs liant les composantes d'une tâche forcée.

La structure présentée en figure 2.1 est décrite simplement par la ligne CDL suivante :

```
A ( |C <> D ) |B ( |<G, |F, <>E)
```

Avantages et inconvénients

Une fois les interactions entre les tâches exprimées, Helios s'occupe du placement de telle tâche sur tel ou tel processeur. Ceci est un avantage pour le programmeur puisqu'il ne s'occupe pas du placement des tâches. Mais s'il espère utiliser au plus N processeurs, il doit "découper" son programme en N tâches. Le mécanisme de placement utilise ici les informations spécifiées lors de la conception de la tâche forcée. Les composantes ayant besoin de certaines ressources sont placées sur les processeurs ayant ces ressources. Les autres sont placées en fonction des besoins en communication (exprimés par les flux) et de la topologie du réseau. Il n'est pas précisé dans les documentations si la charge de chaque processeur est prise en compte lors du placement. Cet algorithme de placement est similaire à un partitionnement de graphe (c.f. Chapitre 5).

Le programmeur n'a pas à écrire dans son application du routage de messages, puisque celui-ci est effectué de manière transparente. En revanche, la surchage de traitement des messages est importante. Ainsi envoyer un gros message est plus efficace que de nombreux petits. La communication synchrone est beaucoup moins efficace que la communication asynchrone pour la même raison. Il vaut mieux effectuer la séquence suivante :

```
Envoie_Message(X)
travail_a_faire
Attends_Reponse(Y)
```

plutôt que celle-ci :

```
Envoie_Message(X)
Attends_Reponse(Y)
travail_a_faire
```

Helios n'est pas adapté au parallélisme massif pour les raisons suivantes :

- Le traitement des messages est trop coûteux au niveau du noyau Helios.

- Le grain de parallélisme exprimable par le programmeur et pris en compte par le système est beaucoup trop gros.
- Le parallélisme obtenu au niveau des tâches n'est pas "exportable" vers d'autres processeurs puisque non contrôlable par le système.
- Helios ne propose pas de modèle de programmation différent de celui d'Unix. Le langage CDL ne correspond pas à un modèle d'exécution supporté par le noyau, mais apporte toutefois une aide pour réaliser des programmes parallèles à partir de n tâches "Unix-like".

2.3.3 TDS

Pour pouvoir utiliser le Transputer Development System (TDS) [TDS86, Tra88], il faut posséder une machine hôte interfacée avec un transputer. En effet, le TDS est un programme s'exécutant sur le transputer, la machine hôte réalisant l'interface (écran, clavier, fichiers) nécessaire pour la programmation ; un processus serveur tourne sur la machine hôte pour réaliser ces services.

Ce système de développement est basé sur un éditeur pleine page à plis. L'avantage est la visibilité sur l'écran de la structure du programme. Les fonctions autres que celles d'édition sont accessibles à l'aide des touches spécialisées du clavier :

- vérificateur de syntaxe de programme
- compilateur
- éditeur de lien et extracteur
- "import/export" de fichiers de la machine hôte
- gestionnaire de bibliothèques
- etc ...

Le langage utilisé dans TDS est OCCAM [PH88]. Le parallélisme de grain fin est facilement exprimable par le programmeur.

Mais il n'y a pas d'outil automatique de répartition des processus sur le réseau de processeurs. TDS fournit au programmeur un langage de configuration qui est une extension d'OCCAM. Il suffit de décrire le réseau de

processeurs en effectuant le placement des processus sur tel ou tel processeur et en liant les noms logiques des canaux paramètres des processus avec l'identité des liens physiques. Aucun support de routage ou de multiplexage des voies de communications n'est fourni par TDS ; le programmeur doit prévoir lors la conception la gestion complète des liens physiques.

Une fois l'allocation effectuée par le programmeur, l'extracteur est lancé sur les processus préalablement compilés. Ce dernier utilise l'information de placement pour :

- extraire un arbre de chargement initial des processeurs du réseau à partir de la description fournie par l'utilisateur.
- extraire le code à charger processeur par processeur,
- fabriquer une image binaire contenant le code de chargement et les processus de l'application.

Le programmeur peut éventuellement utiliser des langages séquentiels classiques comme C, Pascal, Fortran, ... Cette intégration est du type récupération de binaire. Un programme harnais écrit en OCCAM gère le parallélisme entre des processus séquentiels communiquant sur des liens.

Avantages et inconvénients

Les avantages du TDS sont :

- Expression d'un parallélisme fin au niveau du langage OCCAM, qui pousse le programmeur à penser son programme en réseaux de processus et non plus en terme d'appel à des routines présentes en bibliothèques. Ceci est apporté par le langage et non par le TDS lui-même.
- TDS est bien adapté aux développements de programmes dédiés, puisque ceux-ci interagissent peu avec la machine hôte. En effet, l'accès aux ressources de la machine hôte passe toujours par le goulot d'étranglement constitué par la voie de communication machine hôte / transputeur racine (celui sur lequel tourne le TDS).

Les inconvénients :

- Le programmeur doit écrire toute la couche de gestion de routage de son application. Celle-ci n'a aucun rapport avec son programme.

- L'allocation des processus au processeur est effectuée de manière manuelle. L'unité d'allocation autorisée par le langage de configuration est à gros grain de parallélisme.
- Si la structure du réseau de processeurs est modifiée, le programmeur doit réécrire une bonne partie de l'application : la couche de routage et de multiplexage des liens doit être modifiée en fonction du nouveau découpage de l'application en processus à gros grain.
- C'est un système fermé ; le programmeur ne peut pas utiliser les services offerts par le système d'exploitation de la machine hôte. En effet, le format des fichiers foldés empêche qu'ils soient utilisables par des outils autres que ceux intégrés dans TDS.

2.3.4 Inmos Stand-Alone Toolset

Comme pour le TDS, il est nécessaire d'avoir une machine hôte interfacée avec un transputer. Les principaux outils offerts par ce système de développement [Occ87d,Occ87a,Occ87c,Occ87b] sont (c.f. figure 2.2) :

- un compilateur OCCAM
- un éditeur de liens
- un configureur (appelé aussi extracteur)
- un gestionnaire de bibliothèques

Tout ces outils s'exécutent sur le transputer et utilisent la machine hôte via des communications avec le serveur `afserver` pour obtenir les services écran, clavier, fichier, etc ...

Le programmeur peut donc utiliser toutes les facilités offertes par le système d'exploitation de la machine hôte. En effet, le format des fichiers n'est pas foldé comme dans le TDS.

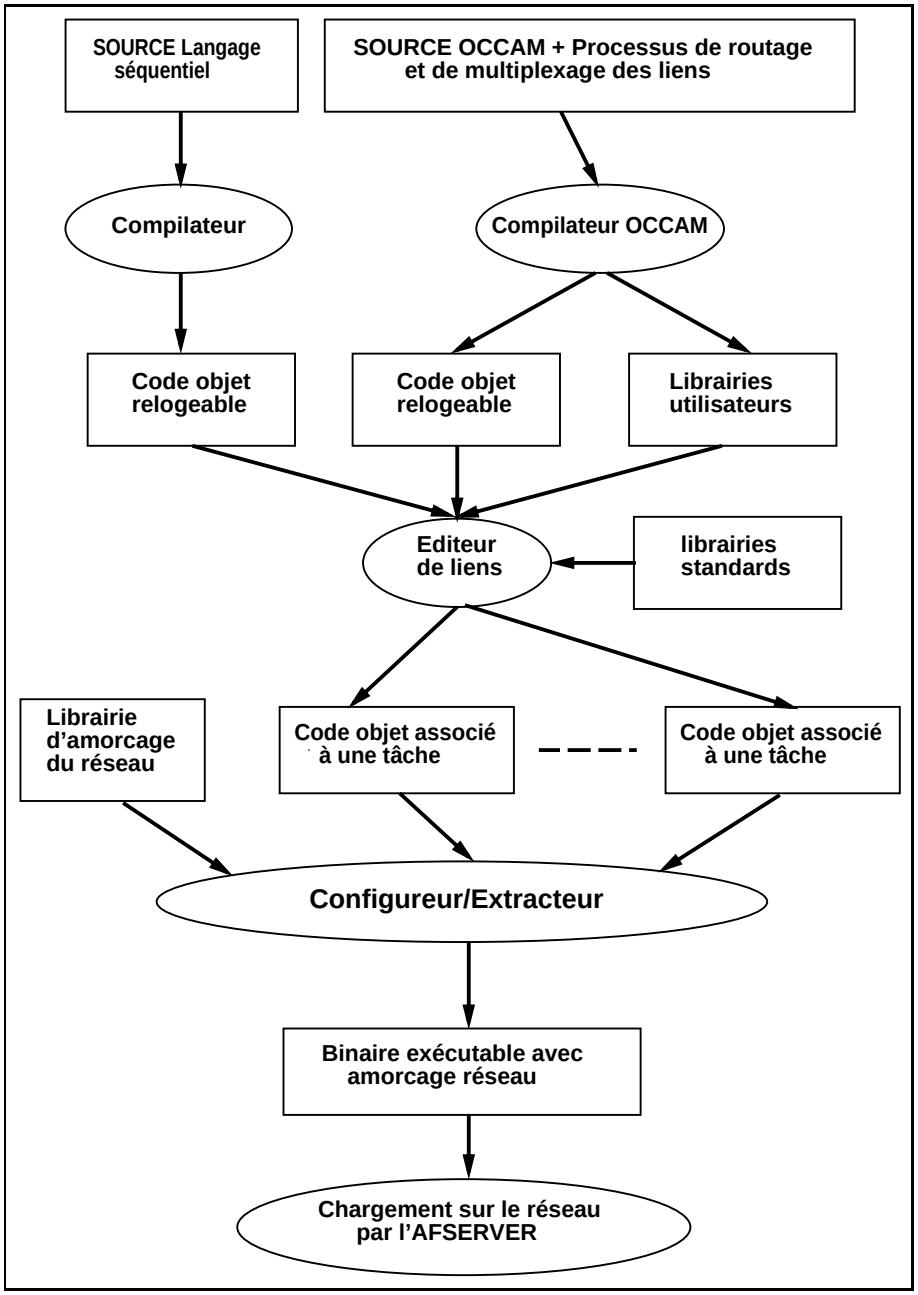


Figure 2.2: Structure du Stand-Alone Toolset d'Inmos

Voici les principales étapes à marquer lors du développement :

1. Le programmeur conçoit son programme en un ensemble de tâches à faire tourner en parallèle. Il peut avoir à regrouper certaines tâches en une plus grosse de manière à avoir une correspondance une tâche - un processeur. Une tâche peut être écrite dans un mélange d'OCCAM et d'autres langages comme C, Pascal, ...

2. Le programmeur réalise lui-même la partie routage des messages et multiplexage/démultiplexage des liens physiques qu'il intègre à chaque tâche.

3. Le compilateur et l'éditeur de liens sont appelés pour obtenir une image binaire par tâche.

4. Ensuite, le programmeur définit le réseau de processeurs. A chaque processeur est assignée une tâche, et à chaque lien physique un lien logique inter-tâches.

5. Le programmeur active le configureur qui :

   (a) analyse la description du réseau de manière à définir un chemin de chargement de tous les processeurs,

   (b) récupère les binaires associés à chaque tâche (résultats de la phase d'édition de liens),

   (c) fabrique un code binaire chargeable sur le réseau.

6. Le binaire obtenu est chargé par l'utilitaire `afserver` sur le réseau de processeurs.

Avantages et inconvénients

Ce système présente les mêmes avantages et inconvénients que le TDS. La seule amélioration est l'ouverture sur le système d'exploitation de la machine hôte qui offre une meilleure ergonomie d'utilisation et qui permet d'aborder des projets de taille plus importante que celle des applications dédiées.

2.4 Le système PDS

Tous les systèmes de développement existants actuellement nécessitent une intervention plus ou moins importante du programmeur. En effet, celui doit souvent faire des choix pour :

- Découper son application en un ensemble de processus.
- Effectuer le placement de ces processus sur le multiprocesseur.
- Ecrire un algorithme de routage nécessaire pour réaliser les communications inter-processus.

Ces trois actions ont des conséquences importantes quant au comportement du programme lors de son exécution. Malheureusement, si l'utilisateur ne connait pas précisement les caractéristiques physiques de sa machine cible, il peut ne pas choisir les meilleures possibilités. Si son application doit tourner sur plusieurs multiprocesseurs dont la topologie est différente, il doit refaire son travail autant de fois qu'il y a de machines. Cette démarche est importante et nous estimons, en particulier, que les phases de placement et de routage ne doivent pas être, en général, du ressort de l'utilisateur. D'autre part, si le multiprocesseur cible est à topologie reconfigurable, le problème du routage et du multiplexage des voies de communications devient difficilement contrôlable par un programmeur non spécialiste.

Cette automatisation est le sujet principal de nombreuses recherches. Citons, par exemple, le travail de Hiromi OHARA et Hajime IIZUKA [OI88] qui permet d'alléger la programmation en OCCAM d'une machine multiprocesseurs à base de transputers (Topologie statique), en générant automatiquement le placement des processus sur les processeurs.

2.4.1 Préprocesseur de configuration [OI88]

L'utilisateur de ce préprocesseur (figure 2.3) peut écrire un programme source OCCAM sans se soucier de la topologie de la machine cible, l'information concernant la configuration matérielle de celle-ci étant utilisée en seconde entrée du préprocesseur. A partir de ces deux entrées, le préprocesseur produit un code source OCCAM (composé de plusieurs fichiers) réalisant l'allocation des processus sur les processeurs. Le code "*système*" correspondant

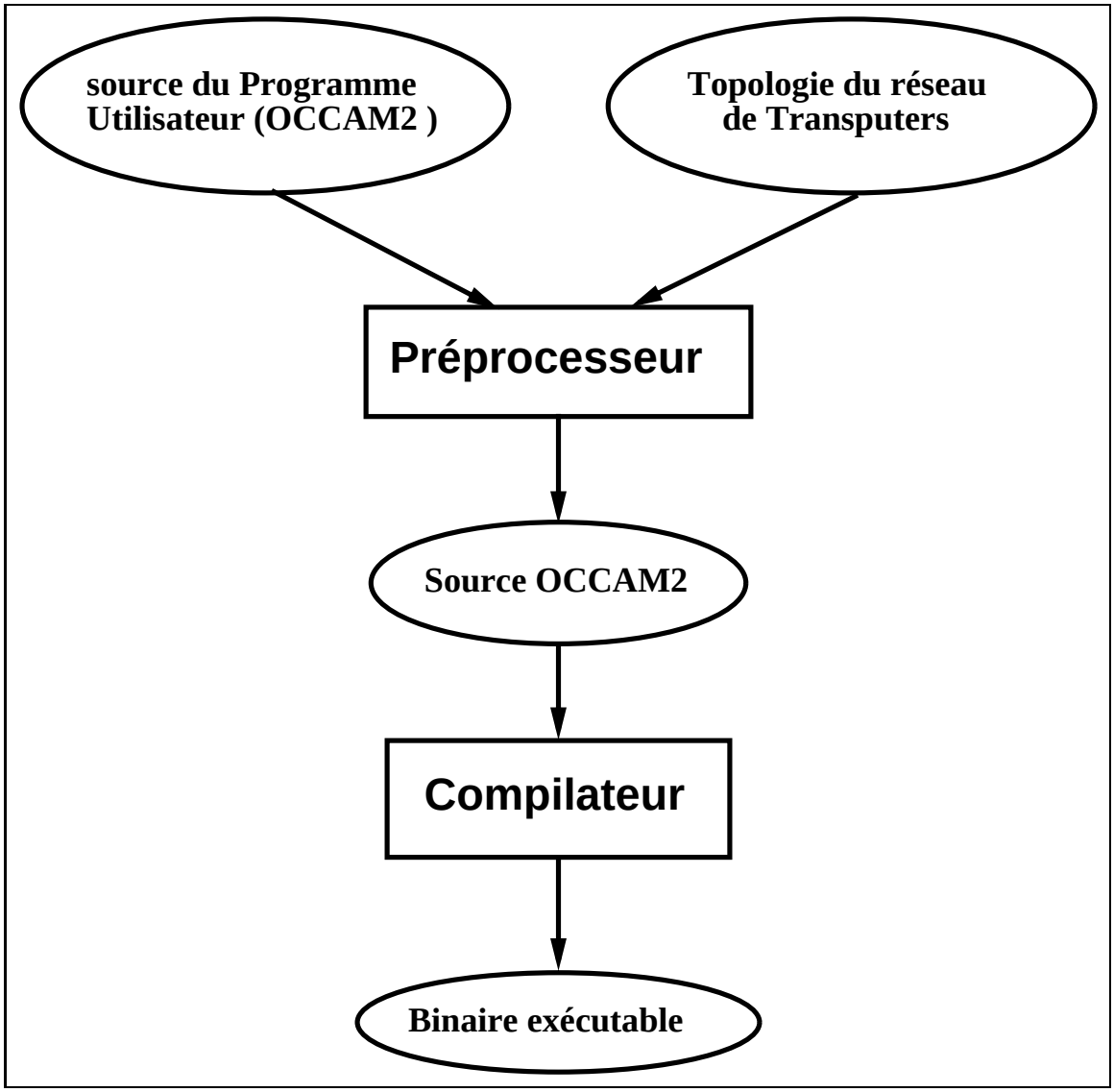


Figure 2.3: Structure du système de OHARA et IIZUKA

au routage des messages entre les processeurs est lui aussi généré automatiquement.

La partie d'extraction du parallélisme est réduite ici à sa plus simple expression puisque seul le niveau le plus englobant est pris en compte. Ainsi, une application constituée d'abord d'une partie séquentielle, puis d'une partie parallèle sera allouée sur un seul processeur. Ceci est une restriction sévère puisque de nombreuses applications répondent au schéma classique d'exécution suivant :

1. partie initialisation (lecture de données,...) (code séquentiel)
2. partie calcul (code parallèle)
3. partie traitement des résultats (code séquentiel)

La partie calcul pouvant être elle-même constituée de parties séquentielles et de parties parallèles. C'est cette partie que l'on s'attache d'habitude à optimiser, mais les deux autres ne sont pas négligeables car elles correspondent à des migrations de données de l'utilisateur vers le programme et vice-versa.

Avec ce type de préprocesseur, le programmeur doit exprimer le parallélisme de son application au niveau le plus englobant de son programme. Cette approche permet déjà de s'éloigner de la machine physique puisque la partie placement des tâches et routage est effectuée automatiquement. Nous pensons qu'il faut affiner de manière automatique le grain du parallélisme qui est laissé ici au choix du programmeur.

2.4.2 L'approche utilisée dans PDS

Nous proposons un système de développement de programmes parallèles appelé "Parallel programming Development System" (PDS), qui permet de réaliser de manière automatique les trois actions précitées (c.f. figure 2.4).

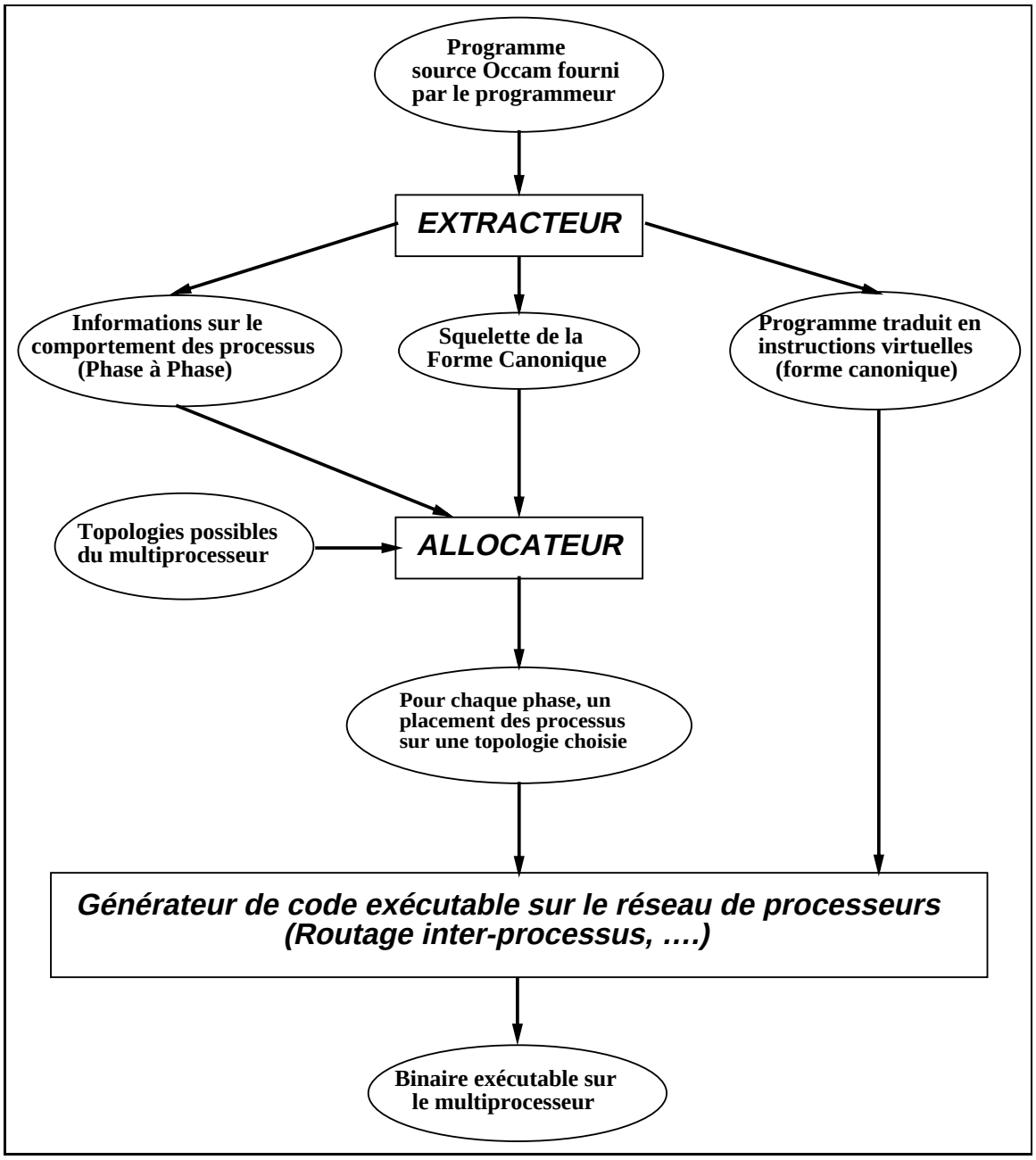


Figure 2.4: Structure du PDS

La clef de voûte de ce système est la définition d'une machine virtuelle permettant de s'abstraire de la machine multiprocesseur disponible. Cette machine virtuelle détaillée dans le chapitre 3, supporte une exécution de programmes parallèles présentés sous une forme canonique que nous définissons. Baptisée "Parallel Virtual Machine" (PVM), elle permet de réaliser une interface propre entre les trois modules de base du PDS :

1. La fonction principale de l'**extracteur** est de transformer un programme source produit par le programmeur en un programme exprimé en instruction virtuelle de la PVM. De plus, ce module collecte les informations nécessaires au module d'allocation.
2. Le module d'**allocation** reçoit en entrée :
   (a) Les informations collectées par l'extracteur.
   (b) Un graphe de l'exécution du programme transformé sur la machine virtuelle.
   (c) Un choix de topologies possibles, de manière à ne pas restreindre notre étude à une architecture à topologie fixe.

   Pour chaque transition de la forme canonique nous obtenons :

   (a) La topologie la plus adaptée à l'exécution de la transition.
   (b) Un placement des processus de la transition sur cette topologie.

   Le résultat obtenu après allocation est un placement des processus de la machine virtuelle sur une succession de topologies de processeurs.
3. Le troisième module, appelé **générateur de code**, récupère le code virtuel généré par l'extracteur, les placements des processus produits par l'allocateur et construit les processus réalisant la fonction de routage des messages entre les processeurs. Sa seconde tâche est de compiler le code de la machine virtuelle de manière à obtenir un code binaire exécutable sur le multiprocesseur concerné. Ce module de la chaîne de développement PDS est celui qu'il faut réécrire pour tout changement de multiprocesseur.

Chapitre 3

Machine Virtuelle

Dans ce chapitre, nous présentons une machine virtuelle appelée "**Parallel Virtual Machine**" (PVM) permettant de s'éloigner de toute contrainte liée à une machine physique donnée comme : le nombre de liens externes de communication, la politique de gestion des processus sur les processeurs, etc...

La première partie de ce chapitre présente une terminologie (reprise ultérieurement dans la suite du chapitre), la seconde est consacrée au problème de la cohérence des accès concurrents sur les données globales du programme. La troisième expose la machine virtuelle dans son ensemble. Dans la dernière partie, nous détaillons le jeu d'instructions de la PVM.

3.1 Terminologie

Tout programme est décomposable en deux éléments :

- le premier caractérise l'état du programme : les données.
- le second permet de faire évoluer cet état : le code.

Un programme peut donc être considéré comme un automate qui, pas à pas, en exécutant du code, modifie les données initiales pour obtenir finalement des données résultats. Nous désignons également les données du programme par le terme **d'environnement global**.

Cet environnement global contient donc :

- les données initiales du problème que le programme doit résoudre.

- les données finales "fruits du travail" du programme.
- des données intermédiaires correspondant à des états transitoires de l'automate.

En observant l'évolution de l'automate, nous nous apercevons que des groupes de données peuvent évoluer de manière relativement indépendante les uns des autres. Ceci laisse présager une évolution parallèle possible de plusieurs données au même instant.

C'est le cas d'une transition modifiant simultanément plusieurs données de l'environnement global. En effet, cette transition peut être découpée en un ensemble de processus parallèles coopérant entre eux de manière à passer d'un état i à un état $i+1$. Une telle transition est dite **transition parallèle**.

Lors de la compilation d'un langage à faire exécuter sur une architecture parallèle, le principal problème est la détection automatique des transitions parallèles. Malheureusement, sur l'ensemble des données d'un programme, nous ne pouvons pas toujours savoir à quelle donnée de l'environnement global va accéder un processus. C'est le cas (Figure 3.1) de l'accès à un élément d'un tableau par l'intermédiaire d'une valeur d'index communiquée par un autre processus.

```
processus P:        processus Q:
...                 ...
VAL = ...           ...
send_to(Q,VAL)      rec_from(P,i)
...                 a = ... tab_data[i] ....
```

Figure 3.1: Accès à un tableau par un index non connu

Ceci implique que le modèle d'exécution doit tenir compte de deux types d'accès possibles aux données globales :

1. Le premier que nous appelons *Accès Statique (i.e. Résolu à la Compilation)* puisqu'au moment de la compilation nous connaissons l'identité des données accédées par le programme.
2. Le second, nommé *Accès Dynamique (i.e. Résolu à l'Exécution)* désignera l'accès à une donnée dont l'identité est inconnue au moment de la compilation.

Avant de détailler les problèmes d'accès à l'environnement global, une présentation du modèle d'exécution est nécessaire. Ce modèle est étroitement lié au concept de mémoire virtuelle qui représente l'environnement global.

Une application (Figure 3.2) est constituée d'une séquence de tâches de calcul qui modifient pas à pas l'environnement global. Celui-ci n'est observable qu'entre l'exécution des tâches.

```
Application::   SEQ
                   Tache 1
                     .
                     .
                     .
                   Tache N
```

Figure 3.2: Définition d'une application

Chaque tâche de calcul (Figure 3.3) représente l'exécution du code correspondant à une transition dans l'automate. Toute tâche est composée d'un ensemble de processus de calcul qui s'exécutent en parallèle. Ceux-ci modifient de manière concertée les données globales du programme de façon à transiter de l'état i vers l'état $i+1$.

```
Tache i::  PAR
             Processus calcul 1
                     .
                     .
                     .
             Processus calcul N
```

Figure 3.3: Définition d'une tâche

Le processus de calcul est l'unité de base allouée sur un processeur. Il n'est pas nécessairement séquentiel, et peut activer ce que nous appelons des processus légers. Ceux-ci s'exécutent dans le contexte du processus de calcul qui les a générés, en utilisant les possibilités du noyau de processus intégré sur chaque processeur.

3.2 Cohérence des accès aux données globales

Pendant l'exécution d'une application, il faut s'assurer du maintien de la cohérence des données présentes dans l'environnement global. Si deux processus d'une même tâche veulent modifier une même variable globale, se pose alors la question de la valeur à retenir :

- la plus récente,
- la plus ancienne en annulant l'effet de la seconde mise à jour par un système de copie locale,
- etc ...

Pour résoudre cette question, nous imposons les deux règles suivantes :

R1 : Si une donnée globale est accédée seulement en lecture par un processus de calcul d'une tâche T, alors tous les autres processus de calcul de la tâche T ne peuvent accéder cette donnée qu'en lecture.

R2 : Si une donnée est accédée en écriture par un processus de calcul d'une tâche T, alors les autres processus de calcul de la même tâche T ne doivent pas y accéder (même en lecture).

Au moment de la compilation, une bonne partie des vérifications peut être effectuée. Mais dans l'accès dynamique et dans certains cas d'accès statique il faudra effectuer un contrôle dynamique.

Considérons par exemple, le cas où deux processus de calcul Pi et Pj modifient la valeur d'une variable V mais où chacun d'eux le fait sous une condition donnée, respectivement Ci et Cj. Pour être en accord avec les deux règles énoncées ci-dessus, il est nécessaire que Ci et Cj ne soient jamais vraies en même temps. Dans le cas où Ci ou Cj ne sont pas évaluables au moment de la compilation, la vérification est à faire au moment de l'exécution.

Ce cas particulier illustre le fait que même si l'on peut connaître l'identité des variables globales accédées par un groupe de processus, on ne peut pas savoir au moment de la compilation si une variable est modifiée par un et un seul processus de calcul. La seule solution alors est d'effectuer un contrôle dynamique.

Que le contrôle soit dynamique ou statique, les vérifications auxquelles il faut procéder se résument à la table 3.1 ; le nouvel état de la donnée est

obtenu en fonction de son état courant et du mode d'accès effectué par un processus de calcul P_k (k désignant l'identité du processus). Ces états sont au nombre de quatre :

Non_Accédé : pas d'accès sur la donnée globale.

Lecture(Pj) : accès en lecture seule par le processus Pj.

Lecture_tous : accès en lecture seule pour tous les processus d'une tâche.

Ecriture(Pj) : accès en lecture/écriture pour le processus Pj.

type de l'accès	état courant de la donnée			
	Non_Accédé	Lecture(P_j)	Lecture_tous	Ecriture(P_j)
Lecture(P_k)	Lecture(P_k)	$k = j$ Lecture(P_j) $k \neq j$ Lecture_tous	Lecture_tous	$k = j$ Ecriture(P_j) $k \neq j$ Erreur
Ecriture(P_k)	Ecriture(P_k)	$k = j$ Ecriture(P_k) $k \neq j$ Erreur	Erreur	$k = j$ Ecriture(P_j) $k \neq j$ Erreur

Table 3.1: tableau des vérifications

3.3 Machine Virtuelle

Après avoir précisé les problèmes rencontrés lors de la vérification de la cohérence des accès aux données globales, nous pouvons présenter la machine virtuelle.

Elle est décomposable en cinq parties :

1. un groupe de processeurs de travail (puissance de traitement parallèle).
2. Un réseau de communications interprocesseur, avec son contrôleur.
3. Une mémoire virtuelle.
4. Un réseau de communications entre la mémoire virtuelle et les processeurs de travail.
5. Un contrôleur central.

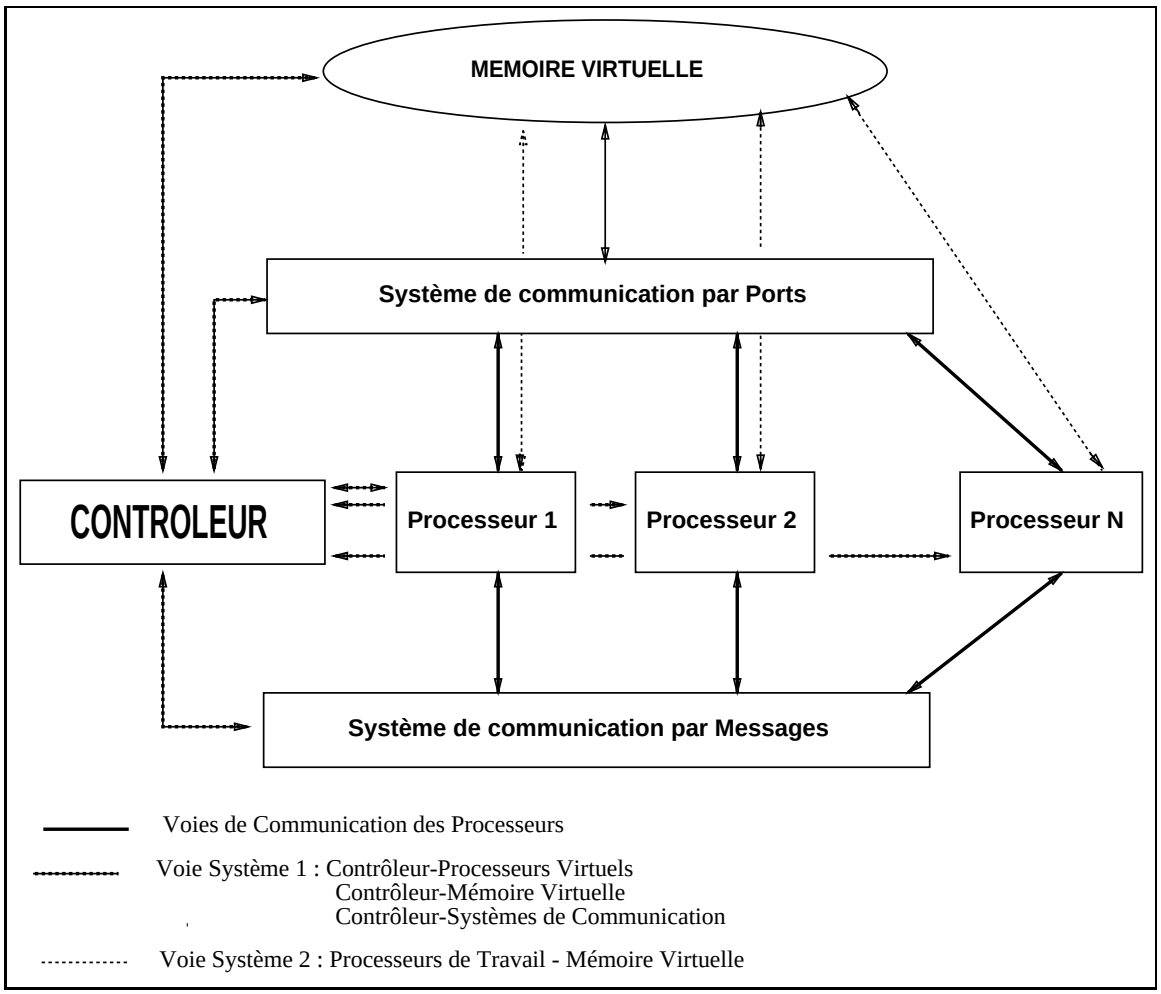


Figure 3.4: Structure de la Machine Virtuelle

3.3.1 Processeurs Virtuels

Nous supposons que nous avons un nombre non limité de processeurs. Chaque processus de calcul s'exécute sur un processeur de traitement virtuel. Le réseau de communications inter-processeurs est nécessaire pour permettre des échanges synchrones ou asynchrones par messages entre les processus de calcul.

3.3.2 Réseau de communications interprocesseurs virtuel

Nous considérons que tous les processus savent dès le moment de la compilation avec qui ils vont communiquer. Nous éliminons le cas d'établissement de communications "dynamiques" (cas rarement rencontré dans les programmes). En effet, ce genre d'approche est surtout adopté pour des algorithmes de routages, par exemple.

Nous supposons que nous avons un système de communication permettant à deux processeurs virtuels de communiquer par un lien virtuel bidirectionnel. Toute communication entre eux passe nécessairement par la création de ce lien qui n'est connu que des deux processeurs interlocuteurs.

La demande de création d'un lien virtuel doit être exprimée par les deux processus en respectant le protocole suivant :

Considérons que $P_j$et $P_k$veulent communiquer entre eux. $P_j$est le premier à exprimer la demande de création de lien, et se bloque tant que le processus P_kn'a pas exprimé la même demande. Lorsque c'est le cas, les deux processus reprennent leurs activités avec un lien virtuel établi entre eux.

Pour ce qui est de l'utilisation de ce lien virtuel, nous laissons deux choix de protocoles possibles au programmeur, à exprimer lors de la création du lien :

- soit un protocole synchrone (Type Rendez-vous).
- soit un protocole asynchrone (Type Dma).

Pour détruire un lien virtuel, il faut que les deux processeurs virtuels en fassent la demande. La destruction ne peut être effectuée qu'en l'absence de toute communication sur le lien concerné. Ceci est tout à fait contrôlable dès la compilation en comptant le nombre d'ordres d'émission et de réception effectués de part et d'autre du lien.

3.3.3 Mémoire Virtuelle

Dans le modèle d'exécution présenté, la mémoire virtuelle contient toutes les données globales de l'application. Tout accès à l'une d'elles effectué par un processus de calcul se fait obligatoirement par la mémoire virtuelle. Entre l'application et la mémoire virtuelle, les communications sont effectuées par des objets de communication bi-directionnels que nous appelons des **ports**. La durée de vie d'un port est celle d'un processus de calcul.

Utilisation des ports

Nous distinguons deux groupes de primitives d'accès aux ports suivant leurs fonctionnalités :

- primitives de gestion.
- primitives d'accès.

Primitives de gestion Nous qualifions ainsi, les primitives permettant à un processus de calcul de demander à la mémoire virtuelle une création ou une destruction de port.

Avant d'utiliser un port, tout processus de calcul doit demander la création de celui-ci. Cette demande désigne la donnée à accéder, la manière de contrôler l'accès (statique ou dynamique) et le type de l'accès effectué sur cette donnée (écriture, lecture, ...). Ces informations sont nécessaires à la mémoire virtuelle pour qu'elle puisse assurer les vérifications nécessaires.

La destruction d'un port est l'opération contraire. Lorsqu'un processus de calcul n'utilise plus une donnée globale, il doit restituer le port qu'il utilisait à la mémoire virtuelle. Cette restitution peut être effectuée par le processus, ou de manière systématique en fin d'exécution du processus.

Primitives d'accès Contrairement aux précédentes, les primitives d'accès utilisent un port pré-existant afin de réaliser le protocole entre la mémoire virtuelle et un processus de calcul. Nous avons adopté ici une approche clients-serveur, où la mémoire virtuelle joue le rôle de serveur auprès des processus de calcul.

Le processus de calcul dépose sa demande sur le port. Il lit la réponse du serveur en se mettant à l'écoute sur ce port. Cette lecture est une opération

bloquante. La mémoire virtuelle, en émettant sur le port les informations correspondant à la demande déposée, va débloquer le processus de calcul en attente. L'avantage de ce protocole par rapport à celui du Rendez-vous, est la possibilité pour le processus de calcul d'utiliser le temps d'attente pendant lequel la mémoire virtuelle traite sa requête pour effectuer un autre traitement.

Le format de la requête utilisée ici est basé sur les types d'accès possibles sur une donnée globale, à savoir lecture ou écriture. En retour, le processus demandeur reçoit un compte-rendu l'informant du déroulement de sa requête selon l'une des deux formes suivantes :

1. La lecture ou l'écriture demandée s'est effectuée sans problème.
2. Une erreur est détectée par la mémoire virtuelle ; le non-respect des règles de cohérence d'accès en est un exemple.

Fonctionnement

Côté mémoire virtuelle La mémoire virtuelle doit assurer la cohérence des demandes d'accès aux données globales. Deux types de contrôle possibles supposent deux classes de port à distinguer :

1. Les ports à contrôle statique.
2. Les ports à contrôle dynamique.

Un processus de calcul utilise de manière différente ces deux classes de port.

Si tous les processus de calcul d'une tâche n'utilisent que des ports à contrôle statique, les requêtes qu'ils déposent ont été validées au moment de la compilation ; donc aucun cas d'erreur n'est à traiter au niveau de la mémoire virtuelle. Ce cas de figure regroupe une grande majorité des applications rencontrées. Dans les applications scientifiques, plus particulièrement, nombreux sont les algorithmes travaillant sur des données stockées dans des tableaux et accédées de manière itérative sur les index.

En revanche, pour les applications utilisant simultanément des ports à contrôle statique et dynamique, le traitement est plus complexe. Il est nécessaire d'introduire la notion d'état de contrôle d'une donnée, qui permet à la mémoire virtuelle d'avoir un résumé de tous les types d'accès des ports faisant référence à cette donnée.

contrôle du port demandé	état de contrôle courant de la donnée			
	INDIFFERENT	DYNAMIQUE	STATIQUE	MIXTE
STATIQUE	STATIQUE	MIXTE	STATIQUE	MIXTE
DYNAMIQUE	DYNAMIQUE	DYNAMIQUE	MIXTE	MIXTE

Table 3.2: évolution de l'état de contrôle

Au début de chaque tâche, cet état est initialisé à la valeur INDIFFERENT qui signifie que pour le moment aucun port n'a été demandé pour accéder à la donnée. Puis, suivant le type d'accès demandé à chaque création de port, cet état évolue comme précisé dans la table 3.2 qui nous fournit le nouvel état de contrôle en fonction de l'état courant et du type de contrôle du port spécifié lors de la demande de création.

Grâce à l'état de contrôle d'une donnée, la mémoire virtuelle sait distinguer les deux cas de traitement suivants :

1. Un traitement minimal pour le cas où cet état de contrôle est STATIQUE, puisque tous les contrôles de cohérence des accès ont été effectués lors de la compilation. La seule préoccupation est de se souvenir du type d'accès exprimé lors de la demande de création de port qui correspond à l'état de la donnée à savoir : lecture(P_j), écriture(P_j) ou lecture_tous.

2. Un traitement plus important puisqu'il met en action ce que nous appelons une protection dynamique des données. Celle-ci entre en jeu dès lors que l'état de contrôle courant n'est plus STATIQUE. Il nous faut assurer ici les mêmes règles de cohérence d'accès que celle réalisées statiquement lors de la compilation du programme. La table 3.3 que nous présentons ici a été enrichie par rapport à la table 3.1 en fonction du type d'accès exprimable lors de la création des ports à savoir lecture(P_j), écriture(P_j) et lecture_tous.

Pour que ces vérifications soient effectuées lors de l'exécution, la mémoire virtuelle maintient donc outre la valeur de la donnée, l'état de contrôle des ports accédant à cette donnée et l'état des accès effectués sur cette donnée.

type de l'accès demandé sur le port	état courant de la donnée ($j \neq k$)			
	NON_ACCEDE	Lecture(P_j)	Ecriture(P_j)	Lecture_tous
Lecture(P_j)	Lecture(P_j)	Lecture(P_j)	Ecriture(P_j)	Lecture_tous
Lecture(P_k)	Lecture(P_k)	Lecture_tous	ERREUR	Lecture_tous
Ecriture(P_j)	Ecriture(P_j)	Ecriture(P_j)	Ecriture(P_j)	ERREUR
Ecriture(P_k)	Ecriture(P_k)	ERREUR	ERREUR	ERREUR
Lecture_tous	Lecture_tous	Lecture_tous	ERREUR	Lecture_tous

Table 3.3: Transition de l'état de la donnée

Côté processus de calcul Le processus de calcul peut exprimer ses demandes de création de ports selon les deux démarches suivantes :

1. Nous regroupons toutes les demandes de création de port en prologue du processus. Ainsi, celui-ci peut débuter son travail en ayant l'accès à toutes les données à atteindre éventuellement. Puis, en épilogue, nous restituons tous les ports demandés pendant le prologue. L'avantage de cette méthode est que le processus de calcul n'a pas à attendre pendant son exécution une création de port. Mais l'inconvénient majeur est que toutes les demandes de créations de ports vont être effectuées en même temps vers la mémoire virtuelle puisque tous les processus de calcul débutent simultanément. Le temps de service d'une demande de création de port étant non négligeable, le processus perd du temps par rapport à la démarche suivante.

2. Nous choisissons d'effectuer les demandes au coup par coup. Ceci a l'avantage de permettre à un processus de calcul de pouvoir commencer son travail plus tôt puisqu'il n'a pas à attendre la création de tous les ports. D'autre part, ceci équilibre mieux la charge de travail de la mémoire virtuelle qui fournit seulement ce qui est nécessaire à l'exécution du processus. De même, la restitution de ports peut être à l'initiative

du processus de calcul qui prend cette décision dès que le port devient inutile.

Chaque processus de calcul maintient un cache local des données globales qu'il a lues, de manière à ne pas surcharger le réseau de communications entre la mémoire virtuelle et les processus de calcul. Nous pouvons choisir plusieurs politiques de gestion de ce cache, citons-en une qui nous semble être intéressante :

- Tous les accès en lecture se font dans le cache sauf le premier accès qui initialise la valeur dans le cache.
- Tous les accès en écriture se font dans le cache et nous répercutons instantanément les modifications vers la mémoire virtuelle.

Cette démarche délègue la mise à jour du cache à un processus léger de l'application. Ainsi, le travail en cours de traitement n'est pas interrompu pendant la durée de communication entre la mémoire virtuelle et le processeur de traitement.

3.3.4 Contrôleur central

La machine virtuelle doit exécuter des enchaînements de tâches de calcul. Pour que le séquencement soit réalisé, un protocole de contrôle doit être mis en place.

Pour activer une tâche, le contrôleur émet vers chaque processeur de travail et vers la mémoire virtuelle l'ordre DEBUT_TACHE. Ensuite, le contrôleur écoute en priorité la mémoire virtuelle qui l'informe ainsi le plus rapidement possible d'un cas d'erreur détecté au moment d'un accès. Simultanément, il attend de chaque processeur de travail une réponse FIN_TACHE qui lui signifiera que ce même processeur est redevenu inactif. Lorsque tous les processeurs ont achevé leur travail et qu'aucune erreur n'a été détectée, le contrôleur peut activer la tâche suivante. Dans le cas d'une détection d'erreur, le contrôleur prend la décision de stopper l'application et de rapporter un message d'erreur à l'utilisateur.

L'algorithme exécuté pour le contrôle de l'exécution d'une tâche est présenté en figure 3.5.

```
/* TRAITEMENT D'UNE TACHE I                                        */
/* Nb_Proc[I] designe le nombre de processeurs ACTIF               */
/* de la tache I                                                   */
/* Processeur[j] designe le jeme processeur de traitement          */
Par j = 1 a Nb_Proc[I]
  emet(Processeur[j], DEBUT_TACHE)
  TACHE_NON_FINIE = VRAI
  FAUTE_MEMOIRE = FAUX
  Nb_processeur_actif = Nb_Proc[I]
  Tant Que (TACHE_NON_FINIE)
    Alternative
      garde : Recoit(Mem_virtuelle,MEM_FAUTE)
        FAUTE_MEMOIRE = VRAI
        TACHE_NON_FINIE = FAUX
      garde : Alternative j =1 a Nb_Proc[I]
          garde : recoit(processeur[j], FIN_TACHE)
            Nb_processeur_actif = Nb_processeur_actif+1
  Si (Nb_processeur_actif = 0)
    alors TACHE_NON_FINIE = FAUX
  Si (FAUTE_MEMOIRE)
    alors STOPPER_APPLICATION
/* FIN DU TRAITEMENT DE LA TACHE I */
```

Figure 3.5: Algorithme exécuté par le contrôleur

3.4 Instructions de la PVM

Le jeu d'instructions de la machine virtuelle décrite précédemment est constitué d'un jeu d'instructions classique d'un calculateur séquentiel enrichi de celles propres à gérer le parallélisme. Elles se classent donc en quatre groupes :

1. accès à la mémoire virtuelle.
2. gestion des processus légers d'un processus de calcul.
3. gestion du séquencement des tâches de l'application.
4. gestion des communications interprocessus.

3.4.1 Accès à la mémoire virtuelle

Deux types de primitives sont nécessaires pour "piloter" les accès à la mémoire virtuelle d'un processus de calcul :

- les primitives de gestion : allouer_port(), détruire_port().
- les primitives d'accès : dépôt_requête(), retirer_compte_rendu().

allouer_port(Pid, Ident, Type_accès, Mode_accès,Adr_port, Status)

Pid (entrée) désigne l'identité du processus de calcul demandant un port.

Ident (entrée) identifie la donnée placée en mémoire virtuelle .

Type_accès (entrée) représente le type de l'accès demandé.

Mode_accès (entrée) désigne le type des opérations (Table 3.3) autorisées sur ce port.

Adr_port (sortie) est l'adresse d'un port utilisable.

Status (sortie) informe le processus demandeur du déroulement de sa demande. Ceci est utile, par exemple, dans le cas où le même processus demande plusieurs fois un port vers une même donnée.

détruire_port(détruire_port(Adr_port,Status)

Adr_port (entrée) est l'adresse d'un port que le processus de calcul veut restituer.

Status (sortie) informe le processus demandeur du déroulement de sa demande. Ceci est utile, par exemple, en cas d'erreur lorsqu'un processus veut rendre un port qu'il aurait déjà restitué.

dépôt_requête(Adr_port,Requête,Status)

Adr_port désigne le port représentant la donnée à atteindre en mémoire virtuelle.

Requête deux formes sont possibles suivant le type d'accès demandé :

demande_lecture Un accès en lecture est demandé sur le port.

(demande_écriture,VAL) Un accès en écriture est demandé pour effectuer la mise à jour de la donnée globale avec la nouvelle valeur VAL.

retirer_compte_rendu(Adr_port, Compte_rendu)

Adr_port désigne le port représentant la donnée à atteindre en mémoire virtuelle.

Compte_Rendu trois formes sont possibles :

1. (Lecture_effectuée, Valeur_lue).
2. (Ecriture_effectuée).
3. (Erreur_detectée, Code_erreur).

3.4.2 gestion des processus légers

Les processus légers d'un même processus de calcul s'exécutent sur le même processeur virtuel et peuvent éventuellement communiquer entre eux. Nous avons donc les intructions suivantes :

1. execute_par(P1,....PN)
2. émet_message_local et reçoit_message_local.

execute_par(P1, ... , Pn)

Pi (entrée) désigne l'identité des processus légers à exécuter.

remarque : Tout processus léger peut créer d'autres processus légers.

émet_message_local(Pid_loc, Message)

Pid_loc (entrée) désigne l'identité du processus local destination.

Message (entrée) contient l'information émise vers le destinataire.

reçoit_message_local(Pid_loc, Message)

Pid_loc (entrée) désigne l'identité du processus local émetteur.

Message (sortie) contient l'information reçue par le destinataire.

3.4.3 gestion des communications inter-processeurs

Les processeurs virtuels expriment des demandes de création et de destruction de liens virtuels à l'aide des instructions créer_lien() et détruire_lien(). Chaque lien est typé suivant le protocole choisi par les processeurs. S'il s'agit du protocole synchrone, les primitives envoyer_synchrone() et recevoir_synchrone() sont à utiliser. Dans le cas du protocole asynchrone, il faut utiliser envoyer_asynchrone() et recevoir_asynchrone(). Il faut y ajouter une instruction permettant de savoir si le message est parti ou non vers le processeur destinataire : test_msg_émis().

créer_lien(Protocole,Proc_Id,Lien_Id,Status)

Protocole (entrée) désigne le type du protocole choisi ; Synchrone ou Asynchrone.

Proc_Id (entrée) est l'identité du processeur avec lequel on veut communiquer.

Lien_Id (sortie) représente le nom logique du lien virtuel obtenu.

Status (sortie) informe du bon ou du mauvais déroulement de la création.

détruire_lien(Proc_Id,Lien_Id,Status)

Proc_Id (entrée) est l'identité du processeur avec lequel le processeur courant communiquait.

Lien_Id (entrée) représente le nom logique du lien virtuel à détruire.

Status (sortie) informe du bon ou du mauvais déroulement de la destruction. Un cas d'erreur possible peut être la demande de destruction d'un lien inexistant.

envoyer_synchrone(Lien_Id,Taille,Message,Status)

envoyer_asynchrone(Lien_Id,Taille,Message,Status)

Lien_Id (entrée) représente le nom logique du lien virtuel à utiliser.

Taille (entrée) représente le volume du message.

Message (entrée) est l'information que l'on veut émettre sur le lien.

Status (sortie) informe du bon ou du mauvais déroulement de la destruction. Un cas d'erreur possible peut être la non-existence du lien virtuel.

Remarque : Dans le cas du protocole asynchrone, le tampon message n'est pas réutilisable par le processeur émetteur tant que celui-ci n'est pas sûr que la transmission du message a été effectuée. Il doit utiliser la fonction test_msg_émis() pour le savoir. La primitive asynchrone n'est pas bloquante.

recevoir_synchrone(Lien_Id,Taille,Message,Status)

recevoir_asynchrone(Lien_Id,Taille,Message,Status)

Lien_Id (entrée) représente le nom logique du lien virtuel à utiliser.

Taille (entrée) représente le volume du message à recevoir.

Message (sortie) est l'information que l'on récupère sur le lien.

Remarque: Ces deux primitives sont bloquantes, puisque le processeur récepteur veut consommer le message en y effectuant un traitement.

test_msg_émis(Lien_Id,Status)

Lien_Id (entrée) représente le nom logique du lien virtuel à utiliser.

Status (sortie) nous informe d'un message éventuellement en cours d'émission sur le lien. Si un transfert est en cours, le message ne peut pas être manipulé par le processeur virtuel. S'il n'y a aucun transfert en cours, le processeur peut réutiliser son message, mais ceci ne signifie pas que l'information soit déjà parvenue à son destinataire : le système de communication peut en avoir fait une copie.

3.4.4 séquencement des tâches de l'application

Deux instructions sont nécessaires aux processeurs virtuels pour suivre les étapes imposées par le contrôleur :

attendre_début_tache

Cette instruction bloque le processeur virtuel courant jusqu'à ce que tous les autres processeurs soient prêts à commencer l'étape N. C'est le contrôleur qui réactivera les processeurs bloqués.

informer_processus_terminé

Lorsqu'un processeur virtuel a terminé sa contribution à la tâche courante, il faut en informer le contrôleur. Ces informations lui permettront de savoir quels sont les processeurs virtuels inactifs.

Chapitre 4

Extraction automatique du parallélisme

Dans ce chapitre, nous présentons le système d'extraction du parallélisme. Un outil expérimental a été développé qui permet, à partir d'un programme OCCAM fourni par l'utilisateur, d'extraire le parallélisme interne à ce programme et de générer un programme OCCAM équivalent mais exprimant davantage de parallélisme. Les problèmes rencontrés lors d'une telle opération sont particulièrement importants (voir section 4.1). Ensuite le programme OCCAM est compilé en instructions PVM, et les informations nécessaires à la phase d'allocation sont extraites.

Le problème est de s'assurer que les modifications effectuées sur un programme n'altèrent pas sa fonction.

Le formalisme mathématique "Communicating Sequential Processes" [Hoa78] sur lequel se sont basés les concepteurs d'OCCAM [PH88] permet d'obtenir de nombreux résultats dont la validité est assurée dans tous les cas de figure. Ces résultats consistent en un ensemble de lois de transformation de programmes dont le lecteur trouvera une spécification en terme algébrique dans [RH86].

Ce chapitre est organisé de la manière suivante : après avoir présenté les problèmes spécifiques à l'extraction du parallélisme, nous abordons une description précise de notre extracteur.

4.1 Problèmes liés à l'extraction du parallélisme

Le but à atteindre est la transformation d'un programme quelconque en un programme parallèle pouvant s'exécuter sur la machine virtuelle présentée au chapitre 3. Le squelette d'un tel programme est présenté en figure 4.1.

```
SEQ
  SEQ1
    ...
  PAR1
    ...
  SEQ2
    ...
  ...
  SEQn
    ...
  PARn
    ...
  SEQn+1
    ...
```

Figure 4.1: squelette de programme s'exécutant sur la machine virtuelle

Nous devons obtenir en sortie de l'extracteur une séquence de N tâches virtuelles (PARi), encadrées par des exécutions séquentielles (SEQi et SEQi+1). Nous appelons ce squelette la "forme canonique" d'un programme.

L'extraction du parallélisme doit tenir compte des trois schémas d'extraction du parallélisme suivants :

1. Décomposer l'algorithme initial en un nombre important de processus de "petite" taille qui peuvent s'exécuter en parallèle. Ceci est aussi nommé décomposition (ou éclatement) du contrôle de l'exécution.

2. Distribuer les données entre les processeurs.

3. Il existe des cas où un groupe de processeurs est piloté par un processeur de contrôle ("Processor Farm")[Pad88].

Pour ce qui est de l'éclatement de l'algorithme en un ensemble de processus plus élémentaires s'exécutant en parallèle, il est nécessaire de prendre en compte les relations de dépendances entre les différents processus.

Relations de dépendances inter-processus

Deux types de dépendances sont possibles entre deux processus :

1. un héritage de l'environnement d'un processus exécutant un constructeur (appelé aussi processus père) et les processus éléments (ou processus fils) de ce constructeur.
2. un échange d'informations entre deux processus par communication sur un canal.

La première dépendance est exprimée dès l'écriture du programme par le programmeur à l'aide des constructeurs de OCCAM : SEQ, PAR, ALT, IF,... Ainsi, les processus éléments d'un constructeur ont la possibilité d'accéder aux variables du processus exécutant ce constructeur. Dans le cas où les processus élements sont actifs au même instant (cas du constructeur PAR), il convient d'appliquer les règles de cohérence telles qu'elles sont définies au chapitre 3. Nous pouvons qualifier ce partage d'environnement d'**implicite** dans le langage OCCAM.

Les communications effectuées sur les canaux permettent au programmeur d'exprimer une relation de dépendance entre des processus actifs simultanément. Ceci est dû au fait que les communications réalisées dans le langage OCCAM sont basées sur le principe du "*Rendez-Vous*". Ceci permet de suspendre l'exécution d'un processus Q en attendant qu'un autre processus P ait produit une valeur nécessaire à Q. Une fois la valeur reçue, Q reprend son exécution.

Considérons le programme OCCAM illustré en figure 4.2.

Le graphe de dépendance correspondant est présenté en figure 4.3.

```
PAR
  SEQ
    C ! VALEUR -- P1
    C ? VALEUR -- P2
    PAR
      P3
      P4
      P5
  SEQ
    C ? V    -- P6
    PAR
      P7
      P8
    C ! VAL -- P9
```

où les identificateurs `C,VALEUR,V,VAL` sont déclarés dans l'environnement du processus exécutant le premier PAR.

Figure 4.2:

Dans cet exemple, nous pouvons constater que :

- (P1,P6), et (P1,P7), par exemple, sont des groupes de processus dont tous les membres **peuvent** être actifs en même temps.
- (SEQ(P1,P2,PAR(P3,P4,P5)),SEQ(P6,PAR(P7,P8),P9)), (P3,P4,P5) et (P7,P8) sont des groupes dont les membres **sont** actifs en même temps ; ces processus sont appelés des processus **frères**. Nous ne pouvons pas considérer le début de l'exécution de l'un d'entre eux sans considérer le démarrage de tous les autres processus membres du groupe.
- Le processus P2 s'exécute avant le groupe PAR(P3,P4,P5). En supposant qu'il existe une horloge universelle, ceci peut être formulé de la façon suivante :

$$heure\_fin\_exécution(P2) < heure\_début\_exécution(PAR(P3, P4, P5))$$

- En considérant la relation de dépendance induite par les communications sur le canal `C`, nous en déduisons :

$$heure\_fin\_exécution(P1) = heure\_fin\_exécution(P6)$$

$$heure\_fin\_exécution(P2) = heure\_fin\_exécution(P9)$$

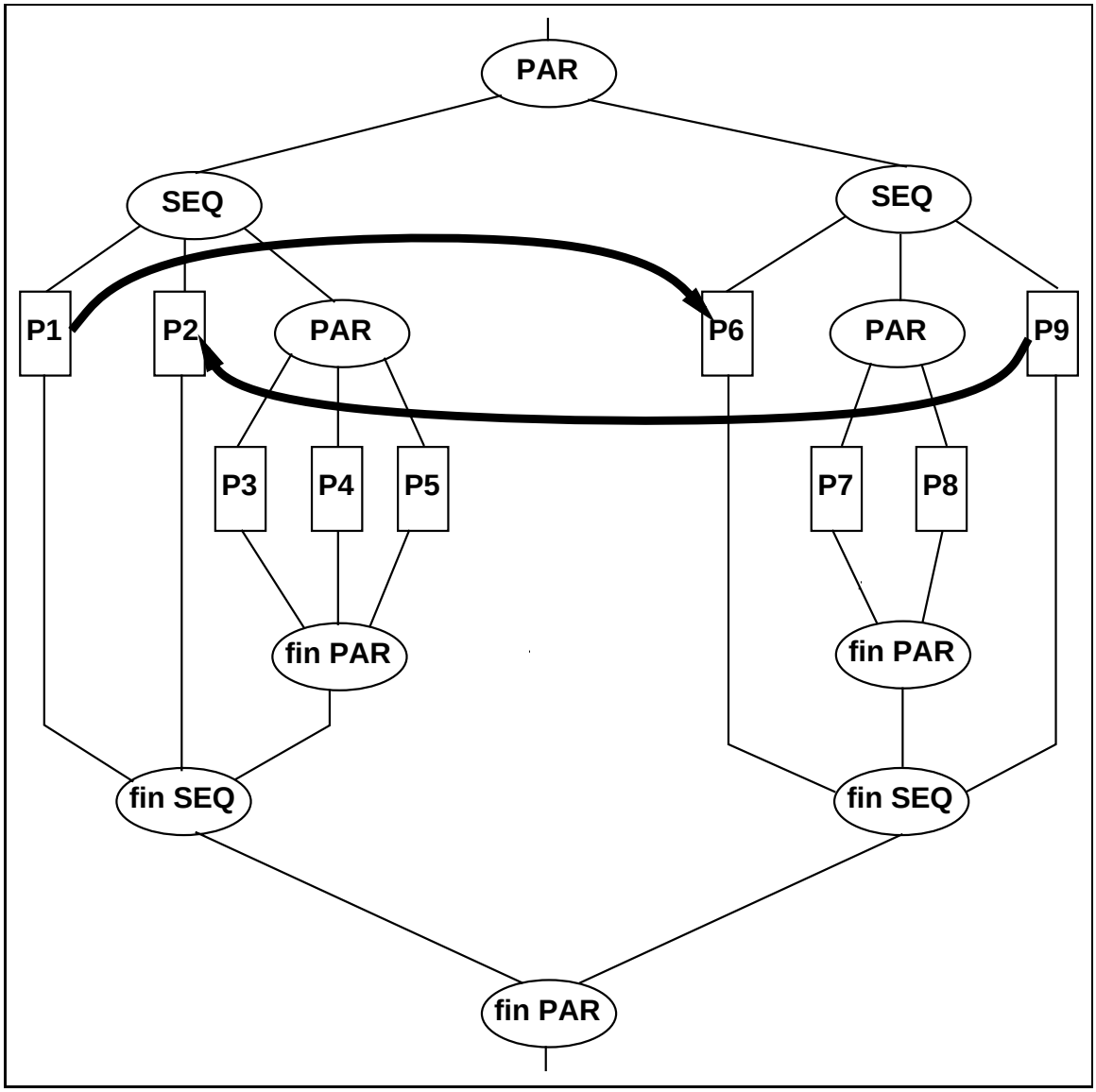


Figure 4.3: Graphe de dépendance

4.2 Système de transformation de PDS

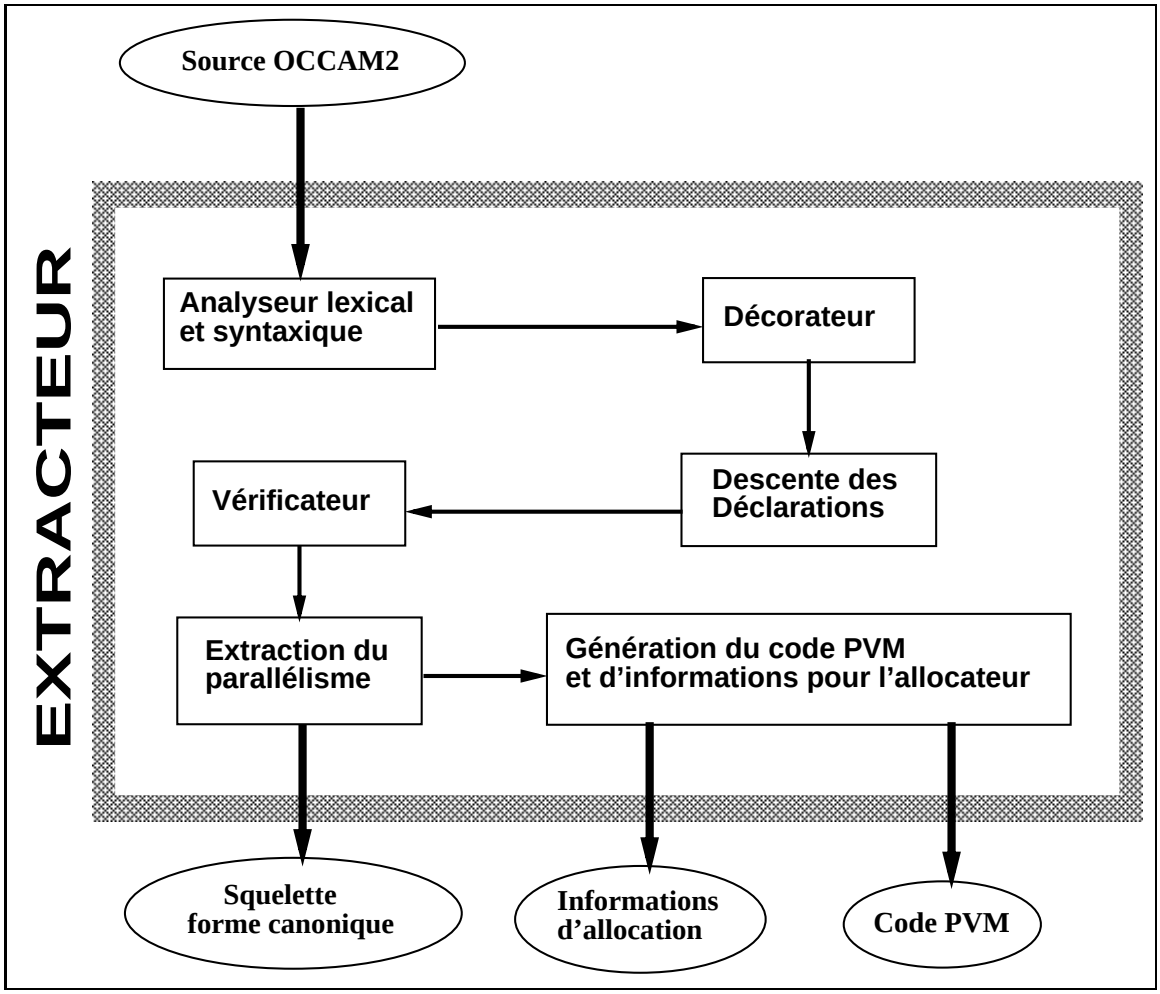


Figure 4.4: Structure de l'extracteur

Notre système est composé de six modules (c.f. figure 4.4). Chacun d'eux a une tâche particulière :

1. Analyse lexicale et syntaxique du programme source
2. Construction d'un arbre syntaxique décoré synthétisant un ordre partiel entre les nœuds processus
3. Descente des déclarations de variables. Nous entendons par là une désimbrication des déclarations de variables de manière à ce que les occurences d'utilisation de chaque variable soit le plus "proche" possible de la déclaration.

4. Détection de quelques erreurs possibles de la part du programmeur

5. Application des règles pour extraire le parallélisme

6. Génération de code pour la machine virtuelle et évaluation des informations nécessaires pour le système d'allocation (phase à phase).

4.2.1 Module d'analyse lexicale et syntaxique

Le premier module consiste en l'analyse lexicale et syntaxique du programme OCCAM2 soumis en entrée de l'extracteur.

Dans le cas présent, nous supposons qu'il existe une procédure OCCAM appelée `main` qui est le point d'entrée du programme. Cette procédure est particulière puisqu'elle régit entièrement le contrôle de l'exécution du programme. Tout programme est constitué d'une liste de déclarations dont l'une d'entre elles est la déclaration de la procédure **main**. Nous obtenons, une fois ces analyses terminées, un arbre syntaxique dont les nœuds principaux ont les caractéristiques suivantes :

nœud L_decl : est un nœud n^{aire}, n étant le nombre de déclarations présentes sous ce nœud.

nœud Type_decl : Type désigne le type des identificateurs déclarés. Ce nœud est n^{aire}, n étant le nombre d'identificateurs définis sous ce nœud.

nœud PROC_decl : Ce nœud réalise la définition d'une procédure (Mot clef PROC du langage OCCAM2). Il a trois nœuds fils :

1. un nœud nom de procédure.

2. un nœud liste de paramètres (L_param).

3. un nœud corps qui contient le corps de la procédure (Corps).

nœud L_param : Ce nœud est similaire à celui des listes de déclarations (L_decl), mais celles-ci sont locales au corps de la procédure.

nœud Bloc et nœud Corps : Ce nœud est binaire et comprend pour premier fils un nœud L_decl et pour second un nœud Code.

nœud While_code : Ce nœud représente le constructeur While de OCCAM2. Il possède comme fils un nœud cond_Bloc.

nœud Code : (ALT_code, SEQ_code, PAR_code, IF_code, ...) Ces nœuds regroupent l'ensemble des processus OCCAM2 sauf le constructeur While. Ils sont étiquettés par l'identité OCCAM du processus représenté, et sont décomposables en deux grands groupes :

1. les nœuds associés aux processus primitifs !, ?, :=, SKIP, STOP qui ont un nombre connu de fils dont la valeur dépend seulement de la nature du processus concerné.
2. les nœuds associés aux processus construits à l'aide des constructeurs PAR, SEQ, ALT, IF qui ont une descendance égale au nombre d'éléments présents dans le constructeur. Le type des nœuds fils est différent suivant la nature des constructeurs. Pour SEQ et PAR, les fils sont de type "bloc", alors que pour ALT et IF ils sont de type "cond_Bloc".

nœud cond_Bloc : Ce nœud possède trois fils :

1. un nœud "garde" contenant la garde ou la condition autorisant l'activation du processus présent dans le nœud code.
2. un nœud de déclarations locales (L_decl) au nœud Corps.
3. un nœud Code représentant le code à exécuter.

```
INT x:
INT y:
INT R:
PROC main ()
  INT z:
  SEQ
    z := 2*x*y
    PAR
      x := x * x
      y := y * y
    R := x + y + z
: -- fin Processus Main
```

Figure 4.5: procédure calculant R = $(x+y)^2$

Considérons le programme suivant qui réalise la fonction R = $(x+y)^2$. Considérons que le programme fourni en entrée est celui présenté en figure 4.5.

L'arbre syntaxique produit par la phase d'analyse est présenté en figure 4.6.

4.2.2 Phase de numérotation

La phase suivante, appelée phase de numérotation, est une phase importante du processus d'extraction du parallélisme puisque les structures de données utilisées par la suite y sont construites. Elle permet de compléter la structure construite précédemment en décorant chaque nœud de l'arbre avec l'information d'ordonnancement correspondante. Ainsi, nous n'avons plus besoin de reparcourir régulièrement l'arbre syntaxique pour pouvoir placer deux nœuds processus l'un par rapport à l'autre.

Règles de numérotation

Nous avons adopté une notation "." traduisant une concaténation de chaînes de caractères représentant des niveaux de branches dans l'arbre. Par convention, la racine de l'arbre est de niveau "adam". Voici les règles de construction des étiquettes à placer sur les principaux nœuds, en fonction du niveau du nœud père. Cette étiquette permet de connaître le niveau de l'ensemble de l'arbre placé sous ce nœud.

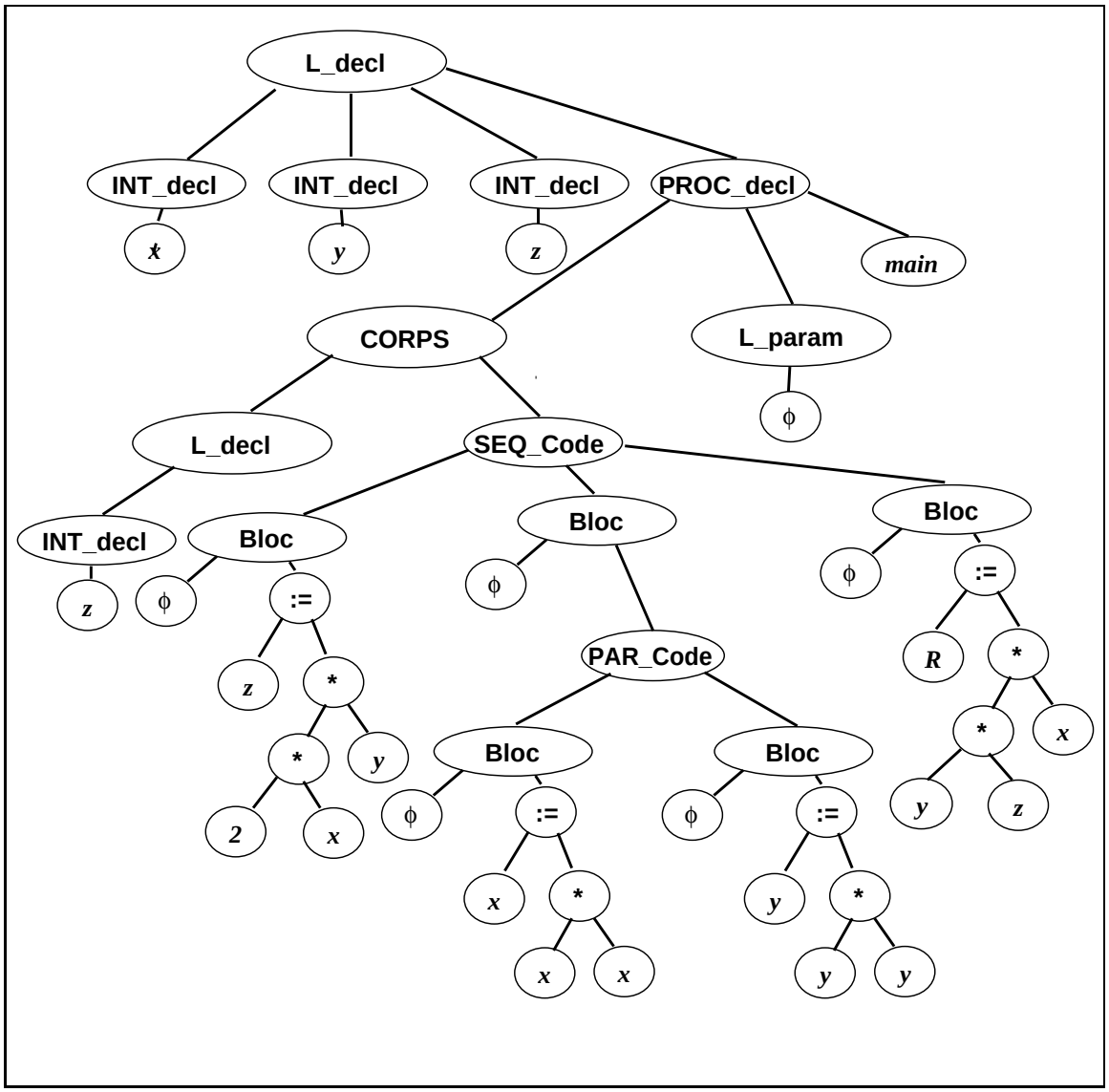


Figure 4.6: Arbre syntaxique produit par l'analyseur

Nœud adam : Niveau = adam.

Nœud L_decl: Niveau = Niveau_précédant.d

Nœud PROC_decl : Niveau = Niveau_précédant.i, où i désigne le fait que cette déclaration est la $i^{ème}$ du nœud L_decl.

Nœud L_param : Niveau = Niveau_précédant.par

Nœud param_decl : Niveau = Niveau_précédant.i, où i désigne le fait que ce paramètre est le $i^{ème}$ du nœud L_param.

Nœuds Bloc, Cond_Bloc : Niveau = Niveau_précédant.i, où i désigne le fait que ce bloc est le $i^{ème}$ du nœud père.

Nœud PAR_code : Niveau = Niveau_précédant.par

Nœud SEQ_code : Niveau = Niveau_précédant.seq

Nœud IF_code : Niveau = Niveau_précédant.if

Nœud ALT_code : Niveau = Niveau_précédant.alt

Nœud WHILE_code : Niveau = Niveau_précédant.while

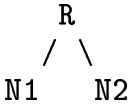
Nœud Corps : Niveau = Niveau_précédant

Notion d'ordre partiel entre les nœuds

Les nœuds constitués de processus simples sont des nœuds terminaux du point de vue des niveaux de numérotation. Ainsi, toutes les occurences d'identificateur présentes dans un nœud associé à un processus simple, sont répertoriées au niveau de ce nœud.

Lorsque le nœud est un processus constructeur, l'ordre entre ses fils est défini comme suit :

Soient R une racine commune entre deux nœuds N_1 et N2,



i et j deux numéros de processus fils d'un même constructeur,

R' et R" deux suites d'arborescence quelconques,

nous avons les propriétés suivantes :

si N1 = R.seq.i.R' et N2 = R.seq.j.R"

Et si i < j alors l'exécution du processus associé au nœud N1 est effectuée avant l'activation du processus associé à N2.

si N1 = R.par.i.R' et N2 = R.par.j.R"

Alors les deux exécutions des processus associés aux nœuds N1 et N2 s'effectuent en parallèle. Nous ne pouvons pas les placer l'une par rapport à l'autre.

si N1 = R.if.i.R' et N2 = R.if.j.R"

Le processus P_1 s'exécute seulement si la condition présente dans le nœud R.if.i.cond est évaluée à Vrai et que toutes les conditions R.if.k.cond, k = 1,...,i-1 sont fausses. De même pour le processus P_2. Ce que nous devons retenir ici est qu'un seul fils du IF est actif à la fois.

si N1 = R.alt.i.R' et N2 = R.alt.j.R"

Contrairement au IF, les processus fils d'un ALT sont choisis de manière égalitaire. Si deux gardes sont passantes en même temps, alors un choix aléatoire est effectué. Comme dans le cas du IF, un seul des fils est actif à la fois.

Le résultat de la numérotation sur l'arbre syntaxique obtenu précédemment est présenté en figure 4.7.

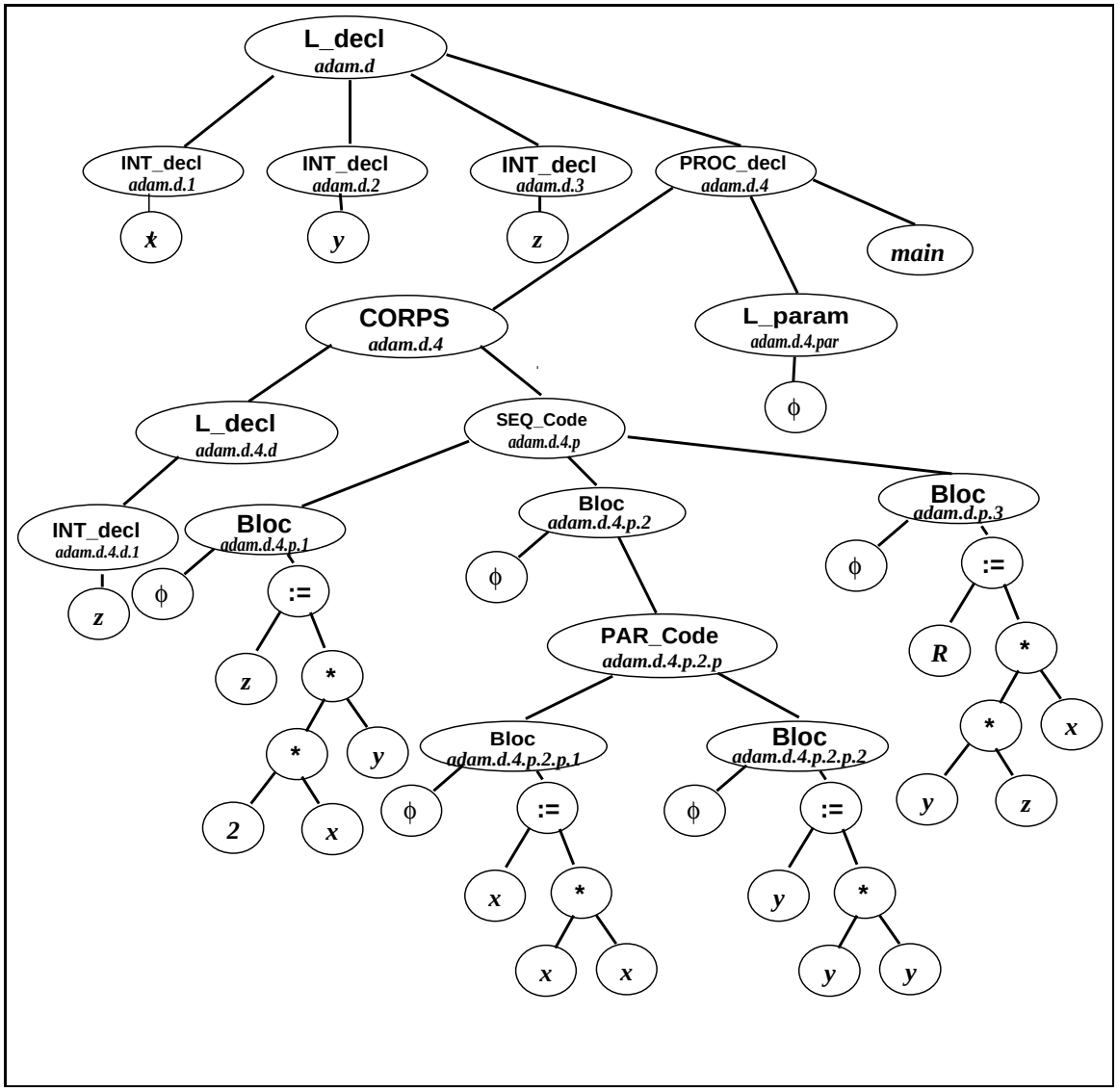


Figure 4.7: Arbre syntaxique décoré

Lors de la décoration de l'arbre, de nombreuses informations sur les déclarations et les utilisations des identificateurs sont collectées :

1. Pour la partie déclaration, une table des symboles contenant l'identification et le niveau de déclaration des variables est construite.
2. Pour les utilisations des identificateurs, nous maintenons pour chaque utilisation le type d'accès effectué (lecture ou écriture) et le niveau du processus effectuant cet accès. Pour les identificateurs de tableau, nous retenons aussi les expressions représentant les calculs d'index.

4.2.3 "Descente" des déclarations des identificateurs

Nous abordons maintenant les phases de transformation de l'arbre décoré. La première d'entre elles consiste à faire descendre les déclarations de variables au point le plus proche possible de leur utilisation. Ceci aura pour conséquence une diminution notable des environnements que se partagent plusieurs processus.

En effet, le programmeur OCCAM conçoit souvent son programme en utilisant les règles de portées de nom ; le programme obtenu est bien plus lisible que le programme équivalent composé de communications par messages entre les différents processus. En outre, l'utilisation abusive d'une telle règle fait souvent générer des programmes incorrects (partage de données en écriture,...). De plus, ces partages de données en lecture/écriture entre plusieurs processus concurrents constituent des obtacles importants à l'exécution de l'algorithme implémenté sur un processeur vers une architecture multiprocesseur sans mémoire commune.

Algorithme

Voici l'algorithme utilisé pour effectuer cette descente des variables :

Pour tout identificateur présent dans la table des déclarations effectuer les opérations suivantes :

1. Chercher toutes les occurences d'utilisation de cet identificateur dans les tables d'utilisation. Si l'identificateur en est absent, la déclaration est donc inutile, l'ôter de la table des déclarations. Dans ce cas, passer à l'identificateur suivant, sinon enchaîner.

2. Dans un programme OCCAM, le même identificateur peut être déclaré plusieurs fois. La dernière déclaration masque toutes les autres. Il convient de s'assurer que les occurences d'utilisation s'appliquent bien à cet identificateur. Ceci est facilement détectable puisque le niveau du nœud père de la déclaration doit être inclus dans le niveau associé à cet accès. Si la liste obtenue est vide, nous pouvons détruire cette déclaration.

3. Il faut déterminer ensuite le nœud racine commun à tous les nœuds utilisations obtenus dans la phase 2. Ceci est effectué de la manière suivante:

   (a) Choisir deux nœuds utilisations. Déterminer ensuite la racine commune "R_com" entre ces deux nœuds.

   (b) Pour chaque nœud utilisation restant, déterminer la racine commune entre R_com et le nœud courant utilisation.

4. La racine obtenue par la phase 3 est celle commune à tous les nœuds d'utilisation. Il suffit donc de trouver le nœud L_decl le plus proche de cette racine, d'y placer la déclaration et de mettre à jour l'ancien nœud L_decl.

Correspondance avec les Lois de [RH86]

Nous faisons ici un rapprochement avec les lois de transformation issues des travaux du Program Research Group.

La première étape de l'algorithme présenté ci-dessus permet d'éliminer les identificateurs déclarés, mais non utilisés dans le programme. Cette modification de programme correspond à l'application des lois 6.3 VARélimination_b et 6.13 CHANélimination_b (c.f. figure 4.8) qui permettent d'ôter toute déclaration d'identificateur pourvu que toutes les occurences de cet identificateur soient libres[1] dans le code des processus héritant cette déclaration.

Dans notre cas, la déclaration et le processus sont reliés entre eux par un nœud Bloc, Bloc_Cond ou Corps. L'obtention d'une liste d'utilisation vide en résultat de la phase 2, signifie que toutes les occurences de cet identificateur sont liées à l'intérieur du processus code frère du nœud L_décl contenant la déclaration que nous voulons déplacer. Ceci est la condition d'application des lois de transformation citées ci-dessus.

[1]Rappel: Un identificateur v est **lié** dans un processus P si toutes les occurences d'utilisation de v dépendent de déclarations internes à P. Dans le cas contraire, v est dit **libre**.

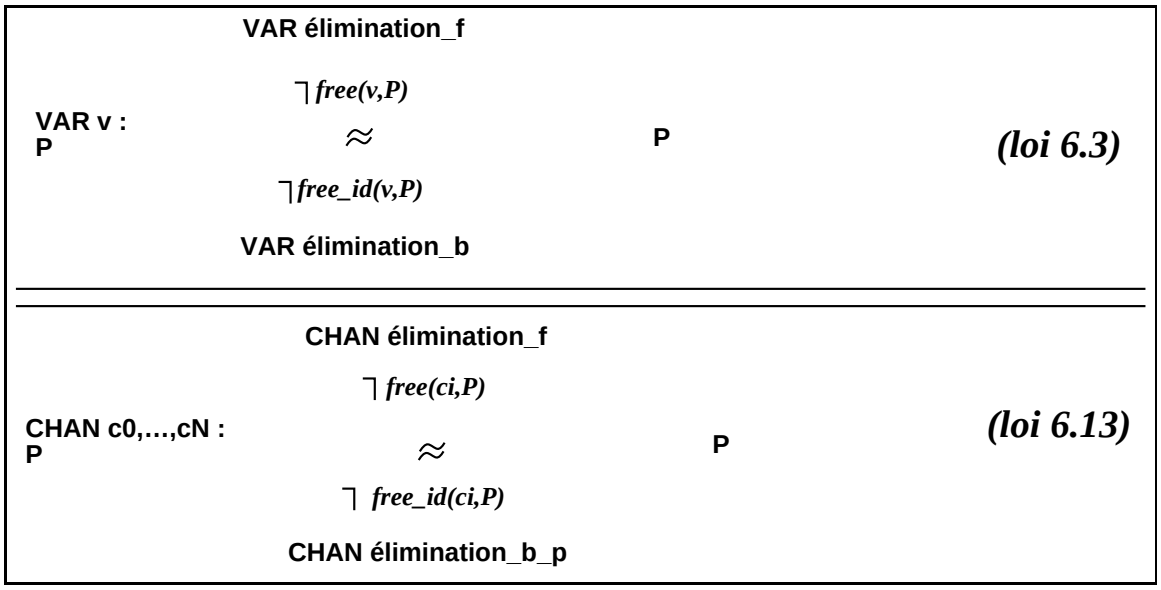


Figure 4.8: Lois 6.3 et 6.13

La seconde étape (Phase 3 de l'algorithme) consiste à trouver le nœud de l'arbre commun à un ensemble de processus. Pour cela, nous calculons la racine commune de l'ensemble obtenu par la phase 2. Cette racine désigne le nœud où nous descendrons la déclaration de l'identificateur. Cette approche est tout à fait similaire à l'utilisation des lois de [RH86] visant à déplacer des déclarations à travers un ALT (VARaltdistribution_b 6.5) (c.f. fig. 4.9), un IF (VARifdistribution_b 6.6) (c.f. fig. 4.9), un SEQ (VARseq1_b 6.7 et VARseq2_b 6.8) (c.f. fig. 4.10) et un PAR (VARpar_b 6.9) (c.f. fig. 4.11).

Le principal problème est de s'assurer de ne pas descendre "trop bas" une déclaration. Ceci aurait pour conséquence de rendre libres des occurences de l'identificateur en question alors qu'elles ne l'étaient pas auparavant. Ceci est impossible dans notre algorithme puisque nous arrêtons la descente au niveau du nœud père des processus qui font un accès à cet identificateur.

4.2.4 Quelques vérifications

Après avoir collecté de nombreuses informations sur le programme pendant la phase de décoration de l'arbre, nous pouvons maintenant effectuer un certain nombre de vérifications statiques :

1. Vérifier si les variables utilisées sont initialisées.
2. Tester la cohérence d'accès sur une donnée.

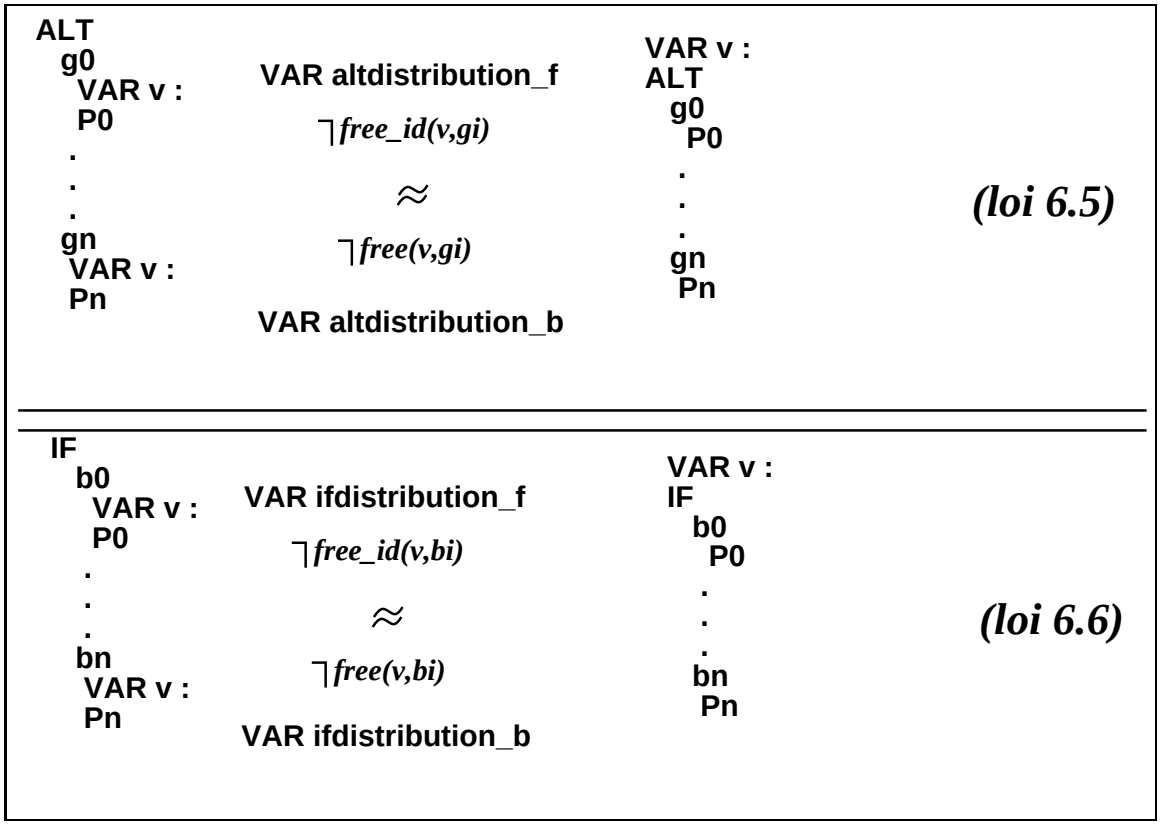


Figure 4.9: lois 6.5 et 6.6

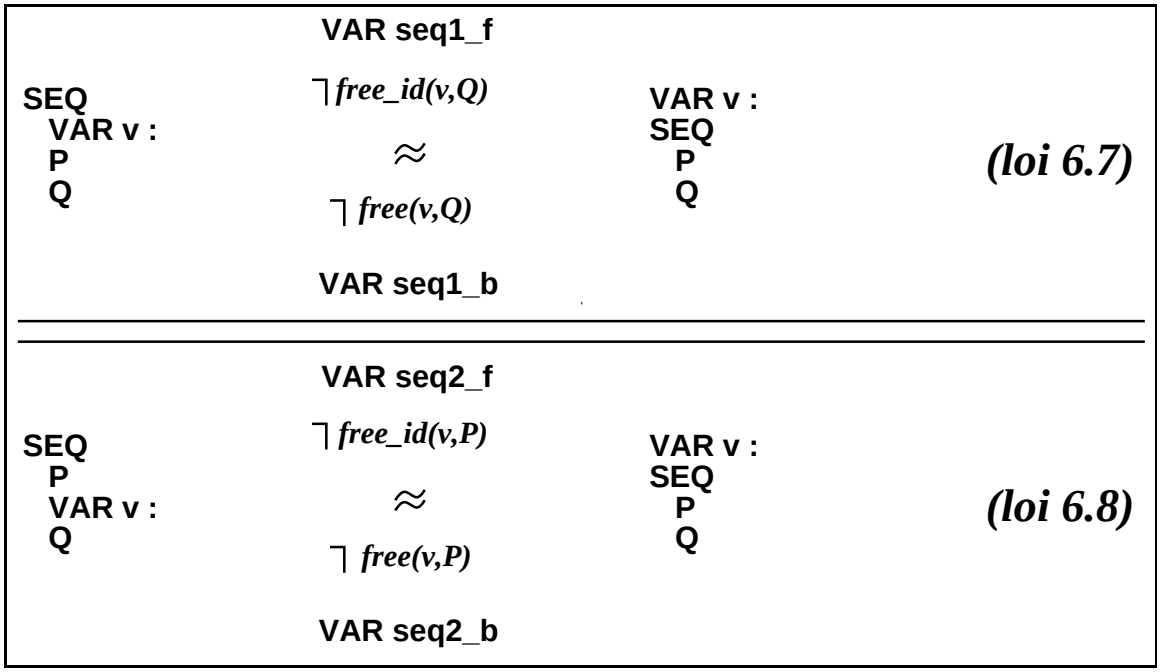


Figure 4.10: lois 6.7 et 6.8

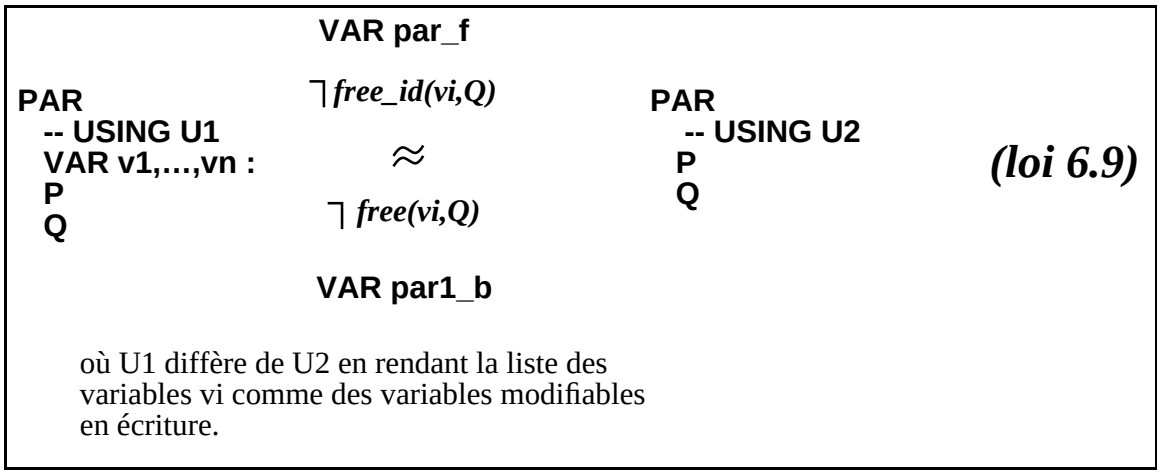


Figure 4.11: loi 6.9

3. Vérifier que chaque canal n'est utilisé qu'en connection bi-point.

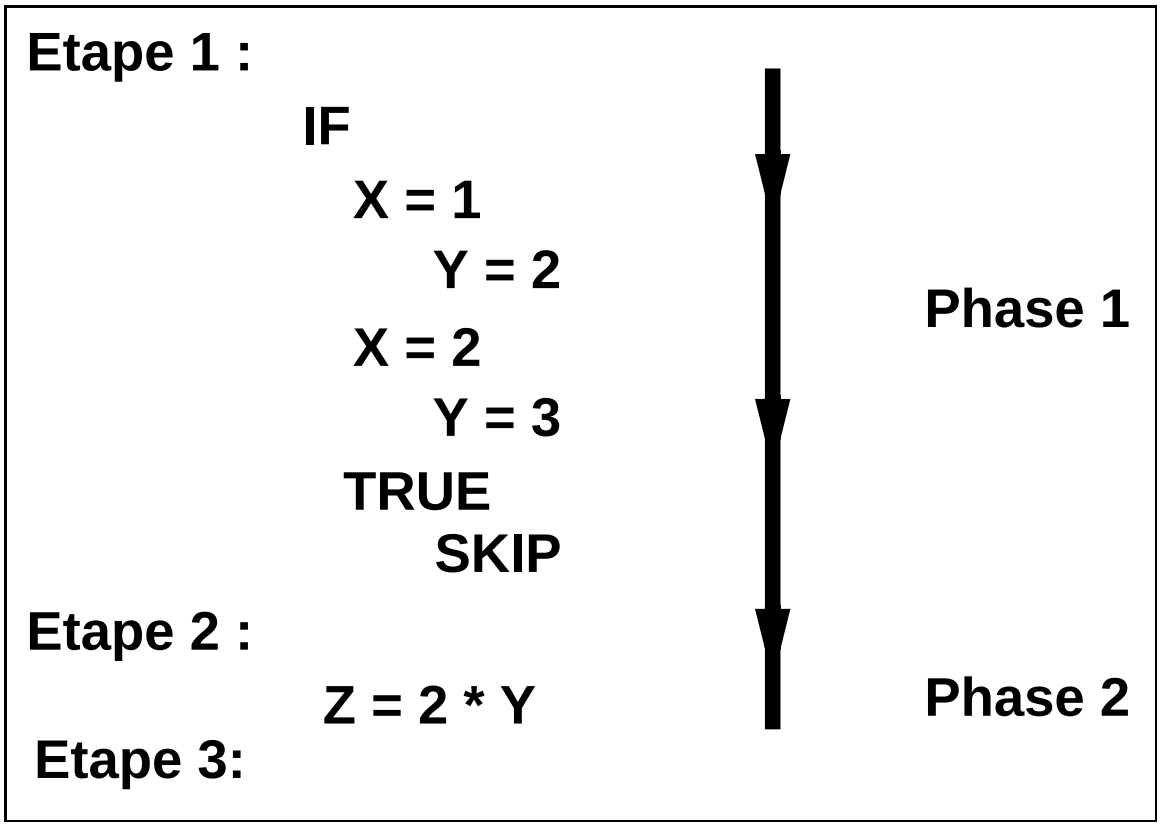
Utilisation de variables non initialisées

Les cas d'utilisation de variables non-initialisées sont détectables par le fait que la première occurence d'utilisation est une lecture. Ceci signifie que le premier élément de la liste des utilisations de cet identificateur est du type lecture. L'algorithme utilisé est le suivant :

Pour tous les identificateurs présents dans la table des déclarations effectuer les opérations suivantes :

1. Filtrer l'ensemble des occurences de manière à obtenir uniquement celles concernant la déclaration courante.
2. Rechercher parmi toutes les occurences présentes dans cet ensemble la plus proche du nœud déclaration, c'est à dire la première à s'exécuter.
3. Si ces occurences sont des lectures, nous avons détecté une erreur.

Cet algorithme permet d'éliminer une partie des erreurs dues à des variables non-initialisées. Il peut toutefois se produire le fait où la première occurence soit encapsulée par un constructeur IF ou ALT. Considérons l'exemple suivant :



En entrant à l'étape 1, considérons que la variable `X` soit initialisée et que la variable `Y` ne le soit pas. Pendant l'exécution de la phase 1 qui nous amène de l'étape 1 vers l'étape 2, `Y` peut être initialisée à 2 ou 3. Pendant la phase 2, la valeur de la variable `Y` est utilisée. Mais celle-ci est indeterminée dans le cas où `X` est différent de 2 et 1. La valeur de `Z` est donc succeptible d'être incohérente puisque `Y` peut être incohérent lui-aussi. Le programme doit donc être considéré comme incorrect, même si à l'exécution `X` vaut toujours 1 ou 2 et que celà ne soit pas décelable lors de la compilation.

Voici un algorithme prenant en compte les constructeurs IF et ALT.

Pour chaque identificateur I présent dans la table des déclarations effectuer les opérations suivantes :

1. Chercher la première occurence L en lecture de l'identificateur I
2. Fabriquer un ensemble O_e des occurences en écriture qui sont en amont de l'occurence de lecture L.
3. Si $O_e = \Phi$ alors CAS d'ERREUR
4. Sinon rechercher dans O_e, une occurence en écriture non sujette à un

constructeur IF ou ALT :

A Si cette occurence existe alors CORRECT

B Sinon il faut chercher une structure IF ou ALT qui possède dans toutes ses branches une initialisation de l'identificateur I :

a Si cette structure est trouvée alors CORRECT

b Sinon CAS d'ERREUR

Vérification statique des accès sur une variable

Dans un programme parallèle, nous devons nous assurer que des accès en écriture ne se font pas de manière concurente sur une même variable ou qu'un processus ne lit pas une donnée en cours d'écriture. Comme nous l'avons présenté dans le Chapitre 3, certains accès ne peuvent pas être vérifiés de manière statique. La phase de détection que nous présentons ici est utilisée lors de la génération de code pour la machine virtuelle. Le résultat de cette phase permet de savoir si les accès à telle ou telle variable sont incorrects, corrects ou à contrôler lors de l'exécution du programme. L'algorithme utilisé ici est le suivant:

Pour chaque identificateur déclaré, effectuer les tâches ci-dessous :

1. Construire une liste L d'occurences d'accès concernant l'identificateur en cours de traitement. Un filtrage est effectué si nécessaire pour assurer la règle de portée des noms.

2. Pour chaque occurence O de la liste L obtenue :

   (a) Oter O de la liste L.

   (b) Pour chaque occurence E de L faire :

      i. Les occurences O et E sont-elles sœurs? (fille d'un même PAR). Dans l'affirmative, il faut appliquer les règles de vérification présentées au chapitre 3. S'il n'y a pas incompatibilité, nous passons à l'occurence E suivante, sinon nous avons détecté une erreur.

      ii. Dans le cas où O et E ne sont pas sœurs, nous enchaînons avec l'occurence E suivante.

Vérification d'utilisation des canaux

Les vérifications effectuées sur l'utilisation des canaux sont simples. Nous nous assurons qu'il y a un correspondant dès lors qu'il y a une communication et que les communications sur les canaux se font suivant le modèle du Rendez-vous. Pour réaliser ces vérifications, nous collectons pour chaque déclaration de canal, toutes les occurences d'utilisation de ce canal. Ensuite, nous vérifions qu'il existe pour chacune d'entre elles un correspondant placé dans un nœud frère d'un constructeur PAR englobant. Si ce n'est pas le cas, nous avons détecté une erreur.

Le modèle du Rendez-vous indique que deux correspondants seulement peuvent communiquer sur un même canal simultanément. Si sur plusieurs branches frères d'un même constructeur PAR, il y a occurence d'un même canal et que ce nombre de branches est supérieur à deux, alors nous en déduisons que ce canal est utilisé par plus de deux correspondants. Il y a donc erreur.

4.2.5 Extraction du parallélisme

Comme nous l'avons présenté précédemment, l'objectif est d'obtenir un programme sous une forme canonique à partir d'un programme quelconque. Ne faire aucune hypothèse sur la forme que doit avoir le programme en entrée empêche d'obtenir de manière simple cette forme canonique.

Construction des états du programme

Pour construire ces états du programme, nous rencontrons de multiples problèmes. Le résultat de la phase de descente des déclarations de variables permet de repérer de manière simple les variables globales du programme : ce sont celles placées au niveau de déclaration "adam".

En prenant les valeurs de l'ensemble de ces données comme définissant l'état à un instant donné du programme, nous devons détecter les moments où ces états évoluent. Cette tâche est particulièrement ardue puisque nous pouvons aisément détecter l'évolution d'une seule donnée, mais qu'il est difficile d'effectuer des regroupements entre les évolutions de plusieurs données.

Nous entendons par là que trouver l'automate d'évolution d'une seule donnée dans l'ensemble du programme est simple, mais qu'essayer de re-

grouper l'ensemble des automates associés aux données, entraîne des problèmes pour positionner dans le temps l'évolution de telle donnée avant telle autre. En effet, il faut considérer simultanément l'ensemble de tous les automates des données globales car les différents états d'une variable peuvent interagir entre eux. De plus certains calculs intermédiaires sont échangés par des données tampons ou par des communications entre différents processus. Nous sommes obligés de tenir compte de ces échanges d'information pour ordonner les états entre eux en rajoutant des états que nous qualifions d'intermédaires.

Nous pensons qu'il faut opter plutôt pour une approche *Data_Flow*. Si nous construisons un graphe de flot de données entre les différents processus du programme initial, nous devrions obtenir des états observables du programme. Une étude est en cours dans ce sens.

Nous supposons donc que le programme fourni en entrée présente déjà la caractéristique principale de la forme canonique, à savoir que les états observables du programme sont définis par le programmeur (voir figure 4.12).

```
SEQ
  SEQ -- etat 0
    ...
  TRANSITION_0
  SEQ -- etat 1
    ...
  TRANSITION_1
  ...
  ...
  TRANSITION_N-1
  SEQ -- etat N
    ...
```

Figure 4.12: Squelette de programme imposé

Chaque transition `TRANSITION_i` correspond à un ensemble de processus OCCAM utilisant des valeurs de l'état i pour obtenir finalement l'état i+1. Nous allons nous attacher à extraire le parallélisme présent dans chacune des transitions de manière à obtenir un maximum de processus à placer sur les processeurs virtuels.

Extraction du parallélisme présent dans les transitions

Tout comme nous avons procédé lors de la descente des variables, nous voudrions utiliser un algorithme s'appuyant sur des lois de transformation.

Pour obtenir une forme canonique à partir du squelette de programme présenté ci-dessus (c.f. figure 4.12), nous devons transformer toutes les transitions du programme afin d'obtenir la forme suivante :

```
PAR
  Job_processeur_1
  ...
  Job_processeur_K
```

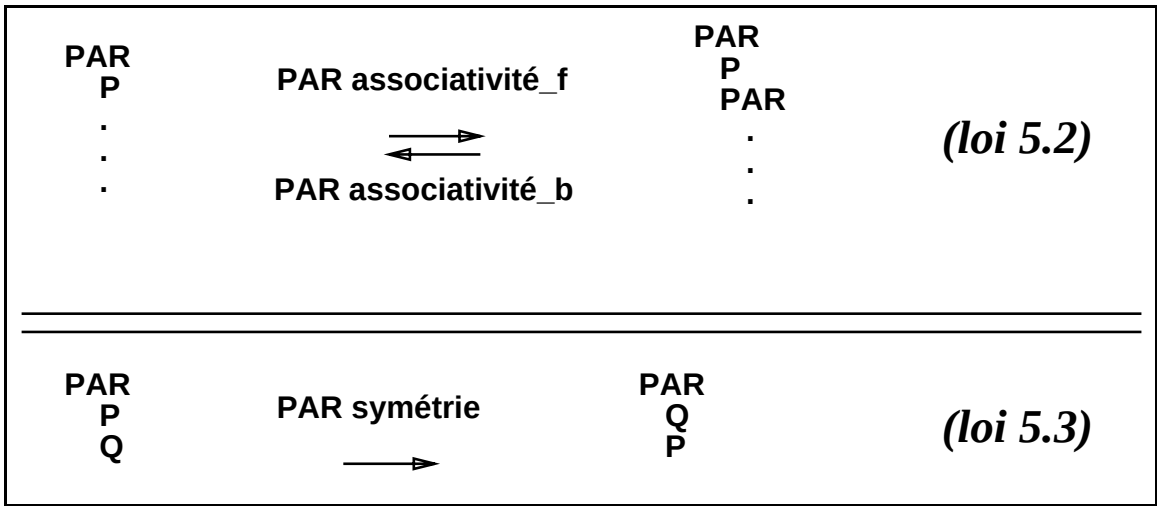


Figure 4.13: lois 5.2 et 5.3

Pour réaliser cette transformation, nous n'avons que peu de lois disponibles. Les seules lois présentées dans [RH86] succeptibles de nous intéresser sont (c.f. figure 4.13) :

1. La loi PAR_associativité_b (5.2) qui permet de désimbriquer des constructeurs PAR.
2. La loi PAR_symétrie (5.3) qui permet de permuter l'ordre d'écriture des éléments d'un constructeur PAR de manière à pouvoir appliquer la loi précédente.

Le nombre de lois à appliquer étant trop insuffisant, nous avons décidé d'en utiliser d'autres non issues de ce système de réécriture.

Autres transformations de programmes

Transformation WHILE - PAR Une transformation concernant le WHILE et le PAR a été présentée dans [Gol87] et nous intéresse particulièrement pour faire "remonter" du parallélisme interne présent dans une boucle WHILE. Considérons un programme écrit sous la forme :

```
WHILE e
CHAN C:
  PAR
    P
    Q
```

Nous voulons obtenir un programme ayant le squelette suivant :

```
PAR
  WHILE e_P'
    P'
  WHILE e_Q'
    Q'
```

Pour ce faire, lors de l'évaluation de l'expression booléenne `e`, peut se poser le problème suivant : certaines données modifiables par `P` et `Q` peuvent être présentes dans cette expression. Si nous décidons que l'évaluation de e est effectuée dans le contexte du processus `Q`, il suffit de repérer l'ensemble des données utilisées par `P` succeptibles de faire évoluer la valeur de `e`.

Reprenons les notations introduites dans [Gol87] permettant de désigner les ensembles de données associés au processus P :

Lp : est l'ensemble des données locales au processus P.

Ip : désigne l'ensemble des données dont la valeur durant l'exécution de P, peut affecter le comportement du processus P.

Op : désigne l'ensemble des données dont la valeur après exécution de P peut être dûe à cette exécution.

Appelons A l'ensemble des données référencées dans l'expression booléenne `e`. Le résultat de la transformation introduite dans [Gol87] est illustré en figure 4.14.

```
CHAN C,init,Data,Bool :
PAR
  SEQ
    init ! Ip Λ A -- emission des valeurs des donnees communes
                  -- a l'evaluation de e et suceptibles de
                  --  modifier le comportement de P.
    Bool ! e      -- emission valeur initiale de e.
    WHILE e
      SEQ
        Q
        Data ? Op Λ A -- reception des modifications effectuees
                      -- par P sur les donnees presentes dans e.
        Bool ! e      -- evaluation de e et emission vers P.
  VAR Fp Λ A, loop :
  SEQ
    init ? Ip Λ A -- reception des donnees succeptibles de
                  --  modifier le comportement de P
  --  presente dans e.
    Bool ? loop   -- reception de la valeur initiale de e.
    WHILE loop
      SEQ
        P
        Data ! Op Λ A -- emission des modifications des donnees
                      -- communes a P et e.
        Bool ? loop   -- reception de la valeur courante de e
                      -- evaluee par le processus executant Q.
```

Figure 4.14: Transformation WHILE-PAR

Transformation SEQ - PAR D'autres transformations possibles consistent à tenter de rendre parallèles les éléments d'un constructeur SEQ. Considérons que nous ayons un programme présenté sous la forme de la figure 4.15.

```
SEQ
  P
  Q
```

Figure 4.15:

Nous pouvons construire un PAR "séquentialisé", en détectant les données issues de l'exécution de P nécessaires à celle de Q. Le processus P émettra, sur un canal commun à P et Q , les données utiles à Q. Ces identificateurs appartiennent donc à $O_p \bigcap I_Q$. Cette transformation présente un intérêt puisque nous pouvons faire ainsi apparaître un constructeur PAR alors qu'il n'en existait qu'un SEQ. Ceci permet une application ultérieure des lois présentées dans [RH86]. Le résultat de la transformation appliquée sur le programme précédent (fig. 4.15) est illustré en figure 4.16.

```
CHAN P_VERS_Q :
PAR
  SEQ
    P
    P_VERS_Q ! Op Λ Iq
  SEQ
    P_VERS_Q ? Op Λ Iq
    Q
```

Figure 4.16: PAR séquentialisé

Une autre transformation est possible dans le cas suivant. Si le processus Q n'utilise pas de variables modifiables par P, i.e. $I_q \bigcap O_p = \phi$ et que le processus Q ne tente pas d'écrire sur des données utilisables par P, i.e. $I_p \bigcap O_q = \phi$ et que les processus P et Q ne modifient pas une même donnée, i.e. $O_p \bigcap O_q = \phi$, alors les deux processus sont totalement indépendants et nous pouvons transformer le constructeur SEQ en PAR comme présenté en figure 4.17. Le processus PAR obtenu exprime un parallélisme "réel" sans synchronisation entre P et Q. C'est un cas qu'il faut essayer de détecter le plus souvent possible puisqu'il fait apparaître du parallélisme non exprimé par le programmeur.

```
PAR
  P
  Q
```

Figure 4.17:

Une troisième transformation est possible dans le cas où P utilise des données succeptibles d'être modifiées par Q, i.e. $I_p \bigcap O_q \neq \phi$, et que Q ne modifie pas des données en même temps que P (et vice-versa), i.e. $I_q \bigcap O_p = \phi$ et que les processus P et Q ne modifient pas une même donnée,
i.e. $O_p \bigcap O_q = \phi$. Il suffit de faire une copie locale dans l'environnement de P des identificateurs présents dans $I_p \bigcap O_q$ dont la valeur risque d'être pertubée par l'exécution du processus Q. Nous obtenons ainsi la transformation montrée en figure 4.18.

```
CHAN FROM_Q :
PAR
  VAR Copies_Locales : -- Ip Λ Oq
  SEQ
    FROM_Q ? Copies_Locales
    P
  SEQ
    FROM_Q ! Ip Λ Oq
    Q
```

Figure 4.18:

4.2.6 Génération de code PVM et Extraction d'informations pour l'allocateur

Ces deux derniers modules de l'extracteur réalisent l'interface avec les autres éléments de la chaîne de développement PDS.

Le programme OCCAM obtenu après les transformations évoquées précédemment doit être traduit en code PVM qui est l'une des entrés du **générateur de code**.

Ce programme OCCAM apparaît sous la forme canonique introduite dans le Chapitre 3. L'**allocateur**, dont la structure est définie dans le Chapitre 5,

doit effectuer un placement des processus virtuels sur une topologie de processeurs. Pour réaliser celui-ci, l'allocateur a besoin d'un ensemble d'informations que nous devons lui produire. Le lecteur en trouvera une description dans le chapitre consacré à l'allocation.

Chapitre 5

Allocation de processus aux processeurs

Après avoir présenté dans une première partie le problème de l'allocation d'un ensemble de processus sur l'une des topologies d'un multiprocesseur, nous détaillons quelques solutions lorsque cette allocation est effectuée de manière **statique**. Ces solutions se basent sur deux types d'algorithmes :

1. par partitionnement de graphes
2. par des méthodes évolutives comme le recuit simulé.

Dans la troisième partie, nous abordons une autre manière d'appréhender ce problème d'allocation : répartir de manière **dynamique** la création des processus. Nous présentons les problèmes rencontrés lors de la mise en œuvre d'une telle solution.

Nous détaillons enfin la structure globale de l'allocateur telle nous l'avons conçue dans PDS.

5.1 Présentation du problème

Le but du placement est la recherche d'une allocation optimale (dans un sens précisé plus loin) des ressources que sont les processeurs d'une machine parallèle.

Le temps global d'exécution d'un programme parallèle dépend du temps d'exécution de chaque processus. Celui-ci peut être scindé en trois parties :

1. Exécution de code réalisant une partie de l'algorithme du programme.

2. Communication avec d'autres processus partenaires.

3. Synchronisation nécessaire à l'établissement de la communication.

La première part de ce temps correspond au temps de calcul utile, et sa part dans le temps total est proportionnelle aux données à traiter. En revanche, les deux autres parts peuvent occuper un temps plus ou moins important suivant le placement choisi. En effet, le temps passé à communiquer est nettement moins coûteux lorsque les deux processus sont placés sur le même processeur. Si ce n'est pas le cas, la charge des liens de communications inter-processeurs entre en ligne de compte. L'importance de la charge des liens de communications croît si les deux processus ne sont pas placés sur deux processeurs voisins ; il apparait ainsi une notion de distance inter-processus.

Enfin, le temps passé à attendre l'établissement de la communication dépend fortement de la nature des processus. Cependant, évaluer ce temps d'attente est une tâche difficile sinon impossible en général.

La seule évaluation que sachent apprécier les algorithmes d'allocation est le temps passé à communiquer et celui passé à dérouler du code. Le temps de mise en communication n'est jamais utilisé puisque difficilement évaluable.

Considérons que nous ayons p processus à placer sur une architecture parallèle composée de n processeurs. La recherche d'un meilleur placement se ramène à l'obtention d'un meilleur élément parmi l'ensemble de toutes les applications de P -ensemble des processus-, vers A -ensemble des processeurs. Ceci permet d'évaluer la complexité du problème puisqu'il y a $Card(A)^{Card(P)} = n^p$ possibilités à étudier.

Un bon placement doit assurer une réduction du temps global du programme à placer. Dans ce but, le critère principal est la minimisation des coûts de communication tout en effectuant un équilibrage de la charge des processeurs. Cette minimisation est connue pour être un problème NP complet. Il a été prouvé dans [GJ79] qu'aucun algorithme réalisant cette minimisation ne peut obtenir une solution exacte en un temps maximum borné par une fonction polynomiale de la taille du problème. En effet, vouloir minimiser les coûts de communication tout en assurant assurer un équilibrage de la charge de tous les processeurs est équivalent au problème de partitionnement d'un graphe dans lequel on exige que le poids de chaque nœud soit

égal [HR73].

Comme une solution optimale ne peut pas être atteinte de manière efficace, les algorithmes de placement vont plutôt chercher à obtenir une bonne solution approchant le plus possible une solution optimale. Différentes approches sont utilisables :

1. Méthodes statiques :
   (a) Méthodes par le partitionnement de graphes.
   (b) Méthodes évolutives : Mécanique statistique et méthode du recuit simulé.
2. Méthodes dynamiques.

D'autre part, lorsqu'on a obtenu un bon placement des processus sur le réseau de processeurs (que ce soit par un algorithme statique ou dynamique), il faut se poser le problème du multiplexage/démultiplexage des voies de communication physiques d'un processeur sur lequel ont été placés plusieurs processus voulant communiquer avec l'extérieur.

5.2 Allocation statique

5.2.1 Modélisation du problème

Le réseau de processeurs et le réseau de processus sont définis ainsi :

- Le réseau physique de processeurs est représenté par un graphe connexe $G_m = (V_m, E_m)$. E_m, l'ensemble des sommets, désigne les processeurs et V_m, l'ensemble des arêtes, représente les connections physiques inter-processeurs. Ce graphe est non-orienté puisque les connections inter-processus sont bi-directionnelles. Souvent, les arêtes de ce graphe sont pondérées par le coût de communication inter-processeurs et les sommets par le coût d'utilisation du processeur.
- Le réseau de processus est modélisé par un graphe $G_p = (V_p, E_p)$, où V_p désigne les processus à placer et E_p, les communications inter-processus. Les communications inter-processus étant bidirectionnelles, ce graphe est lui aussi non-orienté. Les arêtes peuvent être pondérées par les coûts de

communications inter-processus et les sommets par le temps d'exécution des processus.

Le placement statique de processus sur des processeurs est une mise en correspondance du réseau de processus avec le réseau de processeurs au moment de la compilation du programme ou au cours du chargement de ce programme. Le problème du placement se ramène, dans cette méthode, à une transformation de graphe. Il faut appliquer une succession de transformations sur le graphe des processus de manière à le "plier" sur le graphe des processeurs. Ce "pliage" consiste à placer les sommets du graphe des processus sur les sommets du graphe des processeurs suivant des critères visant à minimiser le temps total d'exécution de l'application.

Ceci est souvent formulé de la façon suivante :

$$\text{Il existe une projection } f \text{ de } G_p(X_p, E_p) \text{ dans } G_m(X_m, E_m)$$

$$\text{si et seulement si}$$

$$\exists f : X_p \to X_m \text{ tel que } \forall p_1, p_2 \in X_p, (p_1, p_2) \in E_p \Rightarrow \begin{cases} f(P_1) = f(P_2) \\ \text{ou} \\ (f(P_1), f(P_2)) \in E_m \end{cases}$$

5.2.2 Allocation statique par partitionnement

L'approche utilisée dans la transformation de graphes est le regroupement d'un certain nombre de processus de manière à obtenir un nombre de groupes de processus inférieur ou égal au nombre de processeurs présent sur le multiprocesseur. Le groupement des processus doit se faire de manière rationnelle, et se base sur la constatation suivante :

- Lorsque deux processus qui communiquent entre eux sont placés sur le même processeur, la durée de communication est plus courte que si les deux processus étaient placés sur des processeurs distincts.

D'où l'idée de partitionner les N processus en P groupes de façon à ce qu'un processus communique **beaucoup** avec les processus de son groupe et **assez peu** avec ceux d'autres groupes, c'est à dire que la coupure du graphe des groupes soit minimale.

Il est donc nécessaire de pouvoir évaluer le temps passé à communiquer. C'est la partie finale du travail de la phase d'analyse du programme. Une fois

ces informations extraites par l'analyseur, chaque algorithme de regroupement choisit une fonction coût permettant de savoir si un processus placé dans un groupe donné est mieux situé que placé dans un autre groupe. La fonction coût caractérise entièrement le type d'allocation que va effectuer l'algorithme.

Voici quelques fonctions coûts possibles pour des machines multiprocesseurs, dont les processeurs sont identiques :

1.

$$f_1 = \sum_{l,q_{l \neq q}} \sum_{i,j_{i \neq j}} C_{ij} D_{lq} X_{il} X jq$$

où

$i, j \in P$ ensemble des processus.

$l, q \in M$ ensemble des processeurs.

D_{lq} est le coût de communication entre les processeurs l et q.

C_{ij} est le coût de communication entre les processus i et j.

$X_{il} = 1$ si le processus i est placé sur le processeur l, 0 sinon.

Cette fonction conduit forcément à la solution triviale de minimisation du temps de communication entre les groupes de processus en plaçant tous les processus sur un même processeur, c'est à dire dans le même groupe. Il est donc nécessaire de prendre en compte le temps d'exécution des processus sur tel ou tel processeur.

2.

$$f2 = f1 + \sum_{l} \sum_{i} e_{il} X_{il}$$

où

e_{il} désigne le coût d'exécution du processus i sur le processeur l.

$X_{il} = \begin{cases} 1 & \text{si le processus } i \text{ se trouve sur le processeur } l \\ 0 & \text{sinon} \end{cases}$

Cette fonction permet d'éviter la solution triviale présentée précédemment, puisqu'elle empêche l'obtention de solutions où existent des processeurs inactifs. Mais la charge des processeurs n'est pas prise en compte ici.

3. La fonction suivante permet de tenir compte des coûts d'interférences dûs au placement conjoint de processus sur un même processeur se trouvant ainsi obligé de simuler l'exécution parallèle de processus. Ce coût est appelé coût de perte de parallélisme réel.

$$f3 = f2 + \sum_{l} \sum_{i,j_{i \neq j}} x_{il} x_{jl} B_l$$

où

B_l est le coût d'interférence dû au placement de deux processus sur le même processeur l.

Cette fonction permet de mieux répercuter les effets du partage de la puissance du processeur pour simuler le parallélisme.

D'autres critères sont à retenir suivant les caractéristiques du processeur employé :

1. La taille mémoire d'un processeur nous fournit la contrainte suivante :

$$\sum_{i} S_i X_{ik} \leq M_k$$

où

S_i désigne la taille mémoire nécessaire à l'exécution du processus i .

$$X_{ik} = \begin{cases} 1 & \text{si le processus } i \text{ s'exécute sur le processeur } k \\ 0 & \text{sinon} \end{cases}$$

M_k est la taille mémoire du processeur k.

2. Le nombre maximum de processus pouvant s'exécuter en parallélisme simulé sur un processeur.

$$\sum_{i} X_{ik} \leq N_k$$

où

N_k désigne le nombre maximum de processus présents sur le processeur k.

5.2.3 Allocation statique par méthode évolutive

Recuit simulé

La méthode du recuit simulé est une méthode bien connue pour obtenir le minimum global d'une fonction ayant des minima locaux [KCGV83]. Bien qu'elle ne puisse pas localiser de manière exacte ce minimum global, cette méthode permet de l'approcher dans un délai de calcul raisonnable.

Le recuit simulé [AL85] est une version mathématique du refroidissement physique. Si un liquide est refroidi rapidement, les cristaux du solide obtenu ont de nombreuses imperfections. Celles-ci correspondent à des niveaux d'énergie plus élevés que le niveau d'énergie minimal d'une structure cristalline parfaite. Si ce liquide est refroidi plus lentement, les imperfections diminuent, et le solide obtenu a une structure plus proche de celle du niveau d'énergie minimal.

Comme dans le recuit naturel, où l'état d'énergie d'un liquide est réduit pendant un intervalle de temps donné, le recuit simulé associe une température et une durée de réfrigération à une fonction à minimiser. De longs intervalles de refroidissement induisent des valeurs de fonctions plus petites et requièrent des recherches plus exhaustives sur le domaine de la fonction.

Une technique de minimisation naïve consiste à considérer un point initial du domaine de la fonction et à faire varier de manière aléatoire les valeurs des variables indépendantes de la fonction. Parmi ces valeurs sont acceptées uniquement celles qui diminuent la valeur de celle-ci. Si toutes les variations possibles couvrent entièrement le domaine de la fonction, cette solution est sûre de converger vers le minimum global moyennant un temps de calcul important.

L'efficacité de cette recherche aléatoire peut être améliorée en acceptant certaines données indépendantes qui augmentent la valeur de la fonction. Cette notion est le cœur de la méthode Metropolis Monte Carlo [MRRT53]. Précisément, toutes les modifications réduisant la valeur de la fonction sont acceptées et celles l'augmentant d'une valeur de $\Delta\ S$ sont acceptées avec la probabilité $e^{\frac{-\Delta S}{T}}$ où T est la température. Ceci permet de sortir d'un minimum local. Lorsque la température décroît, la probabilité de sélectionner de grandes valeurs de la fonction diminue.

La méthode Metropolis [MRRT53] a montré que la probabilité de passer d'un point i à un point j du domaine de la fonction est la même que celle du

point j vers le point i. Dans la mécanique statistique, ceci implique que la signification à long terme de toute quantité est la moyenne thermodynamique de cette quantité à la température choisie. Au cas limite, une température au zéro absolu contraint l'intervalle thermodynamique à une seule valeur minimale. Pour le profane, ceci signifie que la méthode Metropolis Monte Carlo est une version exacte du refroidissement réel ; si la température est réduite progessivement vers zéro, nous avons une convergence vers le minimum global de la fonction.

Critère de minimisation

[JOS86] propose le critère suivant :

$$S = S_{calcul} + S_{communication} = \sum_{i=1}^{N} W_i^2 + \frac{t_{communication}}{t_{calcul}} \sum_{p<q} C_{pq}$$

où N est le nombre de processeurs dans la machine,
W_i la charge de chaque processeur i,
C_{pq} le coût de communication entre des processus p et q en fonction de leurs placements,
$t_{communication}$ le temps nécessaire pour permettre à deux processus de communiquer,
et t_{calcul} la durée de calcul d'un processus.

Comme la fonction $\sum_{i=1}^{N} W_i^2$ est minimale lorsque tous les W_i sont égaux, le premier terme de la fonction représente un équilibrage de la charge des processeurs. Le second terme représente le coût de communication dans un réseau maillé. D'autres fonctions de coût de communication sont envisageables.

Implémentation de l'algorithme

En considérant la fonction à optimiser définie ci-dessus, une itération de l'algorithme de recuit simulé consiste à :

- Déplacer temporairement un processus placé sur un processeur vers un autre processeur. Ceci crée une nouvelle partition du réseau de processus sur le maillage de processeurs.
- Calculer le ΔS correspondant à une évolution de la fonction à minimiser S.

- Si $\Delta S < 0$, la modification effectuée diminue la valeur de S. La nouvelle partition est meilleure que la précédente, et devient la partition courante.
- Si $\Delta S > 0$, la modification est acceptée avec la probabilité $e^{\frac{-\Delta S}{T}}$, et devient la partition courante.

Les informations à présenter en entrée de cet algorithme sont :

1. un placement des processus sur le multiprocesseur,
2. une température initiale T,
3. une variation de cette température en fonction du temps.

Le choix de telle ou telle température initiale, et le nombre d'itérations à effectuer avant de changer la valeur de T ne peuvent être déterminés que de manière empirique. La qualité d'un placement peut être vérifiée seulement par exécution de l'algorithme de recuit simulé avec plusieurs températures initiales, différentes valeurs du nombre d'itérations, et des intervalles différents de descente de température. Ceci est une limite importante de cette méthode.

5.3 Allocation dynamique ou adaptative

Les méthodes utilisées par les algorithmes statiques ne sont pas utilisables en cas de placement dynamique pour les raisons suivantes :

1. L'algorithme d'allocation dynamique connait peu d'informations sur le programme utilisateur. C'est à lui d'effectuer les mesures sur chaque processeur pour les utiliser.
2. Cet algorithme ne doit pas être coûteux en termes de temps d'exécution processeur, d'occupation mémoire et de communication sur les liens inter-processeurs.

L'allocation dynamique permet au programme d'être réparti sur l'ensemble des processeurs pendant son exécution. Cette répartition qui se fait à chaque création de processus, doit équilibrer "au mieux" la charge de travail de tous les processeurs.

Le processeur qui effectue la demande de création d'un processus peut décider du sort de celui-ci en fonction des paramètres suivants :

- Sa propre charge de travail.
- La charge de travail que va apporter ce processus au processeur sur lequel il sera créé.
- Le taux d'occupation mémoire de tous les processeurs de la machine.
- La charge de travail des autres processeurs.
- La charge des liens de communication inter-processeurs.

Les paramètres locaux, à savoir le taux d'occupation du processeur, le taux de remplissage de la mémoire associée à ce processeur et la charge des liens de communications sont obtenus à coût relativement faible. Mais si nous voulons maintenir sur chaque processeur un état global de tout le réseau de processeurs, nous nous trouvons confronté aux problèmes suivants :

1. Importance par la taille des tables mémorisant l'état global.
2. Maintien de la cohérence de ces tables ; le temps de mise à jour d'une valeur de charge d'un des processeurs par exemple est non négligeable étant donné qu'il faut diffuser la nouvelle valeur dans tout le réseau.
3. Surcharge du réseau de communication inter-processeurs par les messages de nouvelles valeurs de charges transitant de processeur en processeur.

La maintenance de l'état global du multiprocesseur sur chaque site n'étant pas viable pour des raisons évidentes de performances, nous allons seulement conserver, pour chaque processeur, ce qui dépend des processeurs voisins de celui-ci et que nous pouvons maintenir en permanence à faible coût :

- Le taux d'occupation de la mémoire du processeur.
- Le taux d'occupation de ce processeur lié au nombre de processus présents sur ce site.
- La charge des voies de communication.

La demande de création de processus doit exprimer les doléances suivantes :

- Taille de l'espace mémoire nécessaire.

- Charge de travail supplémentaire induite par l'activation du processus.

- Charge de communication induite sur les liens de communication inter-processeurs.

A l'aide de ces paramètres, l'allocateur présent sur chaque processeur peut dérouler son algorithme de placement en interrogeant éventuellement les autres processeurs pour connaître les paramètres manquants.

Certains algorithmes [ELJ84,Ath87] éludent le problème d'évaluation de la charge en effectuant tout simplement des placements aléatoires de processus créés. D'autres associent statiquement à chaque processeur un groupe de processeurs que l'allocateur peut choisir [WM85]. Une autre politique peut consister à choisir le processeur suivant un ordre prédéterminé (service cyclique) [CK79].

L'avantage de ces trois styles d'allocation dynamique est leur simplicité. Mais leur désavantage majeur est la non prise en compte du coût de migration d'un processus face à la charge du processeur courant et à la charge du système de communication.

Le principe des enchères est une méthode bien connue dans les algorithmes de distribution de charge. Lorsqu'un processeur traite une demande de création de processus, une requête d'enchères contenant les caractéristiques du processus à créer est diffusée à tous les processeurs du réseau. A la réception de cette requête, un processeur envoie son "prix" d'exécution en fonction de son état local (charge, ...) vers le processeur demandeur. Lorsque le processus qui a lancé l'enchère a reçu tous les prix, il choisit le processeur au prix le plus bas, qui correspond à une activité moindre de ce processeur par rapport à tous les autres.

Plusieurs algorithmes de ce type ont été implémentés [SS84,Smi88]. Dans [HCG*82], une variante intéressante est à présenter. Lorsque le meilleur prix est déterminé, le processus à créer devrait être transféré vers le processeur acheteur. Mais il est possible que ce dernier devienne surchargé vu qu'il pourrait être le gagnant de nombreuses enchères. Pour éviter ce genre de situation, lorsque le processeur enchérisseur a choisi le processeur gagnant, un message d'annonce est expédié à celui-ci qui peut ainsi accepter ou décliner le processus proposé suivant l'évolution de son état interne (augmentation de la charge).

Le point le plus critique de cette méthode est que le système de communication doit être particulièrement efficace au niveau de la diffusion des messages. Malheureusement, dans les multi-processeurs où le réseau de connection est bi-point, la diffusion est un problème non trivial assez coûteux à résoudre.

Dans tous les cas il faut déterminer ce que nous appelons le seuil de migration d'un processus. En effet, dans certains cas le processeur sur lequel se fait la demande de création de processus ne peut pas l'exécuter sur place pour des raisons diverses comme saturation de l'espace mémoire, table des processus pleine, etc... Il faut absolument expatrier ce processus. Dans les autres cas, un choix est à faire entre une migration ou une exécution sur place. Si le coût de migration d'un processus est plus important que celui de l'exécution locale, alors ce processus ne doit pas être expatrié.

Le coût de migration est fonction des paramètres suivants :

- La charge de travail qu'apporte ce processus (occupation mémoire, ...)
- La charge du système de communication.
- La charge du processeur courant qui grossit puisqu'il faut émettre ce processus vers un lien extérieur.
- La charge du processeur récepteur qui obtient plus ou moins rapidement le processus à créer.
- La charge des processeurs qui effectuent le routage des messages dans le cas où les deux processeurs ne sont pas voisins.

Citons l'algorithme de [ELG89] qui est un algorithme distribué de répartition de charge. Chaque processeur exécute le même algorithme et joue ainsi le même rôle. Cet algorithme est particulièrement robuste puisque la panne d'un processeur ne fait que dégrader les performances totales de la machine. Le point le plus intéressant qu'il présente à nos yeux est sa simplicité qui permet d'envisager son expérimentation sur une machine à topologie reconfigurable dynamiquement.

Le processus d'allocation est composé ici de deux éléments de base :

1. **Un élément d'information** qui maintient l'état courant du système distribué.

Le critère retenu pour estimer la charge d'un processeur est le nombre de processus en attente d'exécution. Cette valeur est fortement corrélée avec le taux d'utilisation du processeur. L'avantage de ce critère est sa facilité d'évaluation. De manière à condenser cette information, trois états de charge sont créés :

(a) GRANDE CHARGE.

(b) CHARGE NORMALE.

(c) PETITE CHARGE.

Les différentes transitions entre ces états sont présentées en figure 5.1 et permettent de définir deux seuils : Tmin et Tmax. Ceux-ci permettent d'exprimer le nombre maximum et minimum de processus représentant une charge normale du processeur.

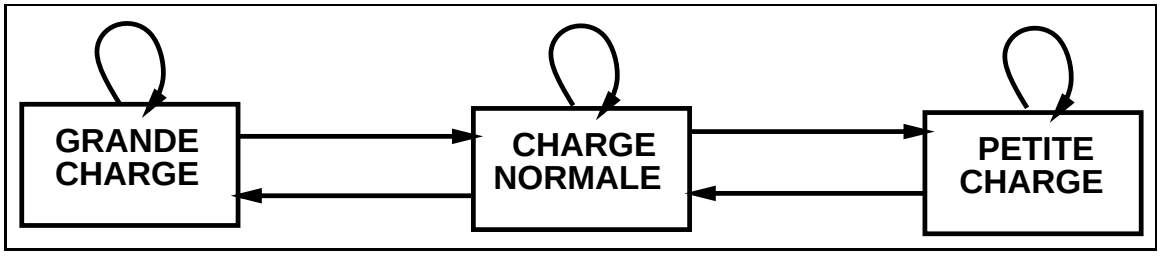


Figure 5.1: transitions d'états de charge d'un processeur

2. **Un élément de contrôle** qui utilise l'élément d'information pour savoir où placer les processus créés.

   Un processeur dans l'état "PETITE CHARGE" peut accepter l'exécution des processus créés sur d'autres processeurs. Si son état est "GRANDE CHARGE", les processus créés doivent être transférés vers d'autres processeurs. L'état "CHARGE NORMALE" indique qu'aucun effort n'est utile pour le transfert des processus créés.

   Chaque processeur maintient l'état de ses voisins. A chaque changement de son état de charge, il diffuse à tous ses voisins son nouvel état. L'avantage de cette méthode est de pouvoir faire varier les paramètres Tmin et Tmax de manière à obtenir une période de transition d'état relativement grande. Ceci évite un flot important de diffusion dans le réseau de processeurs.

   A la création d'un processus, suivant la charge locale et celles des processeurs voisins, le processeur élu (pour exécuter ce processus) est obtenu suivant l'algorithme suivant :

```
Si Etat_processeur <> GRANDE_CHARGE
Alors
   execution locale du processus
Sinon
   Si Il existe un Processeur Voisin
      tel que Etat_processeeur_voisin = PETITE_CHARGE
   Alors
      Execution du processus sur le processeur voisin (*)
   Sinon
      Saturation Locale
   Finsi
Finsi
```

L'écroulement prévisible dans le cas (*) doit être évité. Une solution similaire à celle de [HCG*82] a été retenue ici.

Lorsque survient le cas d'une saturation locale, l'un des voisins est choisi aléatoirement ; une priorité est toutefois donnée au processeur dont l'état n'est pas "GRANDE CHARGE".

Nous avons uniquement présenté jusqu'ici le problème du choix d'un processeur pour la création d'un processus. L'étape suivante consiste à maintenir les voies de communication logiques entre le processus qui vient de migrer et les autres déjà actifs. Ceci n'est possible que si, sur chaque site, il existe un noyau de routage que les processus sont obligés d'utiliser pour communiquer entre eux. Ce noyau doit pouvoir permettre une identification logique des processus entre eux.

5.4 L'allocateur de PDS

Dans le cas de la chaîne de développement PDS, nous avons choisi le principe d'une allocation statique. Ce choix se justifie par une maturité plus importante des algorithmes d'allocation statique par rapport à ceux régulant dynamiquement la charge de processeurs.

Lors de la conception du module d'allocation, nous nous sommes fixé comme objectifs de pouvoir :

1. Utiliser plusieurs topologies du multiprocesseur lorsque celui-ci est capable de les supporter.
2. Avoir plusieurs algorithmes d'allocation possibles.

Ces deux objectifs ajoutent un niveau de complexité au problème de l'allocation. En effet, il convient de déterminer pour un ensemble de processus quel est le meilleur placement entre les $T \times A$ possibles, où T désigne le nombre de topologies possibles, et A le nombre d'algorithmes. Effectuer un tri entre tous les placements obtenus après allocation est réalisable si nous définissons un moyen de classement.

Un bon critère est de choisir le placement minimisant au mieux le temps global d'exécution des processus de la phase ; ceci peut être fait si nous disposons d'une fonction d'évaluation du temps global reflétant fidèlement la réalité. Pour le moment, nombreux sont les algorithmes d'allocation statiques qui prennent en compte une évolution globale du temps d'exécution. L'évaluateur reprendra ce type d'information en essayant de positionner chaque type d'évaluation par rapport aux autres.

Comme nous l'avons présenté au chapitre 4, le module d'extraction de PDS produit un programme présenté sous forme canonique. Le module d'allocation a pour tâche de placer les processus de chaque transition sur le multiprocesseur et son mode opératoire est fortement imprégné de cette forme canonique, comme le montre l'algorithme présenté en figure 5.2.

```
Pour chaque phase
    Pour chaque topologie T
        Pour chaque algorithme d'allocation
            1) executer algorithme d'allocation
            2) Evaluer le temps global d'execution
        Determiner le meilleur algorithme pour la topologie T
    Determiner la meilleure topologie T
```

Figure 5.2: Algorithme de l'allocateur

Les objectifs que nous sommes donnés, associés au modèle d'exécution, impliquent une structure de l'allocateur (c.f. figure 5.3) plus générale que celle des allocateurs disponibles actuellement. Nous y retrouvons la succession de transitions dont il faut placer les processus, et la nécessité de choisir un placement parmi une liste de placements obtenus en sortie d'algorithmes semblables à ceux décrits précédemment. Le lecteur trouvera une description détaillée des composantes du module d'allocation dans le rapport de thèse de M. Léon Mugwaneza (à paraître). Nous nous contentons seulement de présenter de manière succinte l'interface entre l'extracteur et l'allocateur.

5.4.1 Informations extraites pour l'allocation

Les informations que l'on doit extraire du programme parallèle sont:

- le temps d'exécution d'un processus.
- le coût de communication entre les paires de processus.

Comme présenté en début de ce chapitre, nous ne pouvons pas obtenir de valeurs exactes de ces données. Nous nous contenterons donc d'une extraction partielle des informations. Certains algorithmes modélisent le temps réel en utilisant une évaluation du nombre de boucles, de tests conditionnels, etc ... et de statistiques à propos du comportement de ces opérateurs [Kat83].

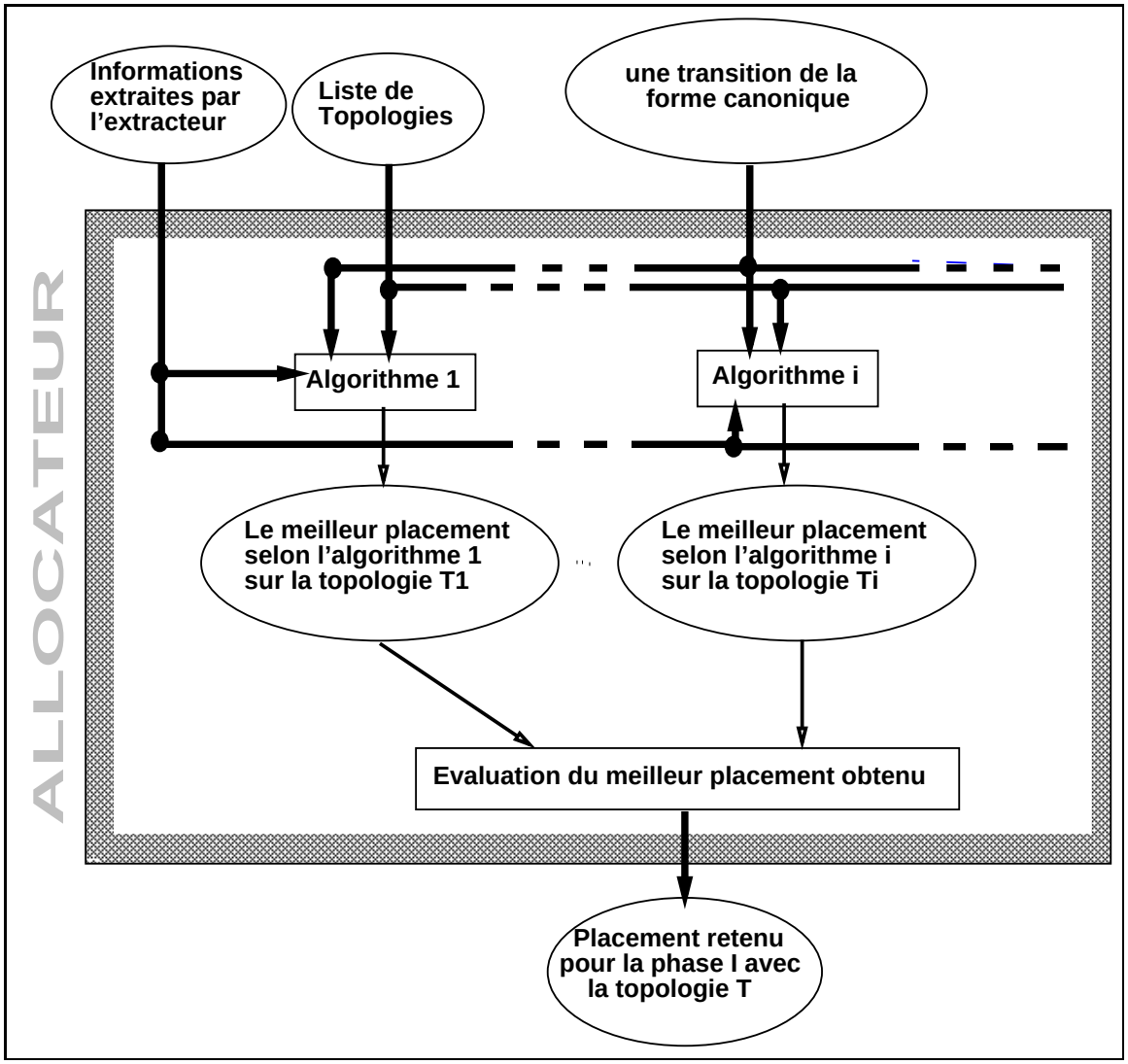


Figure 5.3: Structure de l'allocateur

L'extracteur de PDS produit seulement un ensemble d'informations qui permettront aux algorithmes d'allocation de faire leur modélisation :

- temps passé à exécuter du code utile (addition, ...)
- le nombre de WHILE, IF, ALT, ...
- le nombre de communications et leur volume entre chaque processus de la transition.
- d'autres informations à définir en fonction des algorithmes d'allocation utilisés.

Chapitre 6

Module de génération

Dans ce chapitre, nous présentons le système de génération élément final de la chaîne de développpement PDS. Ce module est particulièrement important puisqu'il doit être réécrit pour chaque multiprocesseur. Cette réécriture est du ressort d'un spécialiste du multiprocesseur en question, de manière à choisir les options adaptées aux caractéristiques de la machine si l'on veut obtenir les meilleures performances possibles.

Dans un premier temps, nous présentons l'impact du travail effectué par le module d'allocation sur la compilation des instructions PVM. Dans la suite du chapitre, nous présentons la structure d'un générateur dédié à une machine supernode.

6.1 Influence du module d'allocation

Le module d'allocation produit, pour chaque transition de la forme canonique, un placement des processus sur les processeurs interconnectés suivant une topologie T (c.f. Chapitre 5). Ce placement affecte la compilation du code PVM en trois points :

1. la gestion des communications interprocessus.

2. la gestion du cache de la mémoire virtuelle.

3. l'enchaînement des transitions.

6.1.1 Impact sur les instructions PVM liées aux communications interprocessus

Le fait que le module d'allocation puisse regrouper des processus sur un même processeur a de fortes implications sur la compilation des instructions créer_lien(), détruire_lien(), envoyer_synchrone(), recevoir_synchrone(), envoyer_asynchrone(), recevoir_asynchrone(), test_msg_émis().

En effet, soient deux processus Pj et Pk qui communiquent entre eux par un lien virtuel C_{ik}. S'ils sont placés sur le même processeur, la communication doit rester interne. Dans le cas contraire, se posent les problèmes suivants :

- routage des messages entre les processeurs.
- multiplexage/démultiplexage des voies de communication.
- assurer par logiciel la synchronisation associée aux primitives envoyer_synchrone() et recevoir_synchrone(), lorsque celle-ci n'est pas directement assurée par le matériel.

6.1.2 Impact sur les instructions PVM assurant l'accès à la mémoire virtuelle

Nous avons présenté au chapitre 3 section 3.3.3, une gestion par cache des données globales accédées par un processus. Nous avions prévu un cache local par processus.

Lorsque des processus sont regroupés sur un même processeur, nous pouvons envisager un regroupement de ces caches en un cache commun à tous les processus chargés sur ce processeur. La cohérence en est assurée lors de la compilation pour les accès de type statique[1], ou lors de l'exécution pour les accès de type dynamique[2].

Le seul cas où plusieurs processus peuvent partager une donnée, est celui où l'accès se fait en lecture seule. Considérons une donnée D, accédée par deux processus Pj et Pk placés sur le même processeur. Deux cas sont possibles :

1. Pk effectue un accès du type accès statique. Nous pouvons éventuellement ne pas interroger systématiquement la mémoire virtuelle.

[1]Rappel : Accès Connu lors de la Compilation
[2]Rappel : Accès Connu lors de l'Exécution

(a) Si le cache commun du processeur contient la valeur de la donnée D, il n'est pas nécessaire d'émettre une requête vers la mémoire virtuelle.

(b) Dans le cas contraire, nous nous trouvons dans la situation du premier accès de ce processeur à la donnée D. Pk émet sa requête, la mémoire virtuelle lui retourne la valeur et le cache est mis à jour.

2. Pk effectue un accès du type accès dynamique.

(a) Si la valeur de D est absente du cache, la requête est émise vers la mémoire qui effectue les contrôles adéquats. S'il n'y a pas eu violation des règles de cohérences d'accès, la valeur est transmise à Pk et introduite dans le cache commun.

(b) Si la valeur de D est présente dans le cache, il convient de s'assurer de la cohérence de l'accès en interrogeant la mémoire virtuelle. Le compte rendu obtenu doit seulement signifier si l'accès est valide ou non. Dans l'affirmative, Pk peut utiliser la valeur présente dans le cache commun.

6.1.3 Impact sur les instructions de séquencement

Le regroupement de plusieurs processus sur un même processeur et la présence d'une seule interface sur la voie de contrôle, impliquent la nécessité d'un regroupement des instructions de séquencement des processus chargés sur le même transputer. Ceci est facilement réalisable sur le transputer qui offre un mécanisme de création de processus avec terminaison synchrone à l'aide des instructions `startp` et `endp`. Le processeur sur lequel on veut placer plusieurs tâches doit donc exécuter un programme qui :

1. attend le signal "*début de phase*" que doit émettre le contrôleur,

2. active les processus de l'application,

3. attend la fin de leur exécution,

4. puis émet le signal "*travail terminé*" vers le contrôleur.

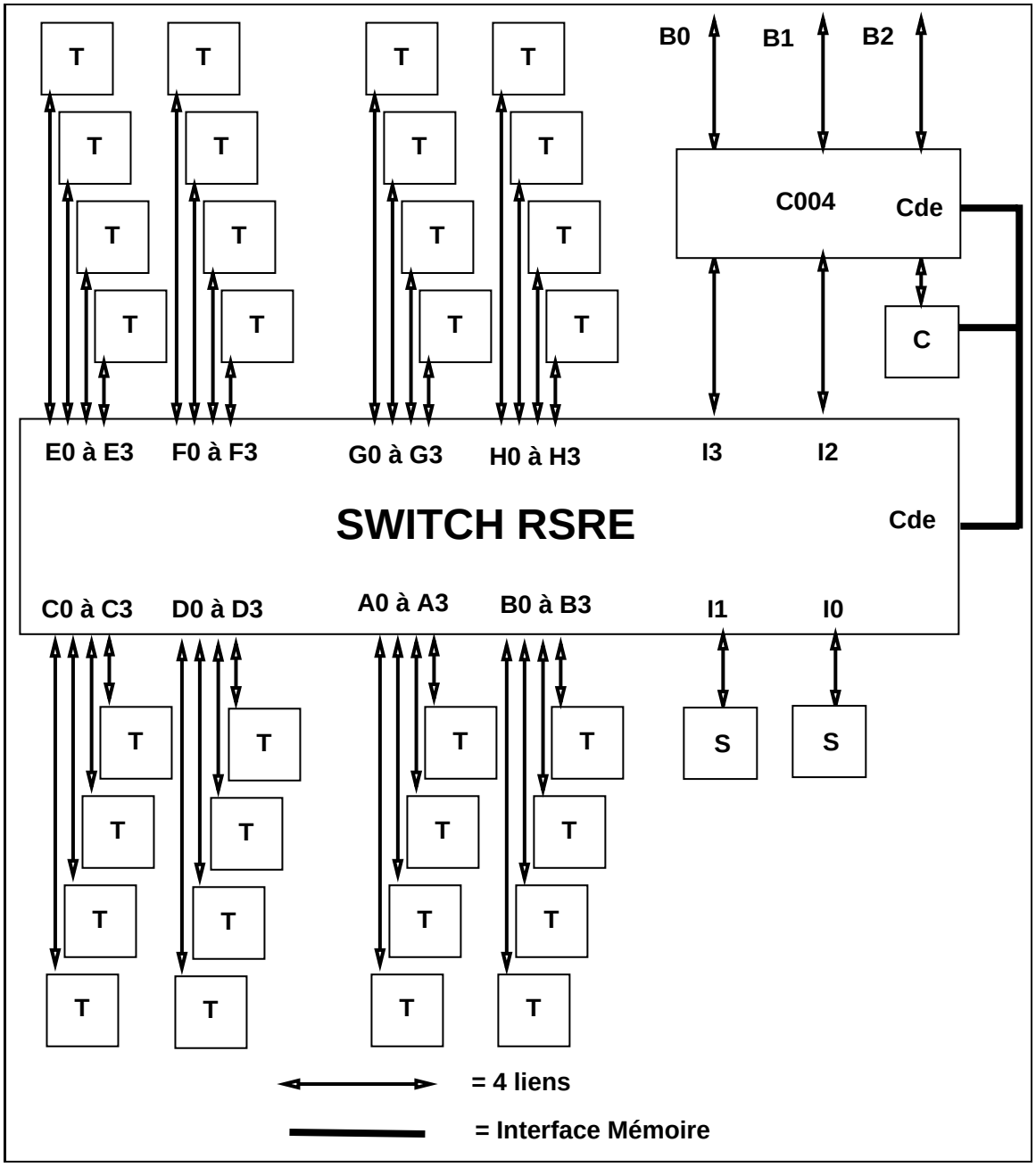


Figure 6.1: Structure Supernode à 32 transputers

6.2 La machine supernode à 32 transputers

Le supernode à 32 processeurs (c.f. figure 6.1) auquel nous nous intéressons, est composé de :

1. 32 transputers autonomes (ayant 2Mo de mémoire locale) (**T**).
2. un transputer pour contrôler la machine (**C**).
3. un système d'interconnection de liens ; le commutateur de RSRE.
4. deux transputers spécialisés (**S**) offrant respectivement un espace mémoire important et/ou un accès disque. Ces transputers ont 16Mo de mémoire locale.
5. un commutateur C004 qui permet d'accéder à des liens câblés vers l'extérieur.

Dans un premier temps, nous allons décrire les fonctionnalités du transputer, puis nous détaillerons le système de reconfiguration ainsi que la voie de contrôle.

6.2.1 Le transputer

Le terme transputer désigne toute une famille de processeurs INMOS intégrant dans le même circuit les éléments suivants :

- un processeur
- un co-processeur flottant
- quatre co-processeurs de communication
- une mémoire locale.

La structure générale d'un transputer est présentée en figure 6.2.

Les liens pilotés par les co-processeurs de communication permettent une communication directe entre des processeurs de la même famille. Cette communication est de type synchrone, octet par octet. Un tampon d'un seul octet est donc nécessaire sur le transputer destinataire pour garantir qu'aucune information n'est perdue. La gestion de ce tampon est réalisée au niveau matériel et par conséquent inaccessible au programmeur.

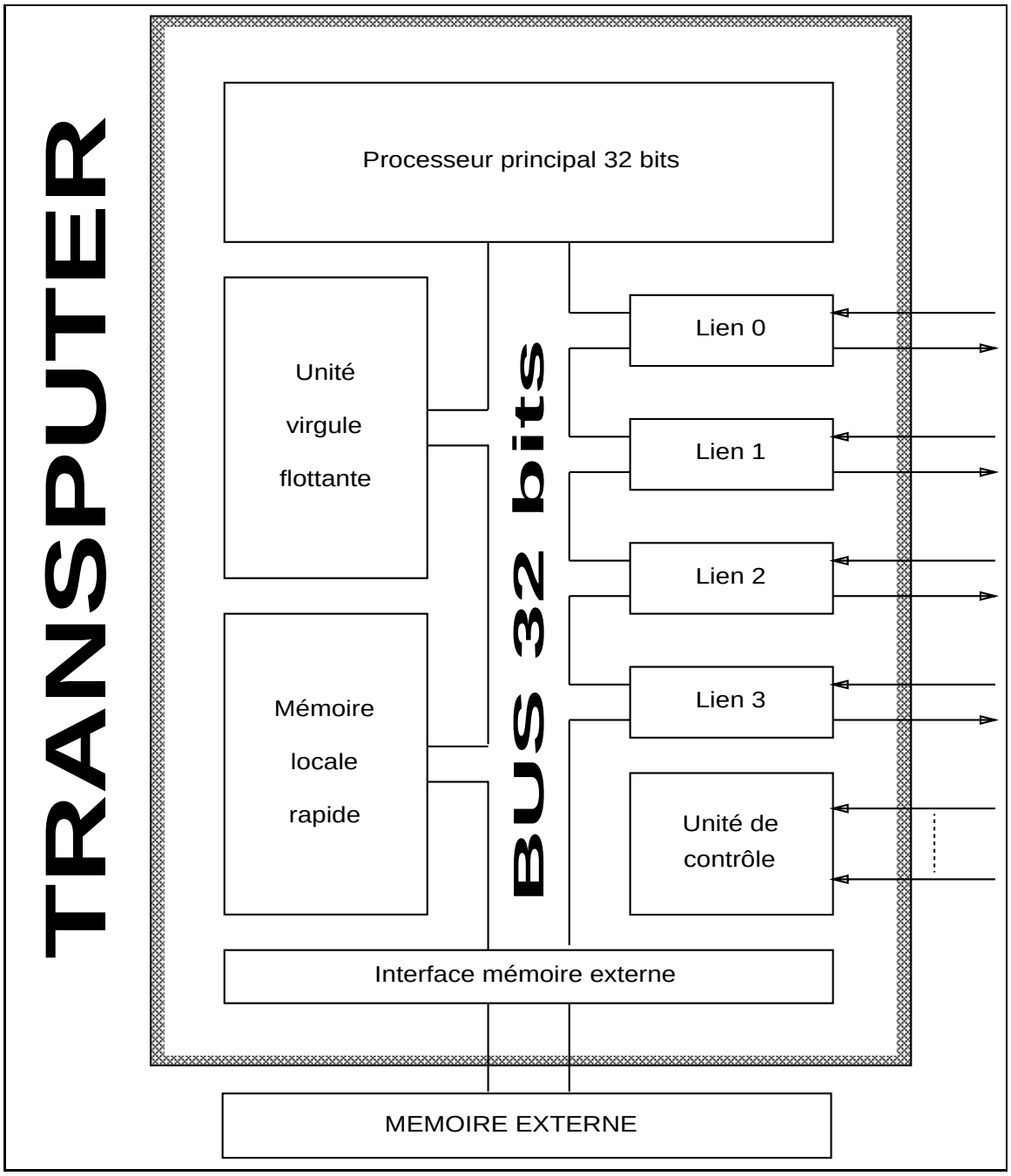


Figure 6.2: Structure d'un transputer

Propriété intéressante, le jeu d'instructions du transputer gère les communications aussi bien internes qu'externes grâce aux mêmes instructions machines. Ceci sera utile lorsque plusieurs processus seront placés sur le même processeur.

Une autre caractéristique du transputer est la gestion directe de la notion de processus au niveau du processeur interne. Ainsi un ordonnanceur de processus est intégré directement dans le microcode, manipulant deux niveaux de priorité de processus .

Pour de plus amples informations sur le transputer, le lecteur est invité à consulter [Lim87] pour la partie *matérielle* et [Eud87,She87,Lim86] pour la partie programmation.

6.2.2 Le système d'interconnexion des liens

Le système d'interconnexion des liens a été conçu de manière à permettre une modification dynamique des connexions.

Le cœur de ce système est une paire de circuits réalisant chacun la commutation de 72×72 liens. Ces deux circuits, dont la conception a été effectuée par RSRE, sont désignés par la suite sous le terme de *commutateur*. La fonction réalisée par ce commutateur est de permettre une modification de deux connexions (4 liens) sans altérer les communications en cours sur les autres liens déjà connectés par le commutateur. Celui-ci est piloté par un transputer particulier, le contrôleur.

De manière à pouvoir communiquer avec l'extérieur, un commutateur C004 [Hil87] a été inséré dans la machine. En plus des possibilités de synchronisation et d'amplification de signaux électriques, le C004 permet de commuter des liens. Dans le multiprocesseur Supernode à 32 transputers, il permet d'interconnecter les 4 liens du transputeur de contrôle, 2 groupes de quatre liens directement sur le commutateur, 3 groupes de 4 liens tirés vers l'extérieur de la machine, et 8 connexions externes à la norme ITEM. Ceci permet à n'importe quel transputer de pouvoir communiquer des informations vers l'extérieur moyennant une reconfiguration.

Le C004 et le commutateur de RSRE sont pilotés par le transputer de contrôle à travers une interface mémoire qui permet de voir les registres de contrôle de ces circuits comme des mots mémoires.

6.2.3 La voie de contrôle

La voie de contrôle permet de gérer la reconfiguration. En effet, le commutateur est piloté par le transputer de contrôle à l'initiative éventuelle des transputers de travail qui lui soumettent leurs requêtes de reconfiguration. Ce flux de communication et de synchronisation doit être supporté par une voie de communication directe entre le contrôleur et les processeurs de travail.

De plus, il est nécessaire de gérer individuellement les signaux de service de chaque transputer (Reset, Analyse,...)

Dans ce multiprocesseur, la voie de contrôle se présente sous la forme d'un bus parallèle à huit bits de type maître-esclave. Le contrôleur est le maître et les processeurs de travail, les esclaves. Les processeurs connectés sur ce bus ont chacun une interface qui pilote les signaux de service et dont les registres sont accessibles dans l'espace d'adressage du processeur.

Ce bus permet de réaliser deux types de dialogue entre le maître et les esclaves :

1. communication d'un octet d'un esclave vers le maître
2. diffusion d'un octet du maître vers un sous-ensemble d'esclaves.

Ces deux dialogues réalisent à la fois une synchronisation et un transfert d'informations entre le maître et le ou les esclaves.

Le transfert d'informations se fait toujours à l'initiative du maître. Il est donc indispensable d'avoir un mécanisme permettant aux esclaves de signaler au maître la nécessité d'un transfert d'informations. C'est toujours le maître qui ordonnera le transfert d'informations. Pour ce faire, le bus de contrôle offre deux mécanismes de synchronisation :

1. Une synchronisation OU permettant le déclenchement d'un signal vers le maître dès lors qu'un esclave en fait la demande.
2. Une synchronisation ET permettant de réaliser un rendez-vous entre un ensemble d'esclaves. Le signal n'est émis que si tous les esclaves de l'ensemble en font la demande.

Ce signal est émis sur la broche "event" du transputer de contrôle. Au cas où celui-ci transfère un octet vers une interface esclave, il est nécessaire de pouvoir interrompre le travail en cours sur le processeur esclave. Ceci est

réalisé par l'émission d'un signal sur la broche "event" du processeur esclave dès qu'il y a écriture de l'octet dans l'interface.

Toute communication sur ce bus met en jeu les trois entités suivantes :

- Le producteur
- Le consommateur
- Une boîte aux lettres dont la taille se réduit ici à un octet. Cette boîte est présente sur chaque processeur esclave.

Tout transfert d'octet utilise ici un protocole du type producteur-consommateur :

1. Dépôt de l'octet dans la boîte aux lettres par le producteur.
2. Emission d'un signal pour avertir le ou les consommateurs.
3. Retrait du message de la boîte aux lettres par le ou les consommateurs.
4. Envoi d'un acquittement vers le producteur.

6.3 La machine virtuelle sur Supernode 32 Transputer

Nous allons détailler ici la construction de la machine virtuelle au-dessus du multiprocesseur Supernode. Nous allons prendre chaque élément de la PVM et le placer sur un élément de Supernode.

1. Le contrôleur de la mémoire virtuelle gère l'enchaînement des phases de la forme canonique. Le seul processeur capable de ceci est le transputer de contrôle. En utilisant la voie de contrôle, celui-ci peut détecter la fin de l'exécution d'une transition. Une fois celle-ci effectuée, il peut reconfigurer la topologie du réseau en vue de l'exécution de la phase suivante.
2. Chaque processeur virtuel est placé sur un transputer de travail.
3. Le système de communication par messages inter-processus est implanté directement sur le commutateur de RSRE et les liens de chaque processeur. Un noyau de routage sera présent sur chaque processeur.

4. Pour ce qui est de la mémoire virtuelle, deux solutions sont possibles :

   (a) Utiliser les processeurs de services pour réaliser cette fonction de manière centralisée.

   (b) Répartir la mémoire virtuelle entre tous les processeurs de travail de la machine.

   La seconde solution est à l'étude et nous présentons ici une réalisation basée sur la première possibilité. Une évaluation de performances est à effectuer pour savoir si ce choix conduit à une solution efficace ou non.

5. Le système de communication par ports aurait pu être envisagé sur la voie de contrôle si celle-ci avait eu des performances honorables, ce qui n'est pas le cas ici (c.f. [Wai90]). Nous allons donc utiliser le même support de communication que celui du système de communication inter-processus : le commutateur de RSRE et les liens des transputers.

6.4 Générateur pour la machine Supernode

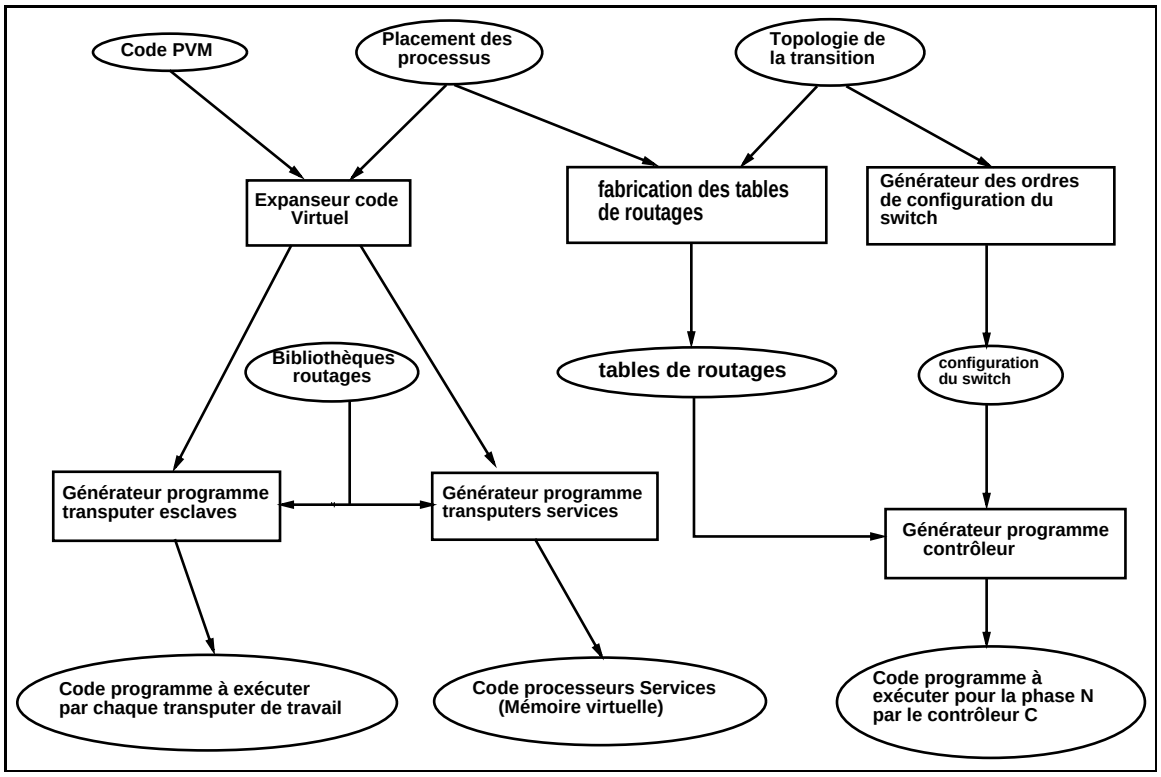


Figure 6.3: Structure du générateur pour supernode

Le comportement du générateur est répétitif sur le nombre de phases présentes dans la forme canonique. La figure 6.3 présente les différents modules de ce générateur :

1. Un **expanseur de code virtuel** traduit le code PVM en code assembleur au format PDS [Eud87]. Le principal travail de ce module est de générer les appels au système de routage qui assure une désignation logique des liens quel que soit l'endroit où se situe le correspondant (interne ou externe au transputer). Une fois cette traduction faite, le code du programme est découpé processeur par processeur, puis introduit dans le générateur de code des transputers de travail. Le second consiste à fabriquer les informations nécessaires pour pouvoir gérer correctement la mémoire virtuelle. Ces informations sont stockées dans des tables transmises au générateur associé au transputer de service.

2. Un **générateur de tables de routage** fabrique, en fonction de la topologie choisie du Supernode, les tables de routage à charger sur chaque

transputer. Nous utiliserons un algorithme sans interblocage tel celui décrit dans [SMT90]. A chaque changement de configuration, la topologie de la machine est modifiée. Il est donc nécessaire de changer le contenu de la table de routage sur chaque processeur de travail. Nous pouvons effectuer cette modification sans aucun problème puisqu'aucune communication inter-processus n'a lieu pendant le changement de configuration. Le calcul des tables de routages est effectué de manière statique étape par étape. Deux possibilités nous sont offertes quant à l'utilisation de ces tables :

(a) les inclure dans le code dédié aux processeurs de travail. Cette inclusion peut être constituée d'un ensemble de tables chargées sur chaque processeur, ou bien la table de routage initiale et la série de modifications à effectuer pour passer de l'étape i vers l'étape $i+1$.

(b) les regrouper sur le processeur de contrôle qui émettra vers les processus de travail les modifications propres à chaque processeur.

Nous avons retenu la seconde solution pour des raisons de gain d'espace mémoire sur les processeurs de travail, bien que cela fasse communiquer de manière directe le transputer de contrôle et le transputer disque (dans le cas d'une saturation de l'espace mémoire du transputer de contrôle).

3. Un **configureur** génère les ordres à envoyer au commutateur RSRE pour obtenir la topologie sélectionnée. [PAC89] contient une étude sur l'utilisation de la reconfiguration des machines Supernodes. En effet, sans entrer dans tous les détails de la conception de la machine que le lecteur trouvera dans [Wai90], il faut savoir que le commutateur de liens ne permet pas de réaliser toutes les connections possibles. Pour surmonter ce problème, [PAC89] propose un algorithme permettant de réaliser n'importe quelle topologie de degré inférieur ou égal à quatre. Nous utiliserons ici ce programme pour calculer les ordres à émettre au commutateur de manière à obtenir la topologie nécessaire à l'exécution d'une transition.

4. Un **"générateur" pour transputer de travail** récupère les sorties de l'expanseur et la bibliothèque de routage pour produire un binaire exécutable sur un transputer esclave. Chaque processeur est donc chargé avec le binaire correspondant à tout le code qu'il doit exécuter pendant la durée de vie de l'application. Le schéma d'exécution d'un processeur est le suivant :

(a) Mise en attente sur la réception de l'information de routage qui sera fournie par le processeur de contrôle.

(b) Une fois cette information reçue et la mise à jour de la table de routage interne effectuée, le processeur se met en attente du signal DEBUT_TACHE que doit émettre le transputer de contrôle signifiant que tous les processeurs de travail ont effectué la mise à jour de leurs tables de routage.

(c) Réception du Signal DEBUT_TACHE qui permet au code dédié à l'étape i de l'application d'être activé.

(d) Une fois que la partie application de la tâche est terminée sur ce processeur, celui-ci émet vers le processeur de contrôle via la voie de contrôle le signal FIN_TACHE.

(e) Retour à l'état (a) de manière à enchaîner vers la transition suivante.

5. Un **"générateur" pour transputer de service** utilise les tables produites par l'expanseur et produit un programme assembleur réalisant les vérifications présentées dans le chapitre 3. L'identification des informations concernant chaque variable est effectuée en utilisant le nom logique de la variable (rendu unique pour toute l'application) et des fonctions de "hash-code". Ceci nous permet de garantir un accès rapide à l'information. D'autre part, la programmation de cette partie est particulièrement soignée vu qu'il faut répondre au processeur de travail le plus rapidement possible ; pour cette raison, nous avons choisi de la développer en assembleur.

6. Un **"générateur" pour transputer de contrôle** génère un programme qui charge les tables de routage sur les processeurs esclaves, configure le commutateur de RSRE, effectue l'initialisation des processeurs de travail, et se met en attente de la terminaison des travaux de tous les processeurs de traitement.

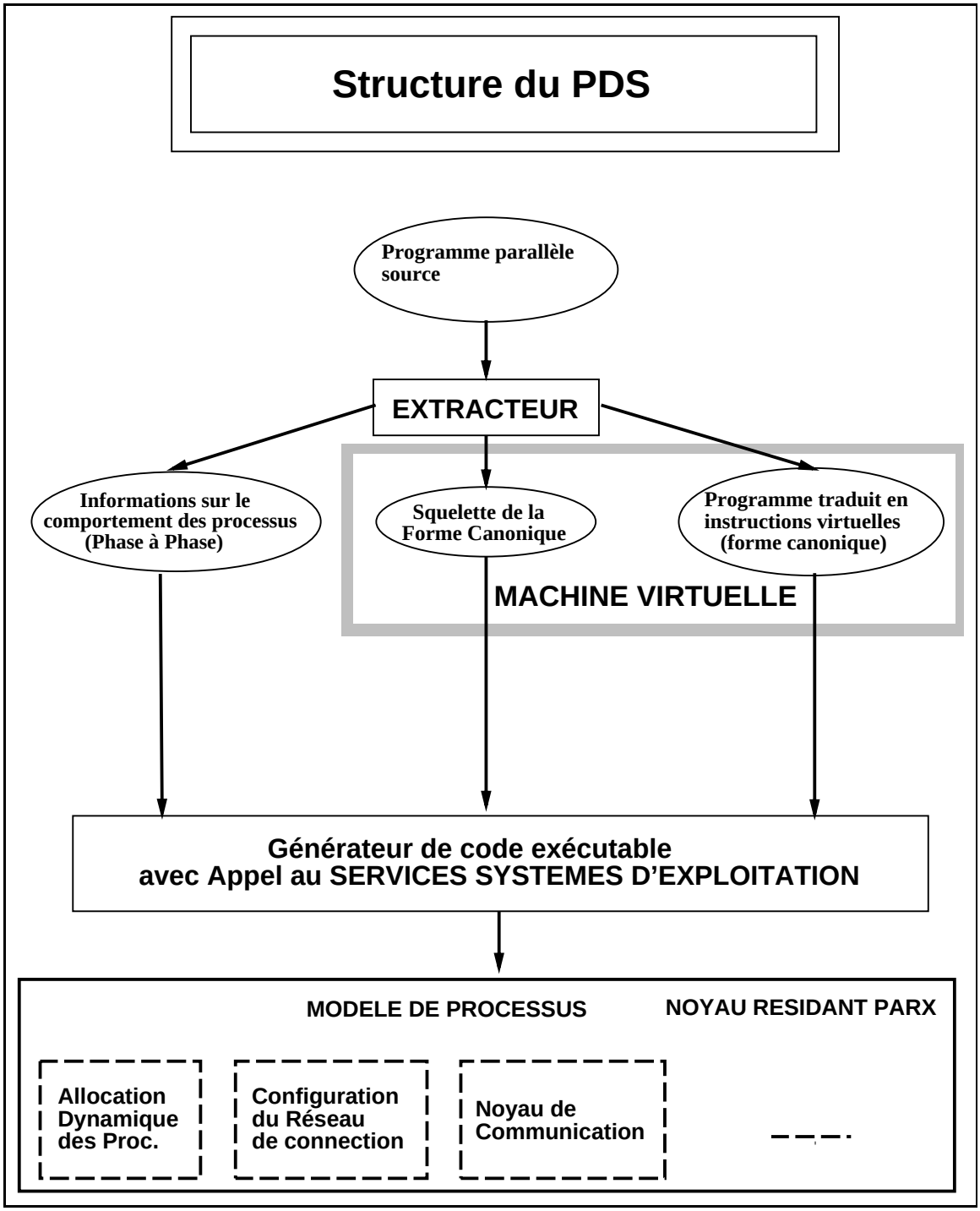


Figure 6.4: Structure de PDS utilisant PARX

Chapitre 7

Conclusions et Perspectives

Dans le chapitre 2, nous avons tiré profit des lacunes des systèmes de développement présents dans le monde industriel, pour proposer un générateur de systèmes de développement de programmes parallèles (PDS) n'ayant pas ces inconvénients. Nous avons abordé dans cette thèse, une structure de PDS dédiée à une utilisation "Load and Go" des machines parallèles. Actuellement, des systèmes d'exploitation pour de telles architectures sont en cours d'étude et des réalisations commencent à voir le jour.

L'intégration de PDS dans un système d'exploitation tel que celui dédié aux architectures Supernodes (PARX) actuellement à l'étude dans le projet ESPRIT "Supernode II", permettrait d'obtenir une chaîne de développement dédiée à une utilisation multi-usagers d'une même machine.

Ceci implique au niveau du système d'exploitation de ces architectures, de nouvelles fonctionnalités spécifiques à l'exploitation du parallélisme comme une allocation dynamique des processus sur les processeurs, une génération automatique de tables de routages, etc ...

La structure de PDS va donc évoluer vers celle présentée en figure 6.4.

Le module d'allocation statique disparait pour laisser place à des appels au service d'allocation dynamique. De même, le module de génération de code, utilisera lui aussi des appels au système d'exploitation pour réaliser les tâches de configuration de la machine, de routage entre les différents processus,...

Il faut remarquer que la machine virtuelle PVM décrite dans cette thèse est reprise intégralement dans le cadre de cette nouvelle chaîne PDS, ce qui permet de réutiliser le travail réalisé au niveau de l'extraction du parallélisme et de maintenir un niveau de compatibilité au niveau utilisateur entre les deux

versions de PDS.

En conclusion, par conséquent, la philosophie de PDS est de permettre à l'utilisateur de s'affranchir des contraintes spécifiques à son architecture. Notre but est fournir ce type d'outil pour des architectures à topologie d'interconnexion reconfigurable par programme comme celle de la famille Supernode.

Bibliographie

[AL85] E.H.L. Aarts and P.J.M. Van Laarhoven. Statistical cooling: a general approach to combinatorial optimization problems. *Philips Journal of Research*, Volume 40(Numéro 4):pp. 193–226, 1985.

[Amd67] Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Procedings*, Volume 30, 1967.

[Ath87] W.C. Athas. *Fine Grain Concurrent Computation.* Technical Report 5242:TR:87, Caltech Computer Science, Pasadena, CA 91125, 1 Mai 1987.

[BCE*88] P. Beskeen, N. Clifton, A. England, A. Evans, N.Garnett, C. Grimsdale, T. King, J. King, and B. Veer. *Helios PC Software Development System.* Perihelion Software Limited, 24/25 Brewmaster Buildings,Charlton Trading Estate Shepton Mallet, Somerset,BA4 5QE, Août 1988.

[BR89] L. BOMANS and D. ROOSE. Benchmarking the ipsc/2 hypercube multiprocessor. *Concurrency: Practice and Exprerience*, Volume 1:4–18, Septembre 1989.

[CK79] Y.C. Chow and W.H. Kohler. Models for dynamic load balancing in heteregenous multiple processor systems. *IEEE Trans. on Computers*, Volume 28, Mai 1979.

[EKN88] A. Evans, J. King, and B. Noble. *Helios C Manual.* Perihelion Software Limited, 24/25 Brewmaster Buildings,Charlton Trading Estate Shepton Mallet, Somerset,BA4 5QE, Juin 1988.

[ELG89] T. EL-GHAZALI. *Allocation dynamique de processus sur une architecture parallèle.* Technical Report, Rapport DEA I.N.P.G., GRENOBLE, 1989.

[ELJ84] D.L. Eager, E.D. Lazowska, and L.H. Jamieson. Adaptative load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, Volume 12(Numéro 5), Mai 1984.

[Eud87] J. Eudes. *Assembleur PDS : Manuel de référence.* LGI-IMAG, Rapport Esprit P 1085, 1987.

[Gar87] N.H. Garnett. Helios - an operating system for the transputer. In 7_{th} *Occam User Group*, pages 411–419, Amsterdam Springfiled, VA, 14-16 Septembre 1987.

[GJ79] M.R. Garey and D.S. Johson. *Computers and Intractability: A Guide of the Theory of NP-Completeness.* 1979.

[Gol87] M. Goldsmith. Occam transformation at oxford. In 7_{th} *Occam User Group*, pages 37–54, Amsterdam Springfiled, VA, 14-16 Septembre 1987.

[HCG*82] K. Hwang, W.J. Croft, G.H. Goble, B.W. Wah, F.A. Briggs, W.R. Simmons, and C.L. Coates. A unix-based local computer network with load balencing. *IEEE Trans. on Computers*, Avril 1982.

[Hil87] G. Hill. *Tecnical note 19: Designs and Applications for the IMS C004.* INMOS Limited, Juin 1987.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, Volume 21:666–677, 1978.

[HR73] L. Hyafil and R.L. Rivest. *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems.* Technical Report 33, INRIA, Rocquencourt, France, 1973.

[Int88] Intel. *iPSC/2 - C Programmer's Reference Manual.* Intel Scientific Computers, Beaverton, Août 1988.

[Int89] Intel. *iPSC/2 - User's Guide.* Intel Scientific Computers, Beaverton, Mars 1989.

[JOS86] J.W.Flower, S.W. Otto, and M.C. Salama. *A Preprocessor for Irregular Finite Element Problems.* 1986.

[Kat83] M.G.H. Katevenis. *Reduced instruction set computer architectures for VSLI.* PhD thesis, University of California, Berkeley, Ca 94720, 1983.

[KCGV83] S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, Volume 220(Numéro 4598):pp. 671–680, Mai 1983.

[KT88] M. Kallstrom and S.S. Thakkar. Programming three parallel computers. *IEEE Software*, 11–22, Janvier 1988.

[Lim86] INMOS Limited. *T2/T4 transputer instruction set.* INMOS Limited, Juillet 1986.

[Lim87] *Transputer architecture.* INMOS Limited, Juillet 1987.

[LMM85] O. Lubeck, J. Moore, and R. Mendez. A benchmark comparison of three supercomputers: fujitsu vp-200, hitachi s-810/20, and cray x-mo/2. *IEEE Computer*, Volume 18(Numéro 12):10–23, Décembre 1985.

[MRRT53] N. Metropolis, N. Rosenbluth, A. Rosenbluth, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, Volume 21(Numéro 6):p. 1087, Juin 1953.

[Occ87a] *Occam 2 Toolset (Getting started).* INMOS Limited, 25 Septembre 1987.

[Occ87b] *Occam 2 Toolset (Occam 2 implementation).* INMOS Limited, 25 Septembre 1987.

[Occ87c] *Occam 2 Toolset (Occam standard libraries).* INMOS Limited, 25 Septembre 1987.

[Occ87d] *Occam 2 Toolset (User Manual).* INMOS Limited, 25 Septembre 1987.

[OI88] H. Ohara and H. Iizuka. A preprocessor to augment the description of occam processes. In *9_{th} Occam User Group*, pages 71–80, Amsterdam Springfiled, VA, 19-21 Septembre 1988.

[PAC89] F. PACULL. *Communication et reconfiguration dans les systèmes dynamiques communicants.* Technical Report, Rapport DEA I.N.P.G., GRENOBLE, Juin 1989.

[Pad88] S.A. Green D.J. Paddon. An extension of the processor farm using a tree architecture. In *9_{th} Occam User Group*, pages 53–69, Amsterdam Springfiled, VA, 19-21 Septembre 1988.

[PH88] *occam 2 - Reference Manual.* INMOS Limited, 1988.

[RH86] A. W. Roscoe and C. A. R. Hoare. *The Laws of Occam Programming.* Technical Report PRG-53, Oxford University Computing Laboratory, Oxford, OX1 3QD, Février 1986.

[Sei85] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, Volume 28(Numéro 1):22–33, Janvier 1985.

[She87] D. Shepherd. *Compiler Writer's Guide.* INMOS Limited, Janvier 1987.

[Smi88] R.G. Smith. *The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver.* 1988.

[SMT90] I. Sahko, L. Mugwaneza, and T.Muntean. *A constant buffering space method for deadlock-free routaging.* Technical Report (à paraître), Laboratoire de Génie Informatique - IMAG, 1990.

[SS84] J.A. Stankovic and I.S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings 4^{th} Conference on Distributed Computing Systems*, Mai 1984.

[SSS88] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. *The C Programmer's Abbreviated Guide to Multicomputer Programming.* Technical Report Caltech-CS-TR-88-1, Caltech Computer Science, Pasadena, CA 91125, 19 Janvier 1988.

[TDS86] *TDS Compiler implementation manual.* INMOS Limited, 19 Novembre 1986.

[Tra88] *Transputer Development System 3.0 (Draft Version).* INMOS Limited, Mars 1988.

[Wai90] P. Waille. *Introduction à l'architecture des machines Supernodes.* Technical Report (à paraître), Laboratoire de Génie Informatique - IMAG, 1990.

[WM85] Y.T Wang and J.T. Morris. Load sharing in distributed systems. *IEEE Trans. on Computers*, Volume C-34(Numéro 2), Mars 1985.

Sommaire

Liste des Figures

Liste des Tables